П.А. БАРАНЧИКОВ
И.В. БАРИНОВ
А.Н. КОРОТАЕВ

ОПЕРАЦИОННЫЕ СИСТЕМЫ

УЧЕБНИК

Рекомендовано
Научно-методическим советом «РГРТУ» в качестве учебника для студентов высших учебных заведений, обучающихся по направлению подготовки 2.09.03.03 «Прикладная информатика» (квалификация «бакалавр»)

Москва
КУРС
2018

УДК 004.451(075.8)
ББК 3973.2-018.2я73-1
Б24

Рецензенты:
Пылькин А.Н., доктор технических наук, профессор, заведующий кафедрой «Вычислительная и прикладная математика» ФГБОУ ВО «РГРТУ»;
Кираковский В.В., кандидат технических наук, доцент, генеральный директор проектного института «Промгражданпроект» (г. Рязань)

Баранчиков П.А.,
Б24 Операционные системы : учебник / П.А. Баранчиков, И.В. Баринов, А.Н. Коротаев. — М.: КУРС, 2018. — 288 с.

ISBN 978-5-906923-86-8 (КУРС)

В учебнике рассмотрены основные вопросы, необходимые для понимания функционирования операционных систем, изучаемые бакалаврами направления «Прикладная информатика». Далее рассматриваются вопросы, связанные с памятью компьютера, схемами ее управления, виртуальной памятью. Также освещены вопросы, касающиеся файловых систем и устройств ввода-вывода.

Отдельно рассмотрены вопросы безопасности в операционной системе, в числе которых были затронуты вопросы криптографии и средства аутентификации.

УДК 004.451(075.8)
ББК 3973.2-018.2я73-1

КУРС INTERACTIVE ON-LINE

ISBN 978-5-906923-86-8 (КУРС)

ВВЕДЕНИЕ

При подготовке бакалавров по направлению 09.03.04 «Программная инженерия» требуется дать представление будущим выпускникам о внутреннем устройстве и механизмах, использующихся в операционных системах, для последующего применения в учебной и практической деятельности. Поэтому целью данного учебника является ознакомление читателей: с базовыми принципами создания операционных систем, основными моделями представления данных, базовой архитектурой компьютерных систем, принципами управления и организации памяти; с принципами распределения ресурсов вычислительных систем, сервисных служб операционных систем, программных пактов, обслуживающих операционные системы; с принципами организации, создания и особенностями эксплуатации операционных оболочек; с организацией сохранности и защиты программных систем.

Материалы в книге изложены в 16 главах.

В первой главе приводится описание эволюции вычислительных систем, указываются основные функции операционных систем и принципы их построения.

Во второй главе объясняется понятие процессов в операционных системах, указаны их состояния и операции, которые можно произвести над ними.

В третьей главе объясняются уровни, критерии, параметры и алгоритмы планирования процессов.

В четвертой главе описываются критические секции процессов, взаимоисключения и организация правильной очередности.

В пятой главе рассматриваются алгоритмы синхронизации процессов.

В шестой главе рассмотрены семафоры, мониторы, сообщения и доказывается их эквивалентность.

В седьмой главе рассматриваются условия возникновения тупиков, основные направления борьбы с ними, а также способы их предотвращения.

В восьмой главе рассматривается организация памяти в компьютере, обьясняется страничная схема организация памяти, а также сегментная и сегментно-страничная организации.

В девятой главе рассматривается виртуальная память, архитектурные средства ее поддержки, а также аппаратно-независимый уровень ее управления.

В десятой главе рассматриваются файловые системы с точки зрения пользователя, организация файлов и доступ к ним, операции над файлами, логическая структура файлового архива, операции над директориями и защита файлов.

В одиннадцатой главе рассматривается общая структура файловой системы, управление внешней памятью, как реализованы директории, как происходит монтирование файловых систем и связывание файлов.

В двенадцатой главе рассматривается система управления вводом-выводом, а также физические принципы организации ввода-вывода.

В тринадцатой главе рассматриваются основные понятия информационной безопасности в операционных системах, приводятся основные угрозы безопасности и формализуется подход к обеспечению информационной безопасности.

В четырнадцатой главе рассматривается механизм использования криптоалгоритмов для защиты операционной системы.

В пятнадцатой главе рассматриваются понятия идентификации, аутентификации, авторизации, а также приводится механизм выявления вторжений в операционную систему.

В шестнадцатой главе рассматриваются механизмы межпроцессного взаимодействия, такие как сигналы, протокол TCP/IP, система D-Bus и сокеты Беркли.

Глава 1
ВВЕДЕНИЕ. ЭВОЛЮЦИЯ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ, ОСНОВНЫЕ ФУНКЦИИ ОПЕРАЦИОННЫХ СИСТЕМ И ПРИНЦИПЫ ИХ ПОСТРОЕНИЯ

Из чего состоит любая вычислительная система? Во-первых, из того, что в англоязычных странах принято называть словом hardware, или аппаратное обеспечение: процессор, память, монитор, дисковые устройства и т.д., объединенные магистральным соединением, которое называется шиной.

Во-вторых, вычислительная система состоит из программного обеспечения. Все программное обеспечение принято делить на две части: прикладное и системное. К прикладному программному обеспечению, как правило, относятся разнообразные банковские и прочие бизнес-программы, игры, текстовые процессоры и т.п. Под системным программным обеспечением обычно понимают программы, способствующие функционированию и разработке прикладных программ. Надо сказать, что деление на прикладное и системное программное обеспечение является отчасти условным и зависит от того, кто осуществляет такое деление. Так, обычный пользователь, неискушенный в программировании, может считать Microsoft Word системной программой, а с точки зрения программиста это — приложение. Компилятор языка Си для обычного программиста — системная программа, а для системного — прикладная. Несмотря на эту нечеткую грань, данную ситуацию можно отобразить в виде последовательности слоев (рис. 1.1), выделив отдельно наиболее общую часть системного программного обеспечения — операционную систему.

1.1. Что такое операционная система

Большинство пользователей имеет опыт эксплуатации операционных систем, но, тем не менее, они затруднятся дать этому понятию точное определение. Давайте кратко рассмотрим основные точки зрения.

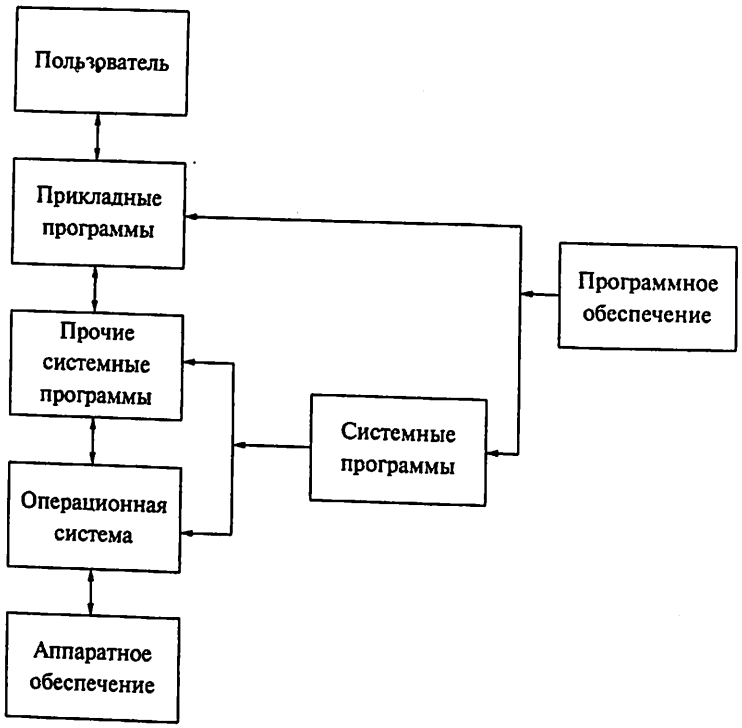

Рис. 1.1. Слои программного обеспечения компьютерной системы

Операционная система как виртуальная машина

При разработке операционных систем широко применяется абстрагирование, которое является важным методом упрощения и позволяет сконцентрироваться на взаимодействии высокоуровневых компонентов системы, игнорируя детали их реализации. В этом смысле ОС представляет собой интерфейс между пользователем и компьютером.

Архитектура большинства компьютеров на уровне машинных команд очень неудобна для использования прикладными программами. Например, работа с диском предполагает знание внутреннего устройства его электронного компонента — контроллера для ввода команд вращения диска, поиска и форматирования дорожек, чтения и записи секторов и т.д. Ясно, что средний программист не в состоянии учитывать все особенности работы оборудования (в современной терминологии — заниматься разработкой драйверов устройств), а должен иметь простую высокоуровневую абстракцию, скажем, представляя информационное пространство диска как набор

файлов. Файл можно открывать для чтения или записи, использовать для получения или сброса информации, а потом закрывать. Это концептуально проще, чем заботиться о деталях перемещения головок дисков или организации работы мотора. Аналогичным образом, с помощью простых и ясных абстракций, скрываются от программиста все ненужные подробности организации прерываний, работы таймера, управления памятью и т.д. Более того, на современных вычислительных комплексах можно создать иллюзию неограниченного размера оперативной памяти и числа процессоров.

Всем этим занимается операционная система. Таким образом, операционная система представляется пользователю виртуальной машиной, с которой проще иметь дело, чем непосредственно с оборудованием компьютера.

Операционная система как менеджер ресурсов

Операционная система предназначена для управления всеми частями весьма сложной архитектуры компьютера. Представим, к примеру, что произойдет, если несколько программ, работающих на одном компьютере, будут пытаться одновременно осуществлять вывод на принтер. Мы получили бы набор строчек и страниц, выведенных различными программами. Операционная система предотвращает такого рода хаос за счет буферизации информации, предназначенной для печати, на диске и организации очереди на печать. Для многопользовательских компьютеров необходимость управления ресурсами и их защиты еще более очевидна. Следовательно, операционная система, как менеджер ресурсов, осуществляет упорядоченное и контролируемое распределение процессоров, памяти и других ресурсов между различными программами.

Операционная система как защитник пользователей и программ

Если вычислительная система допускает совместную работу нескольких пользователей, то возникает проблема организации их безопасной деятельности. Необходимо обеспечить сохранность информации на диске, чтобы никто не мог удалить или повредить чужие файлы. Нельзя разрешить программам одних пользователей произвольно вмешиваться в работу программ других пользователей. Нужно пресекать попытки несанкционированного использования вычислительной системы. Всю эту деятельность осуществляет операционная система как организатор безопасной работы пользователей и их программ. С такой точки зрения операционная система представля-

ется системой безопасности государства, на которую возложены полицейские и контрразведывательные функции.

Операционная система как постоянно функционирующее ядро

Наконец, можно дать и такое определение: операционная система — это программа, постоянно работающая на компьютере и взаимодействующая со всеми прикладными программами. Казалось бы, это абсолютно правильное определение, но, как мы увидим дальше, во многих современных операционных системах постоянно работает на компьютере лишь часть операционной системы, которую принято называть ее ядром.

Как мы видим, существует много точек зрения на то, что такое операционная система. Невозможно дать ей адекватное строгое определение. Нам проще сказать не чт.е. операционная система, а для чего она нужна и что она делает. Для выяснения этого вопроса рассмотрим историю развития вычислительных систем.

1.2. Краткая история эволюции вычислительных систем

Мы будем рассматривать историю развития именно вычислительных, а не операционных систем, потому что hardware и программное обеспечение эволюционировали совместно, оказывая взаимное влияние друг на друга. Появление новых технических возможностей приводило к прорыву в области создания удобных, эффективных и безопасных программ, а свежие идеи в программной области стимулировали поиски новых технических решений. Именно эти критерии — удобство, эффективность и безопасность — играли роль факторов естественного отбора при эволюции вычислительных систем.

Первый период (1945–1955 гг.). Ламповые машины. Операционных систем нет

Мы начнем исследование развития компьютерных комплексов с появления электронных вычислительных систем (опуская историю механических и электромеханических устройств).

Первые шаги в области разработки электронных вычислительных машин были предприняты в конце Второй мировой войны. В середине 40-х гг. были созданы первые ламповые вычислительные устройства и появился принцип программы, хранящейся в памяти машины (John Von Neumann, июнь 1945 г.). В то время одна и та же группа людей участвовала и в проектировании, и в эксплуатации,

и в программировании вычислительной машины. Это была скорее научно-исследовательская работа в области вычислительной техники, а не регулярное использование компьютеров в качестве инструмента решения каких-либо практических задач из других прикладных областей. Программирование осуществлялось исключительно на машинном языке. Об операционных системах не было и речи, все задачи организации вычислительного процесса решались вручную каждым программистом с пульта управления. За пультом мог находиться только один пользователь. Программа загружалась в память машины в лучшем случае с колоды перфокарт, а обычно с помощью панели переключателей.

Вычислительная система выполняла одновременно только одну операцию (ввод-вывод или собственно вычисления). Отладка программ велась с пульта управления с помощью изучения состояния памяти и регистров машины. В конце этого периода появляется первое системное программное обеспечение: в 1951–1952 гг. возникают прообразы первых компиляторов с символических языков (Fortran и др.), а в 1954 г. Нат Рочестер (Nat Rochester) разрабатывает Ассемблер для IBM-701 (рис. 1.2).

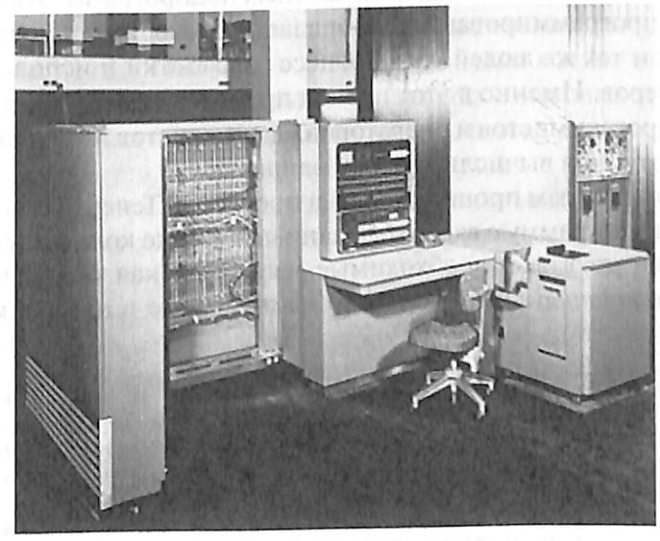

Рис. 1.2. IBM-701

Существенная часть времени уходила на подготовку запуска программы, а сами программы выполнялись строго последовательно. Такой режим работы называется *последовательной обработкой данных*. В целом первый период характеризуется крайне высокой стои-

мостью вычислительных систем, их малым количеством и низкой эффективностью использования.

Второй период (1955–1965 гг.). Компьютеры на основе транзисторов. Пакетные операционные системы

С середины 50-х гг. начался следующий период в эволюции вычислительной техники, связанный с появлением новой технической базы — полупроводниковых элементов. Применение транзисторов вместо часто перегоравших электронных ламп привело к повышению надежности компьютеров. Теперь машины могут непрерывно работать достаточно долго, чтобы на них можно было возложить выполнение практически важных задач. Снижается потребление вычислительными машинами электроэнергии, совершенствуются системы охлаждения. Размеры компьютеров уменьшились. Снизилась стоимость эксплуатации и обслуживания вычислительной техники. Началось использование ЭВМ коммерческими фирмами. Одновременно наблюдается бурное развитие алгоритмических языков (LISP, COBOL, ALGOL-60, PL-1 и т.д.).

Появляются первые настоящие компиляторы, редакторы связей, библиотеки математических и служебных подпрограмм. Упрощается процесс программирования. Пропадает необходимость взваливать на одних и тех же людей весь процесс разработки и использования компьютеров. Именно в этот период происходит разделение персонала на программистов и операторов, специалистов по эксплуатации и разработчиков вычислительных машин.

Изменяется сам процесс прогона программ. Теперь пользователь приносит программу с входными данными в виде колоды перфокарт (рис. 1.3) и указывает необходимые ресурсы. Такая колода получает название *задания*. Оператор загружает задание в память машины

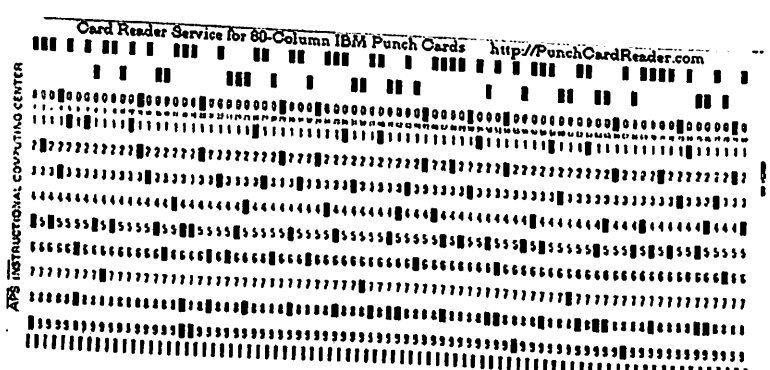

Рис. 1.3. Перфокарта

и запускает его на исполнение. Полученные выходные данные печатаются на принтере, и пользователь получает их обратно через некоторое (довольно продолжительное) время.

Смена запрошенных ресурсов вызывает приостановку выполнения программ, в результате процессор часто простаивает. Для повышения эффективности использования компьютера задания с похожими ресурсами начинают собирать вместе, создавая пакет заданий.

Появляются первые системы пакетной обработки, которые просто автоматизируют запуск одной программы из пакета за другой и тем самым увеличивают коэффициент загрузки процессора. При реализации систем пакетной обработки был разработан формализованный язык управления заданиями, с помощью которого программист сообщал системе и оператору, какую работу он хочет выполнить на вычислительной машине. Системы пакетной обработки стали прообразом современных операционных систем, они были первыми системными программами, предназначенными для управления вычислительным процессом.

Третий период (1965–1980 гг.). Компьютеры на основе интегральных микросхем. Первые многозадачные ОС

Следующий важный период развития вычислительных машин относится к началу 1960-х гг. — 1980 г. В это время в технической базе произошел переход от отдельных полупроводниковых элементов типа транзисторов к интегральным микросхемам. Вычислительная техника становится более надежной и дешевой.

Растет сложность и количество задач, решаемых компьютерами. Повышается производительность процессоров.

Повышению эффективности использования процессорного времени мешает низкая скорость работы механических устройств ввода-вывода (быстрый считыватель перфокарт мог обработать 1200 перфокарт в минуту, принтеры печатали до 600 строк в минуту). Вместо непосредственного чтения пакета заданий с перфокарт в память начинают использовать его предварительную запись, сначала на магнитную ленту, а затем и на диск. Когда в процессе выполнения задания требуется ввод данных, они читаются с диска.

Точно так же выходная информация сначала копируется в системный буфер и записывается на ленту или диск, а печатается только после завершения задания. Вначале действительные операции ввода-вывода осуществлялись в режиме off-line, т.е. с использованием других, более простых, отдельно стоящих компьютеров. В дальнейшем они начинают выполняться на том же компьютере, который произ-

водит вычисления, т.е. в режиме on-line. Такой прием получает название «spooling» (сокращение от Simultaneous Peripheral Operation On Line) или подкачки-откачки данных. Введение техники подкачки-откачки в пакетные системы позволило совместить реальные операции ввода-вывода одного задания с выполнением другого задания, но потребовало разработки аппарата прерываний для извещения процессора об окончании этих операций.

Магнитные ленты были устройствами последовательного доступа, т.е. информация считывалась с них в том порядке, в каком была записана. Появление магнитного диска, для которого не важен порядок чтения информации, т.е. устройства прямого доступа, привело к дальнейшему развитию вычислительных систем. При обработке пакета заданий на магнитной ленте очередность запуска заданий определялась порядком их ввода. При обработке пакета заданий на магнитном диске появилась возможность выбора очередного выполняемого задания. Пакетные системы начинают заниматься планированием заданий: в зависимости от наличия запрошенных ресурсов, срочности вычислений и т.д. выбирается то или иное задание.

Дальнейшее повышение эффективности использования процессора было достигнуто с помощью мультипрограммирования. Идея мультипрограммирования заключается в следующем: пока одна программа выполняет операцию ввода-вывода, процессор не простаивает, как это происходило при однопрограммном режиме, а выполняет другую программу. Когда операция ввода-вывода заканчивается, процессор возвращается к выполнению первой программы. Эта идея напоминает поведение преподавателя и студентов на экзамене. Пока один студент (программа) обдумывает ответ на вопрос (операция ввода-вывода), преподаватель (процессор) выслушивает ответ другого студента (вычисления). Естественно, такая ситуация требует наличия в комнате нескольких студентов. Точно так же мультипрограммирование требует наличия в памяти нескольких программ одновременно. При этом каждая программа загружается в свой участок оперативной памяти, называемый разделом, и не должна влиять на выполнение другой программы. (Студенты сидят за отдельными столами и не подсказывают друг другу.)

Появление мультипрограммирования требует настоящей революции в строении вычислительной системы. Особую роль здесь играет аппаратная поддержка (многие аппаратные новшества появились еще на предыдущем этапе эволюции), наиболее существенные особенности которой перечислены ниже.

- Реализация защитных механизмов. Программы не должны иметь самостоятельного доступа к распределению ресурсов, что приводит к появлению привилегированных и непривилегированных команд. Привилегированные команды, например команды ввода-вывода, могут исполняться только операционной системой. Говорят, что она работает в привилегированном режиме. Переход управления от прикладной программы к операционной системе сопровождается контролируемой сменой режима. Кроме того, это защита памяти, позволяющая изолировать конкурирующие пользовательские программы друг от друга, а операионную систему — от программ пользователей.
- Наличие прерываний. Внешние прерывания оповещают операционную систему о том, что произошло асинхронное событие, например завершилась операция ввода-вывода. Внутренние прерывания (сейчас их принято называть исключительными ситуациями) возникают, когда выполнение программы привело к ситуации, требующей вмешательства операционной системы, например деление на ноль или попытка нарушения защиты.

Развитие параллелизма в архитектуре. Прямой доступ к памяти и организация каналов ввода-вывода позволили освободить центральный процессор от рутинных операций.

Не менее важна в организации мультипрограммирования роль операционной системы. Она отвечает за следующие операции.

1. Организация интерфейса между прикладной программой и ОС при помощи системных вызовов.

2. Организация очереди из заданий в памяти и выделение процессора одному из заданий потребовало планирования использования процессора.

3. Переключение с одного задания на другое требует сохранения содержимого регистров и структур данных, необходимых для выполнения задания, иначе говоря, контекста для обеспечения правильного продолжения вычислений.

4. Поскольку память является ограниченным ресурсом, нужны стратегии управления памятью, т.е. требуется упорядочить процессы размещения, замещения и выборки информации из памяти.

5. Организация хранения информации на внешних носителях в виде файлов и обеспечение доступа к конкретному файлу только определенным категориям пользователей.

6. Поскольку программам может потребоваться произвести санкционированный обмен данными, необходимо их обеспечить средствами коммуникации.

7. Для корректного обмена данными необходимо разрешать конфликтные ситуации, возникающие при работе с различными ресурсами, и предусмотреть координацию программами своих действий, т.е. снабдить систему средствами синхронизации.

Мультипрограммные системы обеспечили возможность более эффективного использования системных ресурсов (например, процессора, памяти, периферийных устройств), но они еще долго оставались пакетными. Пользователь не мог непосредственно взаимодействовать с заданием и должен был предусмотреть с помощью управляющих карт все возможные ситуации. Отладка программ по-прежнему занимала много времени и требовала изучения многостраничных распечаток содержимого памяти и регистров или использования отладочной печати.

Появление электронно-лучевых дисплеев и переосмысление возможностей применения клавиатур поставили на очередь решение этой проблемы. Логическим расширением систем мультипрограммирования стали time-sharing-системы, или системы разделения времени. В них процессор переключается между задачами не только на время операций ввода-вывода, но и просто по прошествии определенного времени. Эти переключения происходят так часто, что пользователи могут взаимодействовать со своими программами во время их выполнения, т.е. интерактивно. В результате появляется возможность одновременной работы нескольких пользователей на одной компьютерной системе. У каждого пользователя для этого должна быть хотя бы одна программа в памяти. Чтобы уменьшить ограничения на количество работающих пользователей, была внедрена идея неполного нахождения исполняемой программы в оперативной памяти. Основная часть программы находится на диске, и фрагмент, который необходимо в данный момент выполнять, может быть загружен в оперативную память, а ненужный — выкачан обратно на диск. Это реализуется с помощью механизма виртуальной памяти. Основным достоинством такого механизма является создание иллюзии неограниченной оперативной памяти ЭВМ.

В системах разделения времени пользователь получил возможность эффективно производить отладку программы в интерактивном режиме и записывать информацию на диск, не используя перфокарты, а непосредственно с клавиатуры. Появление on-line-файлов привело к необходимости разработки развитых файловых систем.

Параллельно внутренней эволюции вычислительных систем происходила и внешняя их эволюция. До начала этого периода вычислительные комплексы были, как правило, несовместимы. Каждый

имел собственную операционную систему, свою систему команд и т.д. В результате программу, успешно работающую на одном типе машин, необходимо было полностью переписывать и заново отлаживать для выполнения на компьютерах другого типа. В начале третьего периода появилась идея создания семейств программно совместимых машин, работающих под управлением одной и той же операционной системы.

Первым семейством программно совместимых компьютеров, построенных на интегральных микросхемах, стала серия машин IBM/360. Разработанное в начале 1960-х гг., это семейство значительно превосходило машины второго поколения по критерию цена/производительность. За ним последовала линия компьютеров PDP, несовместимых с линией IBM, и лучшей моделью в ней стала PDP-11 (рис. 1.4).

Рис. 1.4. ЭВМ PDP 11/40

Сила «одной семьи» была одновременно и ее слабостью. Широкие возможности этой концепции (наличие всех моделей: от миникомпьютеров до гигантских машин; обилие разнообразной периферии; различное окружение; различные пользователи) порождали сложную и громоздкую операционную систему.

Миллионы строчек Ассемблера, написанные тысячами программистов, содержали множество ошибок, что вызывало непрерывный поток публикаций о них и попыток исправления. Только в операционной системе OS/360 содержалось более 1000 известных ошибок. Тем не менее, идея стандартизации операционных систем была широко внедрена в сознание пользователей и в дальнейшем получила активное развитие.

Четвертый период (с 1980 г. по настоящее время). Персональные компьютеры. Классические, сетевые и распределенные системы

Следующий период в эволюции вычислительных систем связан с появлением больших интегральных схем (БИС). В эти годы произошли резкое возрастание степени интеграции и снижение стоимости микросхем. Компьютер, не отличающийся по архитектуре от PDP-11, по цене и простоте эксплуатации стал доступен отдельному человеку, а не отделу предприятия или университета. Наступила эра персональных компьютеров (рис. 1.5). Первоначально персональные компьютеры предназначались для использования одним пользователем в однопрограммном режиме, что повлекло за собой деградацию архитектуры этих ЭВМ и их операционных систем (в частности, пропала необходимость защиты файлов и памяти, планирования заданий и т.п.).

Рис. 1.5. Современный персональный компьютер

Компьютеры стали использоваться не только специалистами, что потребовало разработки «дружественного» программного обеспечения.

Однако рост сложности и разнообразия задач, решаемых на персональных компьютерах, необходимость повышения надежности их работы привели к возрождению практически всех черт, характерных для архитектуры больших вычислительных систем.

В середине 1980-х гг. стали бурно развиваться сети компьютеров, в том числе персональных, работающих под управлением сетевых или распределенных операционных систем.

В сетевых операционных системах пользователи могут получить доступ к ресурсам другого сетевого компьютера, только они должны знать об их наличии и уметь это сделать. Каждая машина в сети работает под управлением своей локальной операционной системы, отличающейся от операционной системы автономного компьютера наличием дополнительных средств (программной поддержкой для сетевых интерфейсных устройств и доступа к удаленным ресурсам), но эти дополнения не меняют структуру операционной системы.

Распределенная система, напротив, внешне выглядит как обычная автономная система. Пользователь не знает и не должен знать, где его файлы хранятся — на локальной или удаленной машине — и где его программы выполняются. Он может вообще не знать, подключен ли его компьютер к сети. Внутреннее строение распределенной операционной системы имеет существенные отличия от автономных систем.

В дальнейшем автономные операционные системы мы будем называть классическими операционными системами.

Пятый период (с 1990 г. по настоящее время). Мобильные компьютеры

Первый настоящий мобильный телефон появился в 1946 г., 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 гг. 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 гг., первый настоящий смартфон появился только в середине 1990-х гг., когда «Nokia» выпустила свой N9000, представлявший собой комбинацию из двух отдельных устройств: телефона и КПК. В 1997 г. 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. В настоящее время доминирующей является операционная система Google Android, а на втором месте находится 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 г.) и Apple iOS (выпущенная для первого iPhone в 2007 г.). Многие ожидали, что RIM будет доминировать на рынке бизнес-устройств, а iOS завоюет рынок потребительских устройств. Для рынка популярность Symbian упала. В 2011 г. «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 г.

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.

Просмотрев этапы развития вычислительных систем, мы можем выделить шесть основных функций, которые выполняли классические операционные системы в процессе эволюции:

- планирование заданий и использования процессора;
- обеспечение программ средствами коммуникации и синхронизации;
- управление памятью;
- управление файловой системой;
- управление вводом-выводом;
- обеспечение безопасности.

Каждая из приведенных функций обычно реализована в виде подсистемы, являющейся структурным компонентом ОС. В каждой операционной системе эти функции, конечно, реализовывались по-своему, в различном объеме. Они не были изначально придуманы как составные части операционных систем, а появились в процессе развития, по мере того как вычислительные системы становились все более удобными, эффективными и безопасными. Эволюция вычислительных систем, созданных человеком, пошла по такому пути, но никто еще не доказал, что это единственно возможный путь их развития. Операционные системы существуют потому, что на данный момент их существование — это разумный способ использования вычислительных систем.

1.3. Основные понятия, концепции операционных систем

В процессе эволюции возникло несколько важных концепций, которые стали неотъемлемой частью теории и практики операционных систем.

Системные вызовы

В любой операционной системе поддерживается механизм, который позволяет пользовательским программам обращаться к услугам ядра операционной системы. В операционных системах наиболее известной советской вычислительной машины БЭСМ-6 соответствующие средства «общения» с ядром назывались экстракодами, в операционных системах IBM они назывались системными макрокомандами и т.д. В ОС Unix такие средства называют системными вызовами.

Системные вызовы (system calls) — это интерфейс между операционной системой и пользовательской программой. Они создают, удаляют и используют различные объекты, главные из которых —

процессы и файлы. Пользовательская программа запрашивает сервис у операционной системы, осуществляя системный вызов. Имеются библиотеки процедур, которые загружают машинные регистры определенными параметрами и осуществляют прерывание процессора, после чего управление передается обработчику данного вызова, входящему в ядро операционной системы. Цель таких библиотек — сделать системный вызов похожим на обычный вызов подпрограммы.

Основное отличие состоит в том, что при системном вызове задача переходит в привилегированный режим или режим ядра (kernel mode). Поэтому системные вызовы иногда еще называют программными прерываниями, в отличие от аппаратных прерываний, которые чаще называют просто прерываниями.

В этом режиме работает код ядра операционной системы, причем исполняется он в адресном пространстве и в контексте вызвавшей его задачи. Таким образом, ядро операционной системы имеет полный доступ к памяти пользовательской программы, и при системном вызове достаточно передать адреса одной или нескольких областей памяти с параметрами вызова и адреса одной или нескольких областей памяти для результатов вызова.

В большинстве операционных систем системный вызов осуществляется командой программного прерывания (INT). *Программное прерывание* — это синхронное событие, которое может быть повторено при выполнении одного и того же программного кода.

Прерывания

Прерывание (hardware interrupt) — это событие, генерируемое внешним (по отношению к процессору) устройством. Посредством аппаратных прерываний аппаратура либо информирует центральный процессор о том, что произошло какое-либо событие, требующее немедленной реакции (например, пользователь нажал клавишу), либо сообщает о завершении асинхронной операции ввода-вывода (например, закончено чтение данных с диска в основную память). Важный тип аппаратных прерываний — прерывания таймера, которые генерируются периодически через фиксированный промежуток времени. Прерывания таймера используются операционной системой при планировании процессов. Каждый тип аппаратных прерываний имеет собственный номер, однозначно определяющий источник прерывания. *Аппаратное прерывание* — это асинхронное событие, т.е. оно возникает вне зависимости от того, какой код

исполняется процессором в данный момент. Обработка аппаратного прерывания не должна учитывать, какой процесс является текущим.

Исключительные ситуации

Исключительная ситуация (exception) — событие, возникающее в результате попытки выполнения программой команды, которая по каким-то причинам не может быть выполнена до конца. Примерами таких команд могут быть попытки доступа к ресурсу при отсутствии достаточных привилегий или обращения к отсутствующей странице памяти. Исключительные ситуации, как и системные вызовы, являются синхронными событиями, возникающими в контексте текущей задачи. Исключительные ситуации можно разделить на исправимые и неисправимые. К исправимым относятся такие исключительные ситуации, как отсутствие нужной информации в оперативной памяти. После устранения причины исправимой исключительной ситуации программа может выполняться дальше. Возникновение в процессе работы операционной системы исправимых исключительных ситуаций считается нормальным явлением. Неисправимые исключительные ситуации чаще всего возникают в результате ошибок в программах (например, деление на ноль). Обычно в таких случаях операционная система реагирует завершением программы, вызвавшей исключительную ситуацию.

Файлы

Файлы предназначены для хранения информации на внешних носителях, т.е. принято, что информация, записанная, например, на диске, должна находиться внутри файла. Обычно под *файлом* понимают именованную часть пространства на носителе информации.

Главная задача файловой системы (file system) — скрыть особенности ввода-вывода и дать программисту простую абстрактную модель файлов, независимых от устройств. Для чтения, создания, удаления, записи, открытия и закрытия файлов также имеется обширная категория системных вызовов (создание, удаление, открытие, закрытие, чтение и т.д.). Пользователям хорошо знакомы такие связанные с организацией файловой системы понятия, как каталог, текущий каталог, корневой каталог, путь. Для манипулирования этими объектами в операционной системе имеются системные вызовы.

Структура операционной системы

До сих пор мы говорили о взгляде на операционные системы извне, о том, что делают операционные системы. Но мы пока ничего

не сказали о том, что они представляют собой изнутри, какие подходы существуют к их построению.

Монолитные системы

По сути дела, операционная система — это обычная программа, поэтому было бы логично и организовать ее так же, как устроено большинство программ, т.е. составить из процедур и функций. В этом случае компоненты операционной системы являются не самостоятельными модулями, а составными частями одной большой программы. Такая структура операционной системы называется *монолитным ядром* (monolithic kernel). Монолитное ядро представляет собой набор процедур, каждая из которых может вызвать каждую. Все процедуры работают в привилегированном режиме. Таким образом, монолитное ядро — это такая схема операционной системы, при которой все ее компоненты являются составными частями одной программы, используют общие структуры данных и взаимодействуют друг с другом путем непосредственного вызова процедур. Для монолитной операционной системы ядро совпадает со всей системой.

Во многих операционных системах с монолитным ядром сборка ядра, т.е. его компиляция, осуществляется отдельно для каждого компьютера, на который устанавливается операционная система. При этом можно выбрать список оборудования и программных протоколов, поддержка которых будет включена в ядро. Так как ядро является единой программой, перекомпиляция — это единственный способ добавить в него новые компоненты или исключить неиспользуемые. Следует отметить, что присутствие в ядре лишних компонентов крайне нежелательно, так как ядро всегда полностью располагается в оперативной памяти. Кроме того, исключение ненужных компонентов повышает надежность операционной системы в целом.

Монолитное ядро — старейший способ организации операционных систем. Примером систем с монолитным ядром является большинство Unix-систем.

Даже в монолитных системах можно выделить некоторую структуру. Как в бетонной глыбе можно различить вкрапления щебенки, так и в монолитном ядре выделяются вкрапления сервисных процедур, соответствующих системным вызовам. Сервисные процедуры выполняются в привилегированном режиме, тогда как пользовательские программы — в непривилегированном. Для перехода с одного уровня привилегий на другой иногда может использоваться главная сервисная программа, определяющая, какой именно системный вызов был сделан, корректность входных данных для этого вызова и передающая управление соответствующей сервисной процедуре с переходом в привилегированный режим работы. Иногда выделяют также набор программных утилит, которые помогают выполнять сервисные процедуры.

Многоуровневые системы (Layered systems)

Продолжая структуризацию, можно разбить всю вычислительную систему на ряд более мелких уровней с хорошо определенными связями между ними, так чтобы объекты уровня N могли вызывать только объекты уровня $N-1$. Нижним уровнем в таких системах обычно является hardware, верхним уровнем — интерфейс пользователя. Чем ниже уровень, тем более привилегированные команды и действия может выполнять модуль, находящийся на этом уровне. Впервые такой подход был применен при создании системы THE (Technishe Hogeschool Eindhoven) Дейкстрой (Dijkstra) и его студентами в 1968 г. Эта система имела следующие уровни (рис. 1.6):

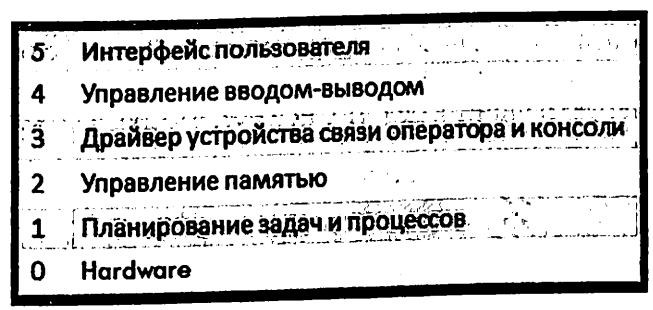

Рис. 1.6. Слоеная система THE

Слоеные системы хорошо реализуются. При использовании операций нижнего слоя не нужно знать, как они реализованы, нужно лишь понимать, что они делают. Слоеные системы хорошо тестируются. Отладка начинается с нижнего слоя и проводится послойно. При возникновении ошибки мы можем быть уверены, что она находится в тестируемом слое. Слоеные системы хорошо модифицируются. При необходимости можно заменить лишь один слой, не трогая остальные. Но слоеные системы сложны для разработки: тяжело правильно определить порядок слоев и что к какому слою относится. Слоеные системы менее эффективны, чем монолитные. Так, например, для выполнения операций ввода-вывода программе пользователя придется последовательно проходить все слои от верхнего до нижнего.

Виртуальные машины

Вначале мы говорили о взгляде на операционную систему как на виртуальную машину, когда пользователю нет необходимости знать детали внутреннего устройства компьютера. Он работает с файлами, а не с магнитными головками и двигателем; он работает с огромной виртуальной, а не ограниченной реальной оперативной памятью; его мало волнует, единственный он на машине пользователь или нет. Рассмотрим несколько иной подход. Пусть операционная система реализует виртуальную машину для каждого пользователя, но не упрощая ему жизнь, а, наоборот, усложняя. Каждая такая виртуальная машина предстает перед пользователем как голое железо — копия всего hardware в вычислительной системе, включая процессор, привилегированные и непривилегированные команды, устройства ввода-вывода, прерывания и т.д. (рис. 1.7). И он остается с этим железом один на один. При попытке обратиться к такому виртуальному железу на уровне привилегированных команд в действительности происходит системный вызов реальной операционной системы, которая и производит все необходимые действия. Такой подход позволяет каждому пользователю загрузить свою операционную систему на виртуальную машину и делать с ней все, что душа пожелает.

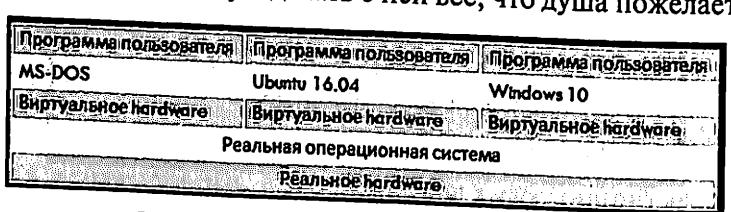

Рис. 1.7. Вариант виртуальной машины

Первой реальной системой такого рода была система CP/CMS, или VM/370, как ее называют сейчас, для семейства машин IBM/370.

Недостатком таких операционных систем является снижение эффективности виртуальных машин по сравнению с реальным компьютером, и, как правило, они очень громоздки. Преимущество же заключается в использовании на одной вычислительной системе программ, написанных для разных операционных систем.

Микроядра

Современная тенденция в разработке операционных систем состоит в перенесении значительной части системного кода на уровень пользователя и одновременной минимизации ядра. Речь идет о подходе к построению ядра, называемом *микроядерной архитектурой* (microkernel architecture) операционной системы, когда большинство ее составляющих являются самостоятельными программами (рис. 1.8). В этом случае взаимодействие между ними обеспечивает специальный модуль ядра, называемый микроядром. Микроядро работает в привилегированном режиме и обеспечивает взаимодействие между программами, планирование использования процессора, первичную обработку прерываний, операции ввода-вывода и базовое управление памятью. Остальные компоненты системы взаимодействуют друг с другом путем передачи сообщений через микроядро.

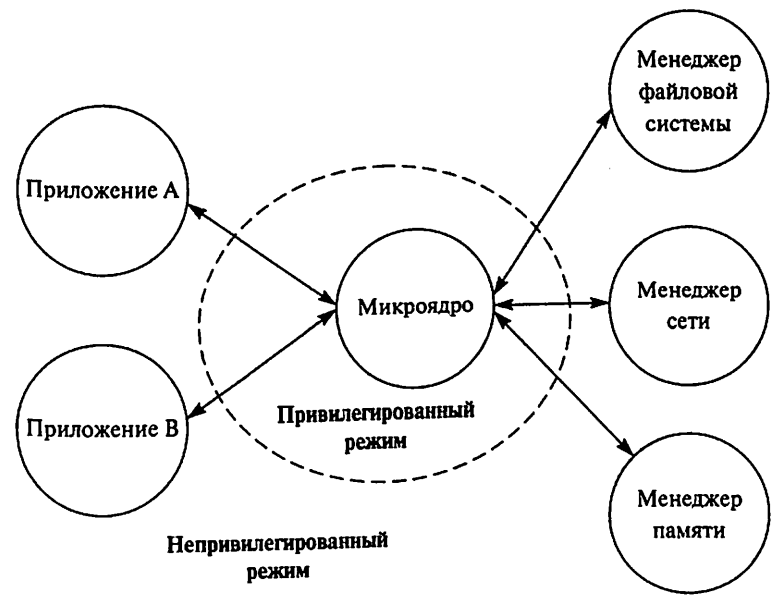

Рис. 1.8. Микроядерная архитектура операционной системы

Основное достоинство микроядерной архитектуры — высокая степень модульности ядра операционной системы. Это существенно упрощает добавление в него новых компонентов. В микроядерной операционной системе можно, не прерывая ее работы, загружать и выгружать новые драйверы, файловые системы и т.д. Существенно упрощается процесс отладки компонентов ядра, так как новая версия драйвера может загружаться без перезапуска всей операционной системы. Компоненты ядра операционной системы ничем принципиально не отличаются от пользовательских программ, поэтому для их отладки можно применять обычные средства. Микроядерная архитектура повышает надежность системы, поскольку ошибка на уровне непривилегированной программы менее опасна, чем отказ на уровне режима ядра.

В то же время микроядерная архитектура операционной системы вносит дополнительные накладные расходы, связанные с передачей сообщений, что существенно влияет на производительность. Для того чтобы микроядерная операционная система по скорости не уступала операционным системам на базе монолитного ядра, требуется очень аккуратно проектировать разбиение системы на компоненты, стараясь минимизировать взаимодействие между ними. Таким образом, основная сложность при создании микроядерных операционных систем — необходимость очень аккуратного проектирования.

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. 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.

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*. 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.

1.4. Классификация 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. Вкратце коснемся девяти операционных систем.

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 дан-

ных, — весьма обычное явление. 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.

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.

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.

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 10 и OS X компании «Apple».

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.

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.

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.

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 военных 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.

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).

Глава 2
ПРОЦЕССЫ, ИХ СОСТОЯНИЯ И ОПЕРАЦИИ НАД НИМИ

Фундаментальным понятием для изучения работы операционных систем является понятие процессов как основных динамических объектов, над которыми системы выполняют определенные действия. Данная глава посвящена описанию таких объектов, их состояний и свойств, их представлению в вычислительных системах, а также операциям, которые могут проводиться над ними.

2.1. Понятие процесса

В первой главе, поясняя понятие «операционная система» и описывая способы построения операционных систем, мы часто применяли слова «программа» и «задание». Мы говорили: вычислительная система исполняет одну или несколько программ, операционная система планирует задания, программы могут обмениваться данными и т.д. Мы использовали эти термины в некотором общеупотребительном, житейском смысле, предполагая, что все читатели одинаково представляют себе, что подразумевается под ними в каждом конкретном случае. При этом одни и те же слова обозначали и объекты в статическом состоянии, не обрабатывающиеся вычислительной системой (например, совокупность файлов на диске), и объекты в динамическом состоянии, находящиеся в процессе исполнения. Это было возможно, пока мы говорили об общих свойствах операционных систем, не вдаваясь в подробности их внутреннего устройства и поведения, или о работе вычислительных систем первого–второго поколений, которые не могли обрабатывать более одной программы или одного задания одновременно, по сути дела не имея операционных систем. Но теперь мы начинаем знакомиться с деталями функционирования современных компьютерных систем, и нам придется уточнить терминологию.

Рассмотрим следующий пример. Два студента запускают программу извлечения квадратного корня. Один хочет вычислить квадратный корень из 4, а второй — из 1. С точки зрения студентов, запущена одна и та же программа; с точки зрения компьютерной системы, ей приходится заниматься двумя различными вычислительными процессами, так как разные исходные данные приводят

к разному набору вычислений. Следовательно, на уровне происходящего внутри вычислительной системы мы не можем использовать термин «программа» в пользовательском смысле слова.

Рассматривая системы пакетной обработки, мы ввели понятие «задание» как совокупность программы, набора команд языка управления заданиями, необходимых для ее выполнения, и входных данных. С точки зрения студентов, они, подставив разные исходные данные, сформировали два различных задания. Может быть, термин «задание» подойдет нам для описания внутреннего функционирования компьютерных систем? Чтобы выяснить это, давайте рассмотрим другой пример. Пусть оба студента пытаются извлечь корень квадратный из 1, т.е. пусть они сформировали идентичные задания, но загрузили их в вычислительную систему со сдвигом по времени. В то время как одно из выполняемых заданий приступило к печати полученного значения и ждет окончания операции ввода-вывода, второе только начинает исполняться. Можно ли говорить об идентичности заданий внутри вычислительной системы в данный момент? Нет, так как состояние процесса их выполнения различно. Следовательно, и слово «задание» в пользовательском смысле не может применяться для описания происходящего в вычислительной системе.

Это происходит потому, что термины «программа» и «задание» предназначены для описания статических, неактивных объектов. Программа же в процессе исполнения является динамическим, активным объектом. По ходу ее работы компьютер обрабатывает различные команды и преобразует значения переменных. Для выполнения программы операционная система должна выделить определенное количество оперативной памяти, закрепить за ней определенные устройства ввода-вывода или файлы (откуда должны поступать входные данные и куда нужно доставить полученные результаты), т.е. зарезервировать определенные ресурсы из общего числа ресурсов всей вычислительной системы. Их количество и конфигурация с течением времени могут изменяться. Для описания таких активных объектов внутри компьютерной системы вместо терминов «программа» и «задание» мы будем использовать новый термин — «процесс».

В ряде учебных пособий и монографий для простоты предлагается рассматривать *процесс* как абстракцию, характеризующую программу во время выполнения. На наш взгляд, эта рекомендация не совсем корректна. Понятие *процесса* характеризует некоторую совокупность набора исполняющихся команд, ассоциированных с ним ресурсов

(выделенная для исполнения память или адресное пространство, стеки, используемые файлы и устройства ввода-вывода и т.д.) и текущего момента его выполнения (значения регистров, программного счетчика, состояние стека и значения переменных), находящуюся под управлением операционной системы. Не существует взаимно-однозначного соответствия между процессами и программами, обрабатываемыми вычислительными системами. Как будет показано далее, в некоторых операционных системах для работы определенных программ может организовываться более одного процесса или один и тот же процесс может исполнять последовательно несколько различных программ. Более того, даже в случае обработки только одной программы в рамках одного процесса нельзя считать, что процесс представляет собой просто динамическое описание кода исполняемого файла, данных и выделенных для них ресурсов. Процесс находится под управлением операционной системы, поэтому в нем может выполняться часть кода ее ядра (не находящегося в исполняемом файле!), как в случаях, специально запланированных авторами программы (например, при использовании системных вызовов), так и в непредусмотренных ситуациях (например, при обработке внешних прерываний).

2.2. Состояния процесса

При использовании такой абстракции все, что выполняется в вычислительных системах (не только программы пользователей, но и, возможно, определенные части операционных систем), организовано как набор процессов. Понятно, что реально на однопроцессорной компьютерной системе в каждый момент времени может исполняться только один процесс. Для мультипрограммных вычислительных систем псевдопараллельная обработка нескольких процессов достигается с помощью переключения процессора с одного процесса на другой. Пока один процесс выполняется, остальные ждут своей очереди.

Как видим, каждый процесс может находиться как минимум в двух состояниях: процесс исполняется и процесс не исполняется. Диаграмма состояний процесса в такой модели изображена на рис. 2.1.

Процесс, находящийся в состоянии «процесс исполняется», через некоторое время может быть завершен операционной системой или приостановлен и снова переведен в состояние «процесс не исполняется». Приостановка процесса происходит по двум причинам: для

его дальнейшей работы потребовалось какое-либо событие (например, завершение операции ввода-вывода) или истек временной интервал, отведенный операционной системой для работы данного процесса. После этого операционная система по определенному алгоритму выбирает для исполнения один из процессов, находящихся в состоянии «процесс не исполняется», и переводит его в состояние «процесс исполняется». Новый процесс, появляющийся в системе, первоначально помещается в состояние «процесс не исполняется».

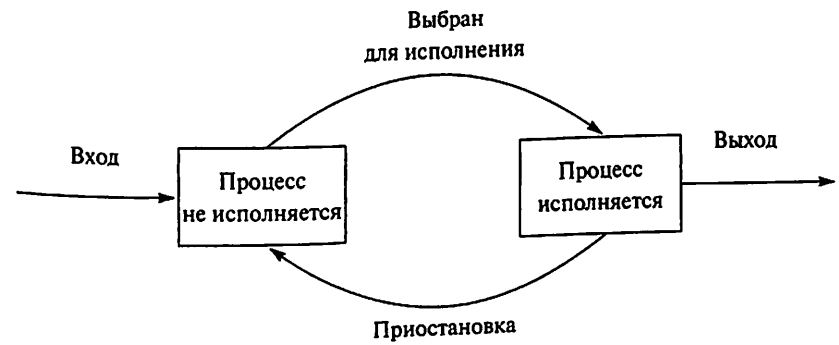

Рис. 2.1. Простейшая диаграмма состояний процесса

Это очень грубая модель, она не учитывает, в частности, то, что процесс, выбранный для исполнения, может все еще ждать события, из-за которого он был приостановлен, и реально к выполнению не готов. Для того чтобы избежать такой ситуации, разобьем состояние «процесс не исполняется» на два новых состояния: «готовность» и «ожидание» (рис. 2.2).

Всякий новый процесс, появляющийся в системе, попадает в состояние готовность. Операционная система, пользуясь каким-либо алгоритмом планирования, выбирает один из готовых процессов и переводит его в состояние «исполнение». В состоянии «исполнение» происходит непосредственное выполнение программного кода процесса. Выйти из этого состояния процесс может по трем причинам:

- операционная система прекращает его деятельность;
- он не может продолжать свою работу, пока не произойдет некоторое событие, и операционная система переводит его в состояние «ожидание»;
- в результате возникновения прерывания в вычислительной системе (например, прерывания от таймера по истечении предусмотренного времени выполнения) его возвращают в состояние «готовность».

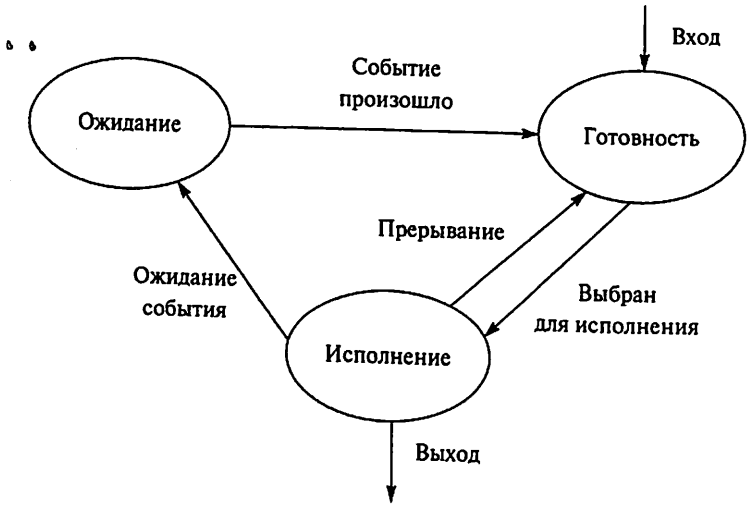

Рис. 2.2. Более подробная диаграмма состояний процесса

Из состояния «ожидание» процесс попадает в состояние «готовность» после того, как ожидаемое событие произошло и он снова может быть выбран для исполнения.

Наша новая модель хорошо описывает поведение процессов во время их существования, но она не акцентирует внимания на появлении процесса в системе и его исчезновении. Для полноты картины нам необходимо ввести еще два состояния процессов: «рождение» и «закончил исполнение» (рис. 2.3).

Теперь для появления в вычислительной системе процесс должен пройти через состояние «рождение». При рождении процесс получает в свое распоряжение адресное пространство, в которое загружается программный код процесса; ему выделяются стек и системные ресурсы; устанавливается начальное значение программного счетчика этого процесса и т.д. Родившийся процесс переводится в состояние «готовность». При завершении своей деятельности процесс из состояния «исполнение» попадает в состояние «закончил исполнение».

В конкретных операционных системах состояния процесса могут быть еще более детализированы, могут появиться некоторые новые варианты переходов из одного состояния в другое. Так, например, модель состояний процессов для операционной системы Windows NT содержит семь различных состояний, а для операционной системы Unix — девять. Тем не менее, все операционные системы подчиняются изложенной выше модели.

Рис. 2.3. Полная диаграмма состояний процесса

2.3. Операции над процессами и связанные с ними понятия

Набор операций

Процесс не может перейти из одного состояния в другое самостоятельно. Изменением состояния процессов занимается операционная система, совершая операции над ними. Количество таких операций в нашей модели пока совпадает с количеством стрелок на диаграмме состояний. Удобно объединить их в три пары:

- создание процесса — завершение процесса;
- приостановка процесса (перевод из состояния «исполнение» в состояние «готовность») — запуск процесса (перевод из состояния «готовность» в состояние «исполнение»);
- блокирование процесса (перевод из состояния «исполнение» в состояние «ожидание») — разблокирование процесса (перевод из состояния «ожидание» в состояние «готовность»).

В дальнейшем, когда мы будем говорить об алгоритмах планирования, в нашей модели появится еще одна операция, не имеющая парной: изменение приоритета процесса.

Операции создания и завершения процесса являются одноразовыми, так как применяются к процессу не более одного раза (некоторые системные процессы при работе вычислительной системы не завершаются никогда). Все остальные операции, связанные с изменением состояния процессов, будь то запуск или блокировка, как правило, являются многоразовыми. Рассмотрим подробнее, как операционная система выполняет операции над процессами.

Process Control Block и контекст процесса

Для того чтобы операционная система могла выполнять операции над процессами, каждый процесс представляется в ней некоторой структурой данных. Эта структура содержит информацию, специфическую для данного процесса:

- состояние, в котором находится процесс;
- программный счетчик процесса или, другими словами, адрес команды, которая должна быть выполнена для него следующей;
- содержимое регистров процессора;
- данные, необходимые для планирования использования процессора и управления памятью (приоритет процесса, размер и расположение адресного пространства и т.д.);
- учетные данные (идентификационный номер процесса, какой пользователь инициировал его работу, общее время использования процессора данным процессом и т.д.);
- сведения об устройствах ввода-вывода, связанных с процессом (например, какие устройства закреплены за процессом, таблицу открытых файлов).

Ее состав и строение зависят, конечно, от конкретной операционной системы. Во многих операционных системах информация, характеризующая процесс, хранится не в одной, а в нескольких связанных структурах данных. Эти структуры могут иметь различные наименования, содержать дополнительную информацию или, наоборот, лишь часть описанной информации. Для нас это не имеет значения. Для нас важно лишь то, что для любого процесса, находящегося в вычислительной системе, вся информация, необходимая для совершения операций над ним, доступна операционной системе. Для простоты изложения будем считать, что она хранится в одной структуре данных. Мы будем называть ее *PCB* (*Process Control Block*) или блоком управления процессом. Блок управления процессом является моделью процесса для операционной системы. Любая операция, производимая операционной системой над процессом, вызывает определенные изменения в PCB. В рамках принятой модели состояний процессов содержимое PCB между операциями остается постоянным.

Информацию, для хранения которой предназначен блок управления процессом, удобно для дальнейшего изложения разделить на две части. Содержимое всех регистров процессора (включая значение программного счетчика) будем называть *регистровым контекстом процесса*, а все остальное — *системным контекстом процесса*. Знания регистрового и системного контекстов процесса достаточно для того, чтобы управлять его работой в операционной системе, совершая над ним операции. Однако этого недостаточно для того, чтобы полностью охарактеризовать процесс. Операционную систему не интересует, какими именно вычислениями занимается процесс, т.е. какой код и какие данные находятся в его адресном пространстве. С точки зрения пользователя, наоборот, наибольший интерес представляет содержимое адресного пространства процесса, возможно, наряду с регистровым контекстом определяющее последовательность преобразования данных и полученные результаты. Код и данные, находящиеся в адресном пространстве процесса, будем называть его *пользовательским контекстом*. Совокупность регистрового, системного и пользовательского контекстов процесса для краткости принято называть просто *контекстом процесса*. В любой момент времени процесс полностью характеризуется своим контекстом.

Одноразовые операции

Сложный жизненный путь процесса в компьютере начинается с его рождения. Любая операционная система, поддерживающая концепцию процессов, должна обладать средствами для их создания. В очень простых системах (например, в системах, спроектированных для работы только одного конкретного приложения) все процессы могут быть порождены на этапе старта системы. Более сложные операционные системы создают процессы динамически, по мере необходимости. Инициатором рождения нового процесса после старта операционной системы может выступить либо процесс пользователя, совершивший специальный системный вызов, либо сама операционная система, т.е. в конечном итоге тоже некоторый процесс. Процесс, инициировавший создание нового процесса, принято называть процессом-родителем (parent process), а вновь созданный процесс — процессом-ребенком (child process). Процессы-дети могут в свою

очередь порождать новых детей и т.д., образуя, в общем случае, внутри системы набор генеалогических деревьев процессов — генеалогический лес. Пример генеалогического леса изображен на рис. 2.4. Следует отметить, что все пользовательские процессы вместе с некоторыми процессами операционной системы принадлежат одному и тому же дереву леса. В ряде вычислительных систем лес вообще вырождается в одно такое дерево.

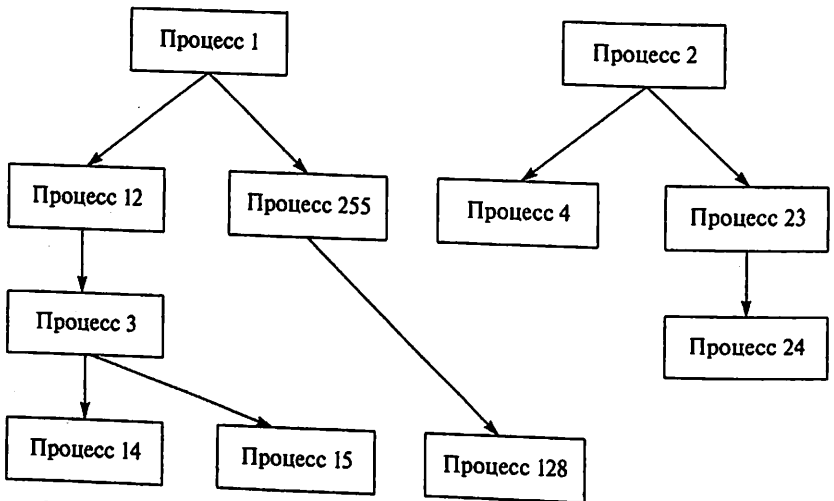

Рис. 2.4. Упрощенный генеалогический лес процессов. Стрелочка означает отношение родитель–ребенок

При рождении процесса система заводит новый PCB с состоянием процесса «рождение» и начинает его заполнять. Новый процесс получает собственный уникальный идентификационный номер. Поскольку для хранения идентификационного номера процесса в операционной системе отводится ограниченное количество битов, для соблюдения уникальности номеров количество одновременно присутствующих в ней процессов должно быть ограничено. После завершения какого-либо процесса его освободившийся идентификационный номер может быть повторно использован для другого процесса.

Обычно для выполнения своих функций процесс-ребенок требует определенных ресурсов: памяти, файлов, устройств ввода-вывода и т.д. Существует два подхода к их выделению. Новый процесс может получить в свое распоряжение некоторую часть родительских ресурсов, возможно разделяя с процессом-родителем и другими процессами-детьми права на них, или может получить свои ресурсы

непосредственно от операционной системы. Информация о выделенных ресурсах заносится в PCB.

После наделения процесса-ребенка ресурсами необходимо занести в его адресное пространство программный код, значения данных, установить программный счетчик. Здесь также возможны два решения. В первом случае процесс-ребенок становится дубликатом процесса-родителя по регистровому и пользовательскому контекстам, при этом должен существовать способ определения, кто для кого из процессов-двойников является родителем. Во втором случае процесс-ребенок загружается новой программой из какого-либо файла.

Операционная система Unix разрешает порождение процесса только первым способом. В ней 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 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, а перед выполнением *execve* — выполнить перенаправление стандартного ввода, стандартного вывода и стандартного вывода сообщений об ошибках.

Операционная система VAX/VMS допускает только второе решение. В Windows 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.

Порождение нового процесса как дубликата процесса-родителя приводит к возможности существования программ (т.е. исполняемых

файлов), для работы которых организуется более одного процесса. Возможность замены пользовательского контекста процесса по ходу его работы (т.е. загрузки для исполнения новой программы) приводит к тому, что в рамках одного и того же процесса может последовательно выполняться несколько различных программ.

После того как процесс наделен содержанием, в PCB дописывается оставшаяся информация и состояние нового процесса изменяется на готовность. Осталось сказать несколько слов о том, как ведут себя процессы-родители после рождения процессов-детей. Процесс-родитель может продолжать свое выполнение одновременно с выполнением процесса-ребенка, а может ожидать завершения работы некоторых или всех своих «детей».

Мы не будем подробно останавливаться на причинах, которые могут привести к завершению жизненного цикла процесса. После того как процесс завершил свою работу, операционная система переводит его в состояние «закончил исполнение» и освобождает все ассоциированные с ним ресурсы, делая соответствующие записи в блоке управления процессом. При этом сам PCB не уничтожается, а остается в системе еще некоторое время. Это связано с тем, что процесс-родитель после завершения процесса-ребенка может запросить операционную систему о причине «смерти» порожденного им процесса и/или статистическую информацию о его работе. Подобная информация сохраняется в PCB отработавшего процесса до запроса процесса-родителя или до конца его деятельности, после чего все следы завершившегося процесса окончательно исчезают из системы. В операционной системе Unix процессы, находящиеся в состоянии «закончил исполнение», принято называть процессами-зомби.

Следует заметить, что в ряде операционных систем (например, в VAX/VMS) гибель процесса-родителя приводит к завершению работы всех его «детей». В других операционных системах (например, в Unix) процессы-дети продолжают свое существование и после окончания работы процесса-родителя. При этом возникает необходимость изменения информации в PCB процессов-детей о породившем их процессе для того, чтобы генеалогический лес процессов оставался целостным. Рассмотрим следующий пример. Пусть процесс с номером 2515 был порожден процессом с номером 2001 и после завершения его работы остается в вычислительной системе неограниченно долго. Тогда не исключено, что номер 2001 будет использован операционной системой повторно для совсем другого процесса. Если не изменить информацию о процессе-родителе для процесса 2515, то генеалогический лес процессов окажется некорректным — процесс 2515 будет считать своим родителем новый процесс 2001, а процесс 2001 будет открещиваться от нежданного потомка. Как правило, «осиротевшие» процессы «усыновляются» одним из системных процессов, который порождается при старте операционной системы и функционирует все время, пока она работает.

Многоразовые операции

Одноразовые операции приводят к изменению количества процессов, находящихся под управлением операционной системы, и всегда связаны с выделением или освобождением определенных ресурсов. Многоразовые операции, напротив, не приводят к изменению количества процессов в операционной системе и не обязаны быть связанными с выделением или освобождением ресурсов.

Кратко опишем действия, которые производит операционная система при выполнении многоразовых операций над процессами.

Запуск процесса. Из числа процессов, находящихся в состоянии «готовность», операционная система выбирает один процесс для последующего исполнения. Для избранного процесса операционная система обеспечивает наличие в оперативной памяти информации, необходимой для его дальнейшего выполнения. Далее состояние процесса изменяется на исполнение, восстанавливаются значения регистров для данного процесса и управление передается команде, на которую указывает счетчик команд процесса. Все данные, необходимые для восстановления контекста, извлекаются из PCB процесса, над которым совершается операция.

Приостановка процесса. Работа процесса, находящегося в состоянии «исполнение», приостанавливается в результате какого-либо прерывания. Процессор автоматически сохраняет счетчик команд и, возможно, один или несколько регистров в стеке исполняемого процесса, а затем передает управление по специальному адресу обработки данного прерывания. На этом деятельность hardware по обработке прерывания завершается. По указанному адресу обычно располагается одна из частей операционной системы. Она сохраняет в его PCB, переводит процесс в состояние «готовность» и приступает к обработке прерывания, т.е. к выполнению определенных действий, связанных с возникшим прерыванием.

Блокирование процесса. Процесс блокируется, когда он не может продолжать работу, не дождавшись возникновения какого-либо события в вычислительной системе. Для этого он обращается к опера-

ционной системе с помощью определенного системного вызова. Операционная система обрабатывает системный вызов (инициализирует операцию ввода-вывода, добавляет процесс в очередь процессов, дожидающихся освобождения устройства или возникновения события, и т.д.) и, при необходимости сохранив нужную часть контекста процесса в его PCB, переводит процесс из состояния «исполнение» в состояние «ожидание».

Разблокирование процесса. После возникновения в системе какого-либо события операционной системе нужно точно определить, какое именно событие произошло. Затем операционная система проверяет, находился ли некоторый процесс в состоянии «ожидание» для данного события, и если находился, переводит его в состояние «готовность», выполняя необходимые действия, связанные с наступлением события (инициализация операции ввода-вывода для очередного ожидающего процесса и т.п.).

Переключение контекста

До сих пор мы рассматривали операции над процессами изолированно, независимо друг от друга. В действительности же деятельность мультипрограммной операционной системы состоит из цепочек операций, выполняемых над различными процессами, и сопровождается переключением процессора с одного процесса на другой.

Давайте для примера упрощенно рассмотрим, как в реальности может протекать операция разблокирования процесса, ожидающего ввода-вывода (рис. 2.5). При исполнении процессором некоторого процесса (на рисунке — процесс 1) возникает прерывание от устройства ввода-вывода, сигнализирующее об окончании операций на устройстве. Над выполняющимся процессом производится операция приостановки. Далее операционная система разблокирует процесс, инициировавший запрос на ввод-вывод (на рисунке — процесс 2), и осуществляет запуск приостановленного или нового процесса, выбранного при выполнении планирования (на рисунке был выбран разблокированный процесс). Как мы видим, в результате обработки информации об окончании операции ввода-вывода возможна смена процесса, находящегося в состоянии «исполнение».

Для корректного переключения процессора с одного процесса на другой необходимо сохранить контекст исполнявшегося процесса и восстановить контекст процесса, на который будет переключен процессор. Такая процедура сохранения/восстановления работоспособности процессов называется переключением контекста. Время, затраченное на переключение контекста, не используется вычислительной системой для совершения полезной работы и представляет собой накладные расходы, снижающие производительность системы. Оно меняется от машины к машине и обычно колеблется в диапазоне от 1 до 1000 микросекунд. Существенно сократить накладные расходы в современных операционных системах позволяет расширенная модель процессов, включающая в себя понятие threads of execution (нити исполнения или просто нити).

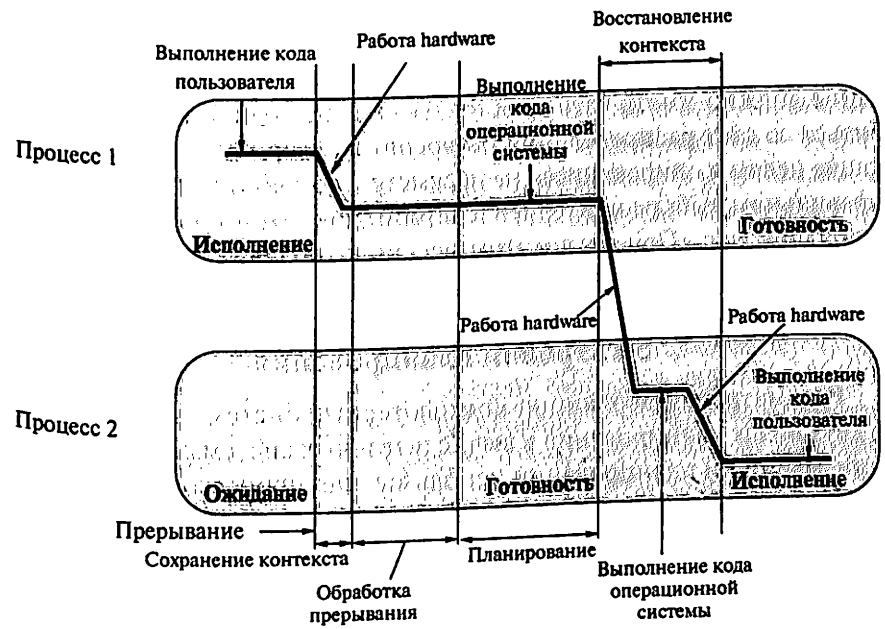

Рис. 2.5. Выполнение операции разблокирования процесса

Глава 3
ПЛАНИРОВАНИЕ ПРОЦЕССОВ

Всякий раз, когда нам приходится иметь дело с ограниченным количеством ресурсов и несколькими их потребителями, будь то фонд заработной платы в трудовом коллективе или отмечание дня рождения с близкими и родными, мы вынуждены заниматься распределением наличных ресурсов между потребителями или, другими словами, планированием использования ресурсов. Такое планирование должно иметь четко поставленные цели (чего мы хотим добиться за счет распределения ресурсов) и алгоритмы, соответствующие целям и опирающиеся на параметры потребителей. Только при правильном выборе критериев и алгоритмов можно избежать таких вопросов, как: «Почему я в этом месяце не получил зарплату?» или «А где моя тарелка?».

3.1. Уровни планирования

Рассматривая эволюцию компьютерных систем, мы говорили о двух видах планирования в вычислительных системах: планировании заданий и планировании использования процессора. Планирование заданий появилось в пакетных системах после того, как для хранения сформированных пакетов заданий начали использоваться магнитные диски. Магнитные диски, являясь устройствами прямого доступа, позволяют загружать задания в компьютер в произвольном порядке, а не только в том, в котором они были записаны на диск. Изменяя порядок загрузки заданий в вычислительную систему, можно повысить эффективность ее использования. Процедуру выбора очередного задания для загрузки в машину, т.е. для порождения соответствующего процесса, мы и назвали *планированием заданий*. Планирование использования процессора впервые возникает в мультипрограммных вычислительных системах, где в состоянии «готовность» могут одновременно находиться несколько процессов. Именно для процедуры выбора из них одного процесса, который получит процессор в свое распоряжение, т.е. будет переведен в состояние «исполнение», мы использовали это словосочетание. Теперь, познакомившись с концепцией процессов в вычислительных системах, оба вида планирования мы будем рассматривать как различные уровни планирования процессов.

Планирование заданий используется в качестве долгосрочного планирования процессов. Оно отвечает за порождение новых процессов в системе, определяя ее степень мультипрограммирования, т.е. количество процессов, одновременно находящихся в ней. Если степень мультипрограммирования системы поддерживается постоянной, т.е. среднее количество процессов в компьютере не меняется, то новые процессы могут появляться только после завершения ранее загруженных. Поэтому долгосрочное планирование осуществляется достаточно редко, между появлением новых процессов могут проходить минуты и даже десятки минут. Решение о выборе для запуска того или иного процесса оказывает влияние на функционирование вычислительной системы на протяжении достаточно длительного времени. Отсюда и название этого уровня планирования — долгосрочное. В некоторых операционных системах долгосрочное планирование сведено к минимуму или отсутствует вовсе. Так, например, во многих интерактивных системах разделения времени порождение процесса происходит сразу после появления соответствующего запроса. Поддержание разумной степени мультипрограммирования осуществляется за счет ограничения количества пользователей, которые могут работать в системе, и особенностей человеческой психологии. Если между нажатием на клавишу и появлением символа на экране проходит 20–30 секунд, то многие пользователи предпочтут прекратить работу и продолжить ее, когда система будет менее загружена.

Планирование использования процессора применяется в качестве краткосрочного планирования процессов. Оно проводится, к примеру, при обращении исполняющегося процесса к устройствам ввода-вывода или просто по завершении определенного интервала времени. Поэтому краткосрочное планирование осуществляется, как правило, не реже одного раза в 100 миллисекунд. Выбор нового процесса для исполнения оказывает влияние на функционирование системы до наступления очередного аналогичного события, т.е. в течение короткого промежутка времени, чем и обусловлено название этого уровня планирования — краткосрочное.

В некоторых вычислительных системах бывает выгодно для повышения производительности временно удалить какой-либо частично выполнившийся процесс из оперативной памяти на диск, а позже вернуть его обратно для дальнейшего выполнения. Такая процедура в англоязычной литературе получила название swapping, что можно перевести на русский язык как «подкачка», хотя в специальной литературе оно употребляется без перевода — свопинг. Когда

и какой из процессов нужно перекачать на диск и вернуть обратно, решается дополнительным промежуточным уровнем планирования процессов — среднесрочным.

3.2. Критерии планирования и требования к алгоритмам

Для каждого уровня планирования процессов можно предложить много различных алгоритмов. Выбор конкретного алгоритма определяется классом задач, решаемых вычислительной системой, и целями, которых мы хотим достичь, используя планирование. Перечислим наиболее основные из них.

- Справедливость — гарантировать каждому заданию или процессу определенную часть времени использования процессора в компьютерной системе, стараясь не допустить возникновения ситуации, когда процесс одного пользователя постоянно занимает процессор, в то время как процесс другого пользователя фактически не начинал выполняться.
- Эффективность — постараться занять процессор на все 100% рабочего времени, не позволяя ему простаивать в ожидании процессов, готовых к исполнению. В реальных вычислительных системах загрузка процессора колеблется от 40 до 90%.
- Сокращение полного времени выполнения (turnaround time) — обеспечить минимальное время между стартом процесса или постановкой задания в очередь для загрузки и его завершением.
- Сокращение времени ожидания (waiting time) — сократить время, которое проводят процессы в состоянии «готовность» и задания в очереди для загрузки.
- Сокращение времени отклика (response time) — минимизировать время, которое требуется процессу в интерактивных системах для ответа на запрос пользователя.

Независимо от поставленных целей планирования желательно также, чтобы алгоритмы обладали следующими свойствами:

- были предсказуемыми. Одно и то же задание должно выполняться приблизительно за одно и то же время. Применение алгоритма планирования не должно приводить, к примеру, к извлечению квадратного корня из 4 за сотые доли секунды при одном запуске и за несколько суток — при втором запуске;
- были связаны с минимальными накладными расходами. Если на каждые 100 миллисекунд, выделенные процессу для использования процессора, будет приходиться 200 миллисекунд на определение того, какой именно процесс получит процессор в свое распоряжение, и на переключение контекста, то такой алгоритм, очевидно, применять не стоит;
- равномерно загружали ресурсы вычислительной системы, отдавая предпочтение тем процессам, которые будут занимать малоиспользуемые ресурсы;
- обладали масштабируемостью, т.е. не сразу теряли работоспособность при увеличении нагрузки. Например, рост количества процессов в системе в два раза не должен приводить к увеличению полного времени выполнения процессов на порядок.

Многие из приведенных выше целей и свойств являются противоречивыми. Улучшая работу алгоритма с точки зрения одного критерия, мы ухудшаем ее с точки зрения другого. Приспосабливая алгоритм под один класс задач, мы тем самым дискриминируем задачи другого класса.

3.3. Параметры планирования

Для осуществления поставленных целей разумные алгоритмы планирования должны опираться на какие-либо характеристики процессов в системе, заданий в очереди на загрузку, состояния самой вычислительной системы, иными словами, на параметры планирования.

Все параметры планирования можно разбить на две большие группы: статические параметры и динамические параметры. Статические параметры не изменяются в ходе функционирования вычислительной системы, динамические же, напротив, подвержены постоянным изменениям.

К статическим параметрам вычислительной системы можно отнести предельные значения ее ресурсов (размер оперативной памяти, максимальное количество памяти на диске для осуществления свопинга, количество подключенных устройств ввода-вывода и т.п.). Динамические параметры системы описывают количество свободных ресурсов на данный момент.

К статическим параметрам процессов относятся характеристики, как правило присущие заданиям уже на этапе загрузки.

- Каким пользователем запущен процесс или сформировано задание.
- Насколько важной является поставленная задача, т.е. каков приоритет ее выполнения.
- Сколько процессорного времени запрошено пользователем для решения задачи.

- Каково соотношение процессорного времени и времени, необходимого для осуществления операций ввода-вывода.
- Какие ресурсы вычислительной системы (оперативная память, устройства ввода-вывода, специальные библиотеки и системные программы и т.д.) и в каком количестве необходимы заданию.

Алгоритмы долгосрочного планирования используют в своей работе статические и динамические параметры вычислительной системы и статические параметры процессов (динамические параметры процессов на этапе загрузки заданий еще не известны). Алгоритмы краткосрочного и среднесрочного планирования дополнительно учитывают и динамические характеристики процессов. Для среднесрочного планирования в качестве таких характеристик может использоваться следующая информация:

- сколько времени прошло с момента выгрузки процесса на диск или его загрузки в оперативную память;
- сколько оперативной памяти занимает процесс;
- сколько процессорного времени уже предоставлено процессу.

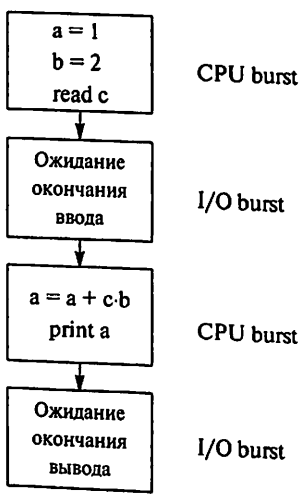

Рис. 3.1. Фрагмент деятельности процесса с выделением промежутков непрерывного использования процессора и ожидания ввода-вывода

Для краткосрочного планирования нам понадобится ввести еще два динамических параметра. Деятельность любого процесса можно представить как последовательность циклов использования процессора и ожидания завершения операций ввода-вывода. Промежуток времени непрерывного использования процессора носит название CPU burst, а промежуток времени непрерывного ожидания ввода-вывода — I/O burst. На рис. 3.1 показан фрагмент деятельности некоторого процесса на псевдоязыке программирования с выделением указанных промежутков. Для краткости мы будем использовать термины «CPU burst» и «I/O burst» без перевода. Значения продолжительности последних и очередных CPU burst и I/O burst являются важными динамическими параметрами процесса.

3.4. Вытесняющее и невытесняющее планирование

Процесс планирования осуществляется частью операционной системы, называемой планировщиком. Планировщик может принимать решения о выборе для исполнения нового процесса из числа находящихся в состоянии «готовность» в следующих четырех случаях.

1. Когда процесс переводится из состояния «исполнение» в состояние «закончил исполнение».
2. Когда процесс переводится из состояния «исполнение» в состояние «ожидание».
3. Когда процесс переводится из состояния «исполнение» в состояние «готовность» (например, после прерывания от таймера).
4. Когда процесс переводится из состояния «ожидание» в состояние «готовность» (завершилась операция ввода-вывода или произошло другое событие).

В случаях 1 и 2 процесс, находившийся в состоянии «исполнение», не может дальше исполняться, и операционная система вынуждена осуществлять планирование, выбирая новый процесс для выполнения. В случаях 3 и 4 планирование может как проводиться, так и не проводиться, планировщик не вынужден обязательно принимать решение о выборе процесса для выполнения, процесс, находившийся в состоянии «исполнение», может просто продолжить свою работу. Если в операционной системе планирование осуществляется только в вынужденных ситуациях, говорят, что имеет место *невытесняющее* (nonpreemptive) *планирование*. Если планировщик принимает и вынужденные, и невынужденные решения, говорят о *вытесняющем* (preemptive) *планировании*. Термин «вытесняющее планирование» возник потому, что исполняющийся процесс помимо своей воли может быть вытеснен из состояния «исполнение» другим процессом. Невытесняющее планирование используется, например, в MS Windows 3.1 и ОС Apple Macintosh. При таком режиме планирования процесс занимает столько процессорного времени, сколько ему необходимо. При этом переключение процессов возникает только при желании самого исполняющегося процесса передать управление (для ожидания

завершения операции ввода-вывода или по окончании работы). Этот метод планирования относительно просто реализуем и достаточно эффективен, так как позволяет выделить большую часть процессорного времени для работы самих процессов и до минимума сократить затраты на переключение контекста. Однако при невытесняющем планировании возникает проблема возможности полного захвата процессора одним процессом, который вследствие каких-либо причин (например, из-за ошибки в программе) зацикливается и не может передать управление другому процессу. В такой ситуации спасает только перезагрузка всей вычислительной системы.

Вытесняющее планирование обычно используется в системах разделения времени. В этом режиме планирования процесс может быть приостановлен в любой момент исполнения. Операционная система устанавливает специальный таймер для генерации сигнала прерывания по истечении некоторого интервала времени — кванта. После прерывания процессор передается в распоряжение следующего процесса. Временные прерывания помогают гарантировать приемлемое время отклика процессов для пользователей, работающих в диалоговом режиме, и предотвращают «зависание» компьютерной системы из-за зацикливания какой-либо программы.

3.5. Алгоритмы планирования

Существует достаточно большой набор разнообразных алгоритмов планирования, которые предназначены для достижения различных целей и эффективны для разных классов задач. Многие из них могут использоваться на нескольких уровнях планирования. Здесь мы рассмотрим некоторые наиболее употребительные алгоритмы применительно к процессу кратковременного планирования.

First-Come, First-Served (FCFS)

Простейшим алгоритмом планирования является алгоритм, который принято обозначать аббревиатурой FCFS по первым буквам его английского названия — First-Come, First-Served (первым пришел, первым обслужен). Представим себе, что процессы, находящиеся в состоянии «готовность», выстроены в очередь. Когда процесс переходит в состояние «готовность», он, а точнее ссылка на его PCB, помещается в конец этой очереди. Выбор нового процесса для исполнения осуществляется из начала очереди с удалением оттуда ссылки на его PCB. Очередь подобного типа имеет в программировании специальное наименование — FIFO, сокращение от First In, First Out (первым вошел, первым вышел).

Такой алгоритм выбора процесса осуществляет невытесняющее планирование. Процесс, получивший в свое распоряжение процессор, занимает его до истечения текущего CPU burst. После этого для выполнения выбирается новый процесс из начала очереди.

Преимуществом алгоритма FCFS является легкость его реализации, но в то же время он имеет и много недостатков. Рассмотрим следующий пример. Пусть в состоянии «готовность» находятся три процесса $p0$, $p1$ и $p2$, для которых известны времена их очередных CPU burst. Эти времена приведены в табл. 3.1 в некоторых условных единицах. Для простоты будем полагать, что вся деятельность процессов ограничивается использованием только одного промежутка CPU burst, что процессы не совершают операций ввода-вывода и что время переключения контекста так мало, что им можно пренебречь.

Таблица 3.1

Процесс	$p0$	$p1$	$p2$
Продолжительность	13	4	1

Если процессы расположены в очереди процессов, готовых к исполнению, в порядке $p0$, $p1$, $p2$, то картина их выполнения выглядит так, как показано на рис. 3.2. Первым для выполнения выбирается процесс $p0$, который получает процессор на все время своего CPU burst, т.е. на 13 единиц времени. После его окончания в состояние «исполнение» переводится процесс $p1$, он занимает процессор на 4 единицы времени. И наконец, возможность работать получает процесс $p2$. Время ожидания для процесса $p0$ составляет 0 единиц времени, для процесса $p1$ — 13 единиц, для процесса $p2$ — 13 + 4 = = 17 единиц. Таким образом, среднее время ожидания в этом случае — (0 + 13 + 17)/3 = 10 единиц времени. Полное время выполнения для процесса $p0$ составляет 13 единиц времени, для процесса

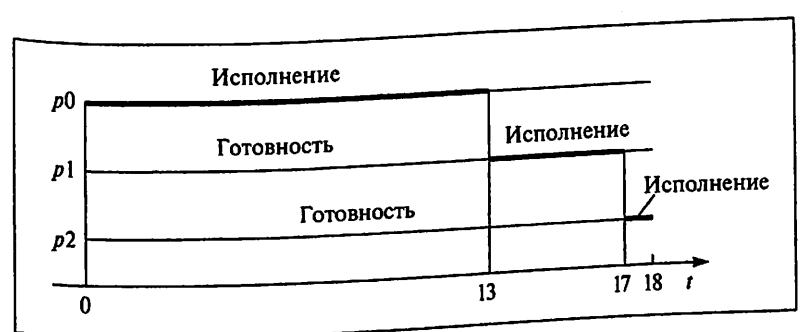

Рис. 3.2. Выполнение процессов при порядке $p0$, $p1$, $p2$

$p1$ — 13 + 4 = 17 единиц, для процесса $p2$ — 13 + 4 + 1 = 18 единиц. Среднее полное время выполнения оказывается равным (13 + 17 + + 18)/3 = 16 единицам времени.

Если те же самые процессы расположены в порядке $p2$, $p1$, $p0$, то картина их выполнения будет соответствовать рис. 3.3. Время ожидания для процесса $p0$ равняется 5 единицам времени, для процесса $p1$ — 1 единице, для процесса $p2$ — 0 единиц. Среднее время ожидания составит (5 + 1 + 0)/3 = 2 единицы времени. Это в 5 (!) раз меньше, чем в предыдущем случае. Полное время выполнения для процесса $p0$ получается равным 18 единицам времени, для процесса $p1$ — 5 единицам, для процесса $p2$ — 1 единице. Среднее полное время выполнения составляет (18 + 5 + 1)/3 = 8 единиц времени, что почти в 2 раза меньше, чем при первой расстановке процессов.

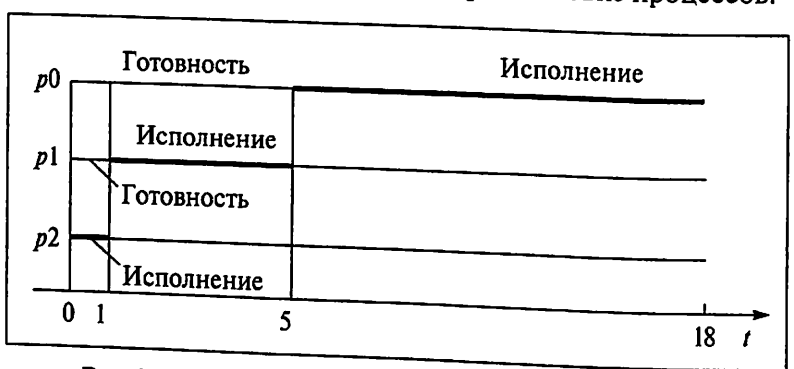

Рис. 3.3. Выполнение процессов при порядке $p2$, $p1$, $p0$

Как мы видим, среднее время ожидания и среднее полное время выполнения для этого алгоритма существенно зависят от порядка расположения процессов в очереди. Если у нас есть процесс с длительным CPU burst, то короткие процессы, перешедшие в состояние «готовность» после длительного процесса, будут очень долго ждать начала выполнения. Поэтому алгоритм FCFS практически неприменим для систем разделения времени — слишком большим получается среднее время отклика в интерактивных процессах.

Round Robin (RR)

Модификацией алгоритма FCFS является алгоритм, получивший название «Round Robin» (Round Robin — это вид детской карусели в США) или сокращенно RR. По сути дела, это тот же самый алгоритм, только реализованный в режиме вытесняющего планирования. Можно представить себе все множество готовых процессов организованным циклически — процессы сидят на карусели. Карусель вращается так, что каждый процесс находится около процессора небольшой фиксированный квант времени, обычно 10–100 миллисекунд (рис. 3.4). Пока процесс находится рядом с процессором, он получает процессор в свое распоряжение и может исполняться.

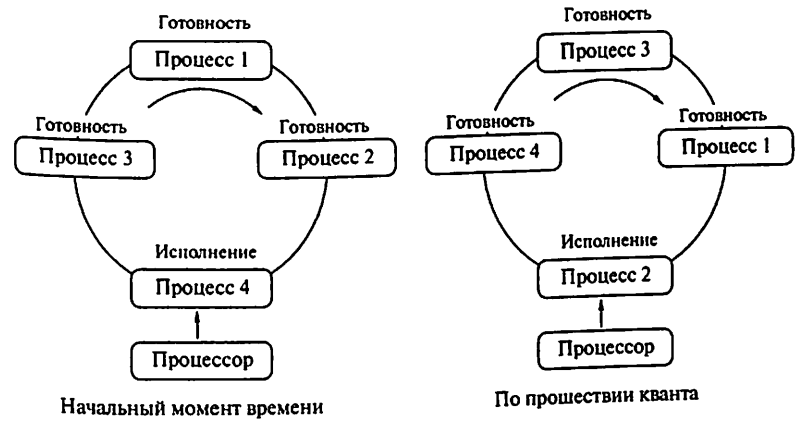

Рис. 3.4. Процессы на карусели

Реализуется такой алгоритм так же, как и предыдущий, с помощью организации процессов, находящихся в состоянии «готовность», в очередь FIFO. Планировщик выбирает для очередного исполнения процесс, расположенный в начале очереди, и устанавливает таймер для генерации прерывания по истечении определенного кванта времени. При выполнении процесса возможны два варианта.

Время непрерывного использования процессора, необходимое процессу (остаток текущего CPU burst), меньше или равно продолжительности кванта времени. Тогда процесс по своей воле освобождает процессор до истечения кванта времени, на исполнение поступает новый процесс из начала очереди, и таймер начинает отсчет кванта заново.

Продолжительность остатка текущего CPU burst процесса больше, чем квант времени. Тогда по истечении этого кванта процесс прерывается таймером и помещается в конец очереди процессов, готовых к исполнению, а процессор выделяется для использования процессу, находящемуся в ее начале.

Рассмотрим предыдущий пример с порядком процессов $p0$, $p1$, $p2$ и величиной кванта времени, равной 4.

Выполнение этих процессов иллюстрируется табл. 3.2. Обозначение «И» используется в ней для процесса, находящегося в состоянии «исполнение», обозначение «Г» — для процессов в со-

стоянии «готовность», пустые ячейки соответствуют завершившимся процессам. Состояния процессов показаны на протяжении соответствующей единицы времени, т.е. колонка с номером 1 соответствует промежутку времени от 0 до 1.

Таблица 3.2

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
p_0	И	И	И	И	Г	Г	Г	Г	Г	И	И	И	И	И	И	И	И	И
p_1	Г	Г	Г	Г	И	И	И	И										
p_2	Г	Г	Г	Г	Г	Г	Г	Г	И									

Первым для исполнения выбирается процесс p_0. Продолжительность его CPU burst больше, чем величина кванта времени, и поэтому процесс исполняется до истечения кванта, т.е. в течение 4 единиц времени. После этого он помещается в конец очереди готовых к исполнению процессов, которая принимает вид p_1, p_2, p_0. Следующим начинает выполняться процесс p_1. Время его исполнения совпадает с величиной выделенного кванта, поэтому процесс работает до своего завершения. Теперь очередь процессов в состоянии «готовность» состоит из двух процессов, p_2 и p_0. Процессор выделяется процессу p_2. Он завершается до истечения отпущенного ему процессорного времени, и очередные кванты отмеряются процессу p_0 — единственному не закончившему к этому моменту свою работу. Время ожидания для процесса p_0 (количество символов «Г» в соответствующей строке) составляет 5 единиц времени, для процесса p_1 — 4 единицы времени, для процесса p_2 — 8 единиц времени. Таким образом, среднее время ожидания для этого алгоритма получается равным (5 + 4 + 8) / 3 = 5,6 (6) единицы времени. Полное время выполнения для процесса p_0 (количество непустых столбцов в соответствующей строке) составляет 18 единиц времени, для процесса p_1 — 8 единиц, для процесса p_2 — 9 единиц. Среднее полное время выполнения оказывается равным (18 + 8 + 9) / 3 = 11,6 (6) единицы времени.

Легко увидеть, что среднее время ожидания и среднее полное время выполнения для обратного порядка процессов не отличаются от соответствующих времен для алгоритма FCFS и составляют 2 и 6 единиц времени соответственно.

На производительность алгоритма RR сильно влияет величина кванта времени. Рассмотрим тот же самый пример с порядком процессов p_0, p_1, p_2 для величины кванта времени, равной 1 (табл. 3.3).

Время ожидания для процесса p_0 составит 5 единиц времени, для процесса p_1 — тоже 5 единиц, для процесса p_2 — 2 единицы. В этом случае среднее время ожидания получается равным (5 + 5 + 2) / 3 = 4 единицам времени. Среднее полное время исполнения составит (18 + 9 + 3) / 3 = 10 единиц времени.

Таблица 3.3

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
p_0	И	Г	Г	И	Г	И	Г	И	Г	И	И	И	И	И	И	И	И	И
p_1	Г	И	Г	Г	И	Г	И	Г	И									
p_2	Г	Г	И															

При очень больших величинах кванта времени, когда каждый процесс успевает завершить свой CPU burst до возникновения прерывания по времени, алгоритм RR вырождается в алгоритм FCFS. При очень малых величинах создается иллюзия того, что каждый из n процессов работает на собственном виртуальном процессоре с производительностью ~ $1/n$ от производительности реального процессора. Правда, это справедливо лишь при теоретическом анализе при условии пренебрежения временами переключения контекста процессов. В реальных условиях при слишком малой величине кванта времени и, соответственно, слишком частом переключении контекста накладные расходы на переключение резко снижают производительность системы.

Shortest-Job-First (SJF)

При рассмотрении алгоритмов FCFS и RR мы видели, насколько существенным для них является порядок расположения процессов в очереди процессов, готовых к исполнению. Если короткие задачи расположены в очереди ближе к ее началу, то общая производительность этих алгоритмов значительно возрастает. Если бы мы знали время следующих CPU burst для процессов, находящихся в состоянии «готовность», то могли бы выбрать для исполнения не процесс из начала очереди, а процесс с минимальной длительностью CPU burst. Если же таких процессов два или больше, то для выбора одного из них можно использовать уже известный нам алгоритм FCFS. Квантование времени при этом не применяется. Описанный алгоритм получил название *«кратчайшая работа первой»* или Shortest Job First (SJF).

SJF-алгоритм краткосрочного планирования может быть как вытесняющим, так и невытесняющим. При невытесняющем SJF-планировании процессор предоставляется избранному процессу на все необходимое ему время, независимо от событий, происходящих в вычислительной системе. При вытесняющем SJF-планировании учитывается появление новых процессов в очереди готовых к исполнению (из числа вновь родившихся или разблокированных) во время работы выбранного процесса. Если CPU burst нового процесса меньше, чем остаток CPU burst у исполняющегося, то исполняющийся процесс вытесняется новым.

Рассмотрим пример работы невытесняющего алгоритма SJF. Пусть в состоянии «готовность» находятся четыре процесса, p_0, p_1, p_2 и p_3, для которых известны времена их очередных CPU burst. Эти времена приведены в табл. 3.4. Как и прежде, будем полагать, что вся деятельность процессов ограничивается использованием только одного промежутка CPU burst, что процессы не совершают операций ввода-вывода и что временем переключения контекста можно пренебречь.

Таблица 3.4

Продолжительность CPU burst процессов

Процесс	p_0	p_1	p_2	p_3
Продолжительность очередного CPU burst	5	3	7	1

При использовании невытесняющего алгоритма SJF первым для исполнения будет выбран процесс p_3, имеющий наименьшее значение продолжительности очередного CPU burst. После его завершения для исполнения выбирается процесс p_1, затем p_0 и, наконец, p_2. Эта картина отражена в табл. 3.5.

Таблица 3.5

Пример работы алгоритма SJF

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
p_0	Г	Г	Г	Г	И	И	И	И	И							
p_1	Г	И	И	И												
p_2	Г	Г	Г	Г	Г	Г	Г	Г	Г	И	И	И	И	И	И	И
p_3	И															

Как мы видим, среднее время ожидания для алгоритма SJF составляет $(4 + 1 + 9 + 0)/4 = 3{,}5$ единицы времени. Легко посчитать, что для алгоритма FCFS при порядке процессов p_0, p_1, p_2, p_3 эта величина будет равна $(0 + 5 + 8 + 15)/4 = 7$ единицам времени, т.е. будет в 2 раза больше, чем для алгоритма SJF. Можно показать, что для заданного набора процессов (если в очереди не появляются новые процессы) алгоритм SJF является оптимальным с точки зрения минимизации среднего времени ожидания среди класса невытесняющих алгоритмов.

Для рассмотрения примера вытесняющего SJF планирования мы возьмем ряд процессов p_0, p_1, p_2 и p_3 с различными временами CPU burst и различными моментами их появления в очереди процессов, готовых к исполнению (табл. 3.6).

Таблица 3.6

Пример параметров процессов

Процесс	Время появления в очереди очередного CPU burst	Продолжительность
p_0	0	6
p_1	2	2
p_2	6	7
p_3	0	5

В начальный момент времени в состоянии «готовность» находятся только два процесса, p_0 и p_3. Меньшее время очередного CPU burst оказывается у процесса p_3, поэтому он и выбирается для исполнения (табл. 3.7). По прошествии 2 единиц времени в систему поступает процесс p_1. Время его CPU burst меньше, чем остаток CPU burst у процесса p_3, который вытесняется из состояния «исполнение» и переводится в состояние «готовность». По прошествии еще 2 единиц времени процесс p_1 завершается, и для исполнения вновь выбирается процесс p_3. В момент времени $t = 6$ в очереди процессов, готовых к исполнению, появляется процесс p_2, но поскольку ему для работы нужно 7 единиц времени, а процессу p_3 осталось трудиться всего 1 единицу времени, то процесс p_3 остается в состоянии «исполнение». После его завершения в момент времени $t = 7$ в очереди находятся процессы p_0 и p_2, из которых выбирается процесс p_0. Наконец, последним получит возможность выполняться процесс p_2.

Таблица 3.7

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
p_0	Г	Г	Г	Г	Г	Г	Г	И	И	И	И	И	И							
p_1			И	И																
p_2								Г	Г	Г	Г	Г	Г	И	И	И	И	И	И	И
p_3	И	И	Г	Г	И	И	И													

Основную сложность при реализации алгоритма SJF представляет невозможность точного знания продолжительности очередного CPU burst для исполняющихся процессов. В пакетных системах количество процессорного времени, необходимое заданию для выполнения, указывает пользователь при формировании задания. Мы можем брать эту величину для осуществления долгосрочного SJF-планирования. Если пользователь укажет больше времени, чем ему нужно, он будет ждать результата дольше, чем мог бы, так как задание будет загружено в систему позже. Если же он укажет меньшее количество времени, задача может не досчитаться до конца. Таким образом, в пакетных системах решение задачи оценки времени использования процессора перекладывается на плечи пользователя. При краткосрочном планировании мы можем делать только прогноз длительности следующего CPU burst, исходя из предыстории работы процесса. Пусть $\tau(n)$ — величина n-го CPU burst, $T(n+1)$ — предсказываемое значение для $n+1$-го CPU burst, α — некоторая величина в диапазоне от 0 до 1.

Определим рекуррентное соотношение

$$T(n+1) = \alpha \cdot \tau(n) + (1-\alpha) \cdot T(n).$$

$T(0)$ положим произвольной константой. Первое слагаемое учитывает последнее поведение процесса, тогда как второе слагаемое учитывает его предысторию. При $\alpha = 0$ мы перестаем следить за последним поведением процесса, фактически полагая

$$T(n) = T(n+1) = \ldots = T(0),$$

т.е. оценивая все CPU burst одинаково, исходя из некоторого начального предположения.

Положив $\alpha = 1$, мы забываем о предыстории процесса. В этом случае мы полагаем, что время очередного CPU burst будет совпадать со временем последнего CPU burst:

$$T(n+1) = \tau(n).$$

Обычно выбирают $\alpha = 1/2$ для равноценного учета последнего поведения и предыстории. Надо отметить, что такой выбор удобен и для быстрой организации вычисления оценки $T(n+1)$. Для подсчета новой оценки нужно взять старую оценку, сложить с измеренным временем CPU burst и полученную сумму разделить на 2, например, сдвинув ее на 1 бит вправо. Полученные оценки $T(n+1)$ применяются как продолжительности очередных промежутков времени непрерывного использования процессора для краткосрочного SJF-планирования.

Гарантированное планирование

При интерактивной работе N пользователей в вычислительной системе можно применить алгоритм планирования, который гарантирует, что каждый из пользователей будет иметь в своем распоряжении $\sim 1/N$ часть процессорного времени. Пронумеруем всех пользователей от 1 до N. Для каждого пользователя с номером i введем две величины: T_i — время нахождения пользователя в системе или, другими словами, длительность сеанса его общения с машиной и τ_i — суммарное процессорное время, уже выделенное всем его процессам в течение сеанса. Справедливым для пользователя было бы получение T_i/N процессорного времени. Если $\tau_i < T_i/N$, то i-й пользователь несправедливо обделен процессорным временем. Если же $\tau_i > T_i/N$, то система явно благоволит к пользователю с номером i. Вычислим для процессов каждого пользователя значение коэффициента справедливости $\tau_i \cdot N/T_i$ и будем предоставлять очередной квант времени готовому процессу с наименьшей величиной этого отношения. Предложенный алгоритм называют алгоритмом гарантированного планирования. К недостаткам этого алгоритма можно отнести невозможность предугадать поведение пользователей. Если некоторый пользователь отправится на пару часов пообедать и поспать, не прерывая сеанса работы, то по возвращении его процессы будут получать неоправданно много процессорного времени.

Приоритетное планирование

Алгоритмы SJF и гарантированного планирования представляют собой частные случаи приоритетного планирования. При приоритетном планировании каждому процессу присваивается определенное числовое значение — приоритет, в соответствии с которым ему выделяется процессор. Процессы с одинаковыми приоритетами планируются в порядке FCFS. Для алгоритма SJF в качестве такого

приоритета выступает оценка продолжительности следующего CPU burst. Чем меньше значение этой оценки, тем более высокий приоритет имеет процесс. Для алгоритма гарантированного планирования приоритетом служит вычисленный коэффициент справедливости. Чем он меньше, тем больше у процесса приоритет.

Алгоритмы назначения приоритетов процессов могут опираться как на внутренние параметры, связанные с происходящим внутри вычислительной системы, так и на внешние по отношению к ней. К внутренним параметрам относятся различные количественные и качественные характеристики процесса, такие как ограничения по времени использования процессора, требования к размеру памяти, число открытых файлов и используемых устройств ввода-вывода, отношение средних продолжительностей I/O burst к CPU burst и т.д. Алгоритмы SJF и гарантированного планирования используют внутренние параметры. В качестве внешних параметров могут выступать важность процесса для достижения каких-либо целей, стоимость оплаченного процессорного времени и другие факторы.

Планирование с использованием приоритетов может быть как вытесняющим, так и невытесняющим. При вытесняющем планировании процесс с более высоким приоритетом, появившийся в очереди готовых процессов, вытесняет исполняющийся процесс с более низким приоритетом. В случае невытесняющего планирования он просто становится в начало очереди готовых процессов. Давайте рассмотрим примеры использования различных режимов приоритетного планирования.

Пусть в очередь процессов, находящихся в состоянии «готовность», поступают те же процессы, что и в примере для вытесняющего алгоритма SJF, только им дополнительно еще присвоены приоритеты (табл. 3.8). В вычислительных системах не существует определенного соглашения, какое значение приоритета — 1 или 4 — считать более приоритетным. Во избежание путаницы, во всех наших примерах мы будем предполагать, что большее значение соответствует меньшему приоритету, т.е. наиболее приоритетным в нашем примере является процесс p_3, а наименее приоритетным — процесс p_0.

Таблица 3.8

Процесс	Время появления в очереди	Продолжительность очередного CPU burst	Приоритет
p_0	0	6	4
p_1	2	2	3

Окончание табл. 3.8

Процесс	Время появления в очереди	Продолжительность очередного CPU burst	Приоритет
p_2	6	7	2
p_3	0	5	1

Как будут вести себя процессы при использовании невытесняющего приоритетного планирования? Первым для выполнения в момент времени $t = 0$ выбирается процесс p_3 как обладающий наивысшим приоритетом. После его завершения в момент времени $t = 5$ в очереди процессов, готовых к исполнению, окажутся два процесса p_0 и p_1. Больший приоритет из них у процесса p_1, он и начнет выполняться (табл. 3.9). Затем в момент времени $t = 8$ для исполнения будет избран процесс p_2, и лишь потом — процесс p_0.

Таблица 3.9

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
p_0	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	И	И	И	И	И	И
p_1			Г	Г	Г	И	И													
p_2							Г	И	И	И	И	И	И	И						
p_3	И	И	И	И	И															

Иным будет предоставление процессора процессам в случае вытесняющего приоритетного планирования (табл. 3.10). Первым, как и в предыдущем случае, начнет исполняться процесс p_3, а по его окончании — процесс p_1. Однако в момент времени $t = 6$ он будет вытеснен процессом p_2 и продолжит свое выполнение только в момент времени $t = 13$. Последним, как и раньше, будет исполняться процесс p_0.

Таблица 3.10

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
p_0	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	И	И	И	И	И	И
p_1			Г	Г	Г	И	Г	Г	Г	Г	Г	Г	Г	И						
p_2							И	И	И	И	И	И	И							
p_3	И	И	И	И	И															

В рассмотренном выше примере приоритеты процессов с течением времени не изменялись. Такие приоритеты принято называть статическими. Механизмы статической приоритетности легко реализовать, и они сопряжены с относительно небольшими издержками на выбор наиболее приоритетного процесса. Однако статические приоритеты не реагируют на изменения ситуации в вычислительной системе, которые могут сделать желательной корректировку порядка исполнения процессов. Более гибкими являются динамические приоритеты процессов, изменяющие свои значения по ходу исполнения процессов. Начальное значение динамического приоритета, присвоенное процессу, действует в течение лишь короткого периода времени, после чего ему назначается новое, более подходящее значение. Изменение динамического приоритета процесса является единственной операцией над процессами, которую мы до сих пор не рассмотрели. Как правило, изменение приоритета процессов проводится согласованно с совершением каких-либо других операций: при рождении нового процесса, при разблокировке или блокировании процесса, по истечении определенного кванта времени или по завершении процесса. Примерами алгоритмов с динамическими приоритетами являются алгоритм SJF и алгоритм гарантированного планирования. Схемы с динамической приоритетностью гораздо сложнее в реализации и связаны с большими издержками по сравнению со статическими схемами. Однако их использование предполагает, что эти издержки оправдываются улучшением работы системы.

Главная проблема приоритетного планирования заключается в том, что при ненадлежащем выборе механизма назначения и изменения приоритетов низкоприоритетные процессы могут не запускаться неопределенно долгое время. Обычно случается одно из двух. Или они все же дожидаются своей очереди на исполнение, или вычислительную систему приходится выключать и они теряются (при остановке IBM 7094 в Массачусетском технологическом институте в 1973 г. были найдены процессы, запущенные в 1967 г. и ни разу с тех пор не исполнявшиеся). Решение этой проблемы может быть достигнуто с помощью увеличения со временем значения приоритета процесса, находящегося в состоянии «готовность». Пусть изначально процессам присваиваются приоритеты от 128 до 255. Каждый раз по истечении определенного промежутка времени значения приоритетов готовых процессов уменьшаются на 1. Процессу, побывавшему в состоянии «исполнение», присваивается первоначальное значение приоритета. Даже такая грубая схема гарантирует, что любому процессу в разумные сроки будет предоставлено право на исполнение.

Многоуровневые очереди (Multilevel Queue)

Для систем, в которых процессы могут быть легко рассортированы по разным группам, был разработан другой класс алгоритмов планирования. Для каждой группы процессов создается своя очередь процессов, находящихся в состоянии «готовность» (рис. 3.5). Этим очередям приписываются фиксированные приоритеты. Например, приоритет очереди системных процессов устанавливается выше, чем приоритет очередей пользовательских процессов. А приоритет очереди процессов, запущенных студентами, ниже, чем для очереди процессов, запущенных преподавателями. Это значит, что ни один пользовательский процесс не будет выбран для исполнения, пока есть хоть один готовый системный процесс, и ни один студенческий процесс не получит в свое распоряжение процессор, если есть процессы преподавателей, готовые к исполнению. Внутри этих очередей для планирования могут применяться самые разные алгоритмы. Так, например, для больших счетных процессов, не требующих взаимодействия с пользователем (фоновых процессов), может использоваться алгоритм FCFS, а для интерактивных процессов — алгоритм RR. Подобный подход, получивший название *многоуровневых очередей*, повышает гибкость планирования: для процессов с различными характеристиками применяется наиболее подходящий им алгоритм.

Рис. 3.5. Несколько очередей планирования

Многоуровневые очереди с обратной связью (Multilevel Feedback Queue)

Дальнейшим развитием алгоритма многоуровневых очередей является добавление к нему механизма обратной связи. Здесь процесс не постоянно приписан к определенной очереди, а может мигрировать из одной очереди в другую в зависимости от своего поведения.

Для простоты рассмотрим ситуацию, когда процессы в состоянии «готовность» организованы в 4 очереди, как на рис. 3.6. Планирование процессов между очередями осуществляется на основе вытесняющего приоритетного механизма. Чем выше на рисунке располагается очередь, тем выше ее приоритет. Процессы в очереди 1 не могут исполняться, если в очереди 0 есть хотя бы один процесс. Процессы в очереди 2 не будут выбраны для выполнения, пока есть хоть один процесс в очередях 0 и 1. И наконец, процесс в очереди 3 может получить процессор в свое распоряжение только тогда, когда очереди 0, 1 и 2 пусты. Если при работе процесса появляется другой процесс

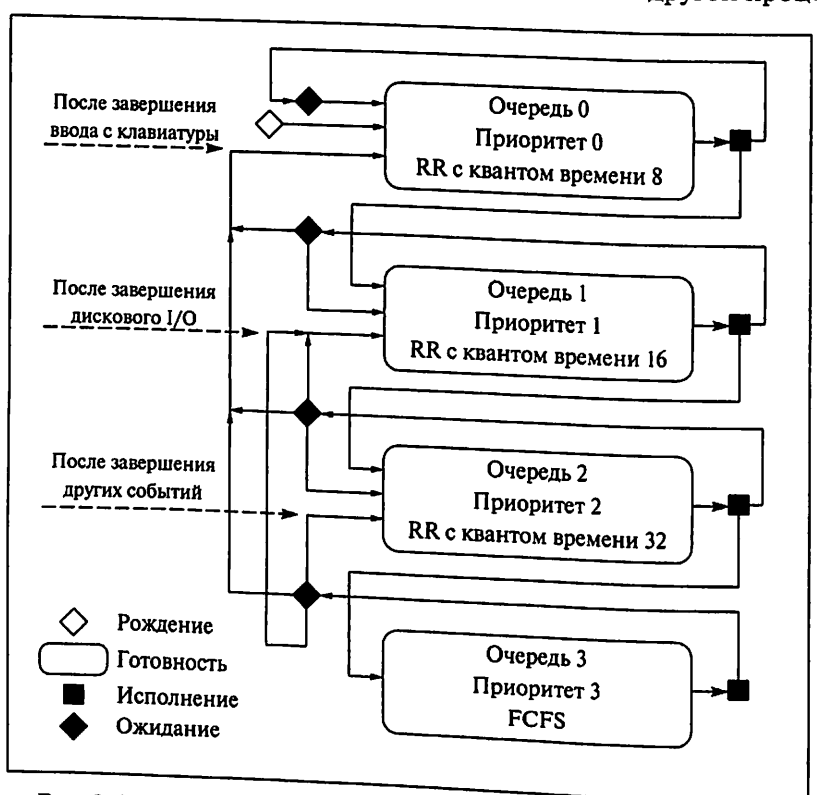

Рис. 3.6. Схема миграции процессов в многоуровневых очередях планирования с обратной связью

в какой-либо более приоритетной очереди, исполняющийся процесс вытесняется новым. Планирование процессов внутри очередей 0–2 осуществляется с использованием алгоритма RR, планирование процессов в очереди 3 основывается на алгоритме FCFS.

Родившийся процесс поступает в очередь 0. При выборе на исполнение он получает в свое распоряжение квант времени размером 8 единиц. Если продолжительность его CPU burst меньше этого кванта времени, процесс остается в очереди 0. В противном случае он переходит в очередь 1. Для процессов из очереди 1 квант времени имеет величину 16. Если процесс не укладывается в это время, он переходит в очередь 2. Если укладывается — остается в очереди 1. В очереди 2 величина кванта времени составляет 32 единицы. Если для непрерывной работы процесса и этого мало, процесс поступает в очередь 3, для которой квантование времени не применяется и, при отсутствии готовых процессов в других очередях, может исполняться до окончания своего CPU burst. Чем больше значение продолжительности CPU burst, тем в менее приоритетную очередь попадает процесс, но тем на большее процессорное время он может рассчитывать. Таким образом, через некоторое время все процессы, требующие малого времени работы процессора, окажутся размещенными в высокоприоритетных очередях, а все процессы, требующие большого счета и с низкими запросами к времени отклика, — в низкоприоритетных.

Миграция процессов в обратном направлении может осуществляться по различным принципам. Например, после завершения ожидания ввода с клавиатуры процессы из очередей 1, 2 и 3 могут помещаться в очередь 0, после завершения дисковых операций ввода-вывода процессы из очередей 2 и 3 могут помещаться в очередь 1, а после завершения ожидания всех других событий — из очереди 3 в очередь 2. Перемещение процессов из очередей с низкими приоритетами в очереди с высокими приоритетами позволяет более полно учитывать изменение поведения процессов с течением времени.

Многоуровневые очереди с обратной связью представляют собой наиболее общий подход к планированию процессов из числа подходов, рассмотренных нами. Они наиболее трудны в реализации, но в то же время обладают наибольшей гибкостью. Понятно, что существует много других разновидностей такого способа планирования, помимо варианта, приведенного выше. Для полного описания их конкретного воплощения необходимо указать:

- количество очередей для процессов, находящихся в состоянии «готовность»;

- алгоритм планирования, действующий между очередями;
- алгоритмы планирования, действующие внутри очередей;
- правила помещения родившегося процесса в одну из очередей;
- правила перевода процессов из одной очереди в другую.

Изменяя какой-либо из перечисленных пунктов, мы можем существенно менять поведение вычислительной системы.

3.6. Взаимодействующие процессы

Для достижения поставленной цели различные процессы (возможно, даже принадлежащие разным пользователям) могут исполняться псевдопараллельно на одной вычислительной системе или параллельно на разных вычислительных системах, взаимодействуя между собой.

Для чего процессам нужно заниматься совместной деятельностью? Какие существуют причины для их кооперации?

1. Повышение скорости работы. Пока один процесс ожидает наступления некоторого события (например, окончания операции ввода-вывода), другие могут заниматься полезной работой, направленной на решение общей задачи. В многопроцессорных вычислительных системах программа разбивается на отдельные кусочки, каждый из которых будет исполняться на своем процессоре.
2. Совместное использование данных. Различные процессы могут, к примеру, работать с одной и той же динамической базой данных или с разделяемым файлом, совместно изменяя их содержимое.
3. Модульная конструкция какой-либо системы. Типичным примером может служить микроядерный способ построения операционной системы, когда различные ее части представляют собой отдельные процессы, взаимодействующие путем передачи сообщений через микроядро.

Наконец, это может быть необходимо просто для удобства работы пользователя, желающего, например, редактировать и отлаживать программу одновременно. В этой ситуации процессы редактора и отладчика должны уметь взаимодействовать друг с другом.

Процессы не могут взаимодействовать, не общаясь, т.е. не обмениваясь информацией. «Общение» процессов обычно приводит к изменению их поведения в зависимости от полученной информации. Если деятельность процессов остается неизменной при любой принятой ими информации, то это означает, что они на самом деле в «общении» не нуждаются. Процессы, которые влияют на поведение друг друга путем обмена информацией, принято называть *кооперативными* или *взаимодействующими процессами*, в отличие от *независимых процессов*, не оказывающих друг на друга никакого воздействия.

Различные процессы в вычислительной системе изначально представляют собой обособленные сущности. Работа одного процесса не должна приводить к нарушению работы другого процесса. Для этого, в частности, разделены их адресные пространства и системные ресурсы, и для обеспечения корректного взаимодействия процессов требуются специальные средства и действия операционной системы. Нельзя просто поместить значение, вычисленное в одном процессе, в область памяти, соответствующую переменной в другом процессе, не предприняв каких-либо дополнительных усилий.

3.7. Категории средств обмена информацией

Процессы могут взаимодействовать друг с другом, только обмениваясь информацией. По объему передаваемой информации и степени возможного воздействия на поведение другого процесса все средства такого обмена можно разделить на три категории.

- *Сигнальные.* Передается минимальное количество информации — один бит, «да» или «нет». Используются, как правило, для извещения процесса о наступлении какого-либо события. Степень воздействия на поведение процесса, получившего информацию, минимальна. Все зависит от того, знает ли он, что означает полученный сигнал, надо ли на него реагировать и каким образом. Неправильная реакция на сигнал или его игнорирование могут привести к неприятным последствиям.
- *Канальные.* «Общение» процессов происходит через линии связи, предоставленные операционной системой, и напоминает общение людей по телефону, с помощью записок, писем или объявлений. Объем передаваемой информации в единицу времени ограничен пропускной способностью линий связи. С увеличением количества информации возрастает и возможность влияния на поведение другого процесса.
- *Разделяемая память.* Два или более процессов могут совместно использовать некоторую область адресного пространства. Создание разделяемой памяти занимается операционная система (если, конечно, ее об этом попросят). Возможность обмена информацией максимальна, как, впрочем, и влияние на поведение другого процесса, но требует повышенной осторожности. Использование разделяемой памяти для передачи/получения ин-

формации осуществляется с помощью средств обычных языков программирования, в то время как сигнальным и канальным средствам коммуникации для этого необходимы специальные системные вызовы. Разделяемая память представляет собой наиболее быстрый способ взаимодействия процессов в одной вычислительной системе.

3.8. Логическая организация механизма передачи информации

При рассмотрении любого из средств коммуникации нас будет интересовать не их физическая реализация (общая шина данных, прерывания, аппаратно разделяемая память и т.д.), а логическая, определяющая в конечном счете механизм их использования. Некоторые важные аспекты логической реализации являются общими для всех категорий средств связи, некоторые относятся к отдельным категориям. Кратко охарактеризуем основные вопросы, требующие разъяснения при изучении того или иного способа обмена информацией.

Как устанавливается связь?

Могу ли я использовать средство связи непосредственно для обмена информацией сразу после создания процесса или первоначально необходимо предпринять определенные действия для инициализации обмена? Например, для использования общей памяти различными процессами потребуется специальное обращение к операционной системе, которая выделит необходимую область адресного пространства. Но для передачи сигнала от одного процесса к другому никакая инициализация не нужна. В то же время передача информации по линиям связи может потребовать первоначального резервирования такой линии для процессов, желающих обменяться информацией.

К этому же вопросу тесно примыкает вопрос о способе адресации при использовании средства связи. Если я передаю некоторую информацию, я должен указать, куда я ее передаю. Если я желаю получить некоторую информацию, то мне нужно знать, откуда я могу ее получить.

Различают два способа адресации: прямую и непрямую. В случае прямой адресации взаимодействующие процессы непосредственно общаются друг с другом, при каждой операции обмена данными явно указывая имя или номер процесса, которому информация предназначена или от которого она должна быть получена. Если и процесс, от которого данные исходят, и процесс, принимающий данные, указывают имена своих партнеров по взаимодействию, то такая схема адресации называется симметричной прямой адресацией. *Ни один другой процесс не может вмешаться в процедуру симметричного прямого общения двух процессов, перехватить посланные или подменить ожидаемые данные.* Если только один из взаимодействующих процессов, например передающий, указывает имя своего партнера по кооперации, а второй процесс в качестве возможного партнера рассматривает любой процесс в системе, например, ожидает получения информации от произвольного источника, то такая схема адресации называется асимметричной прямой адресацией.

При непрямой адресации данные помещаются передающим процессом в некоторый промежуточный объект для хранения данных, имеющий свой адрес, откуда они могут быть затем изъяты каким-либо другим процессом. Примером такого объекта может служить обычная доска объявлений или рекламная газета. При этом передающий процесс не знает, как именно идентифицируется процесс, который получит информацию, а принимающий процесс не имеет представления об идентификаторе процесса, от которого он должен ее получить.

При использовании прямой адресации связь между процессами в классической операционной системе устанавливается автоматически, без дополнительных инициализирующих действий. Единственное, что нужно для использования средства связи, — это знать, как идентифицируются процессы, участвующие в обмене данными.

При использовании непрямой адресации инициализация средства связи может и не требоваться. Информация, которой должен обладать процесс для взаимодействия с другими процессами, — это некий идентификатор промежуточного объекта для хранения данных, если он, конечно, не является единственным и неповторимым в вычислительной системе для всех процессов.

Информационная валентность процессов и средств связи

Следующий важный вопрос — это вопрос об информационной валентности связи. Слово «валентность» здесь использовано по аналогии с химией. Сколько процессов может быть одновременно ассоциировано с конкретным средством связи? Сколько таких средств связи может быть задействовано между двумя процессами?

Понятно, что при прямой адресации только одно фиксированное средство связи может быть задействовано для обмена данными

между двумя процессами, и только эти два процесса могут быть ассоциированы с ним. При непрямой адресации может существовать более двух процессов, использующих один и тот же объект для данных, и более одного объекта может быть использовано двумя процессами.

К этой же группе вопросов следует отнести и вопрос о направленности связи. Является ли связь однонаправленной или двунаправленной? Под однонаправленной связью мы будем понимать связь, при которой каждый процесс, ассоциированный с ней, может использовать средство связи либо только для приема информации, либо только для ее передачи. При двунаправленной связи каждый процесс, участвующий в общении, может использовать связь и для приема, и для передачи данных. В коммуникационных системах принято называть однонаправленную связь симплексной, двунаправленную связь с поочередной передачей информации в разных направлениях — полудуплексной, а двунаправленную связь с возможностью одновременной передачи информации в разных направлениях — дуплексной. Прямая и непрямая адресация не имеет непосредственного отношения к направленности связи.

Особенности передачи информации с помощью линий связи

Как уже говорилось выше, передача информации между процессами посредством линий связи является достаточно безопасной по сравнению с использованием разделяемой памяти и более информативной по сравнению с сигнальными средствами коммуникации. Кроме того, разделяемая память не может быть использована для связи процессов, функционирующих на различных вычислительных системах. Возможно, именно поэтому каналы связи из средств коммуникации процессов получили наибольшее распространение. Коснемся некоторых вопросов, связанных с логической реализацией канальных средств коммуникации.

Буферизация

Может ли линия связи сохранять информацию, переданную одним процессом, до ее получения другим процессом или помещения в промежуточный объект? Каков объем этой информации? Иными словами, речь идет о том, обладает ли канал связи буфером и каков объем этого буфера. Здесь можно выделить три принципиальных варианта.

1. Буфер нулевой емкости или отсутствует. Никакая информация не может сохраняться на линии связи. В этом случае процесс, посылающий информацию, должен ожидать, пока процесс, принимающий информацию, не соблаговолит ее получить, прежде чем заниматься своими дальнейшими делами (в реальности этот случай никогда не реализуется).

2. Буфер ограниченной емкости. Размер буфера равен n, т.е. линия связи не может хранить до момента получения более чем n единиц информации. Если в момент передачи данных в буфере хватает места, то передающий процесс не должен ничего ожидать. Информация просто копируется в буфер. Если же в момент передачи данных буфер заполнен или места недостаточно, то необходимо задержать работу процесса отправителя до появления в буфере свободного пространства.

3. Буфер неограниченной емкости. Теоретически это возможно, но практически вряд ли реализуемо. Процесс, посылающий информацию, никогда не ждет окончания ее передачи и приема другим процессом.

При использовании канального средства связи с непрямой адресацией под *емкостью буфера* обычно понимается количество информации, которое может быть помещено в промежуточный объект для хранения данных.

Поток ввода/вывода и сообщения

Существуют две модели передачи данных по каналам связи — поток ввода-вывода и сообщения. При передаче данных с помощью потоковой модели операции передачи/приема информации вообще не интересуются содержимым данных. Процесс, прочитавший 100 байт из линии связи, не знает и не может знать, были ли они переданы одновременно, т.е. одним куском или порциями по 20 байт, пришли они от одного процесса или от разных. Данные представляют собой простой поток байтов, без какой-либо их интерпретации со стороны системы. Примерами потоковых каналов связи могут служить *pipe* и *FIFO*, описанные ниже.

Одним из наиболее простых способов передачи информации между процессами по линиям связи является передача данных через *pipe* (канал, трубу или, как его еще называют в литературе, конвейер). Представим себе, что у нас есть некоторая труба в вычислительной системе, в один из концов которой процессы могут «сливать» информацию, а из другого конца принимать полученный поток. Такой способ реализует потоковую модель ввода/вывода. Информацией о расположении трубы в операционной системе обладает только процесс, создавший ее. Этой информацией он может поделиться исключи-

тельно со своими наследниками — процессами-детьми и их потомками. Поэтому использовать *pipe* для связи между собой могут только родственные процессы, имеющие общего предка, создавшего данный канал связи.

Если разрешить процессу, создавшему трубу, сообщать о ее местонахождении в системе другим процессам, сделав вход и выход трубы каким-либо образом видимыми для всех остальных, например, зарегистрировав ее в операционной системе под определенным именем, мы получим объект, который принято называть *FIFO* или *именованный pipe*. Именованный *pipe* может использоваться для организации связи между любыми процессами в системе.

В модели сообщений процессы налагают на передаваемые данные некоторую структуру. Весь поток информации они разделяют на отдельные сообщения, вводя между данными, по крайней мере границы сообщений. Примером границ сообщений являются точки между предложениями в сплошном тексте или границы абзаца. Кроме того, к передаваемой информации могут быть присоединены указания на то, кем конкретное сообщение было послано и для кого оно предназначено. Примером указания отправителя могут служить подписи под эпиграфами в книге. Все сообщения могут иметь одинаковый фиксированный размер или могут быть переменной длины. В вычислительных системах используются разнообразные средства связи для передачи сообщений: очереди сообщений, sockets (гнезда) и т.д.

И потоковые линии связи, и каналы сообщений всегда имеют буфер конечной длины. Когда мы будем говорить о емкости буфера для потоков данных, мы будем измерять ее в байтах. Когда мы будем говорить о емкости буфера для сообщений, мы будем измерять ее в сообщениях.

Надежность средств связи

Одним из существенных вопросов при рассмотрении всех категорий средств связи является вопрос об их надежности. Мы будем называть способ коммуникации надежным, если при обмене данными выполняются четыре условия.

1. Не происходит потери информации.
2. Не происходит повреждения информации.
3. Не появляется лишней информации.
4. Не нарушается порядок данных в процессе обмена.

Очевидно, что передача данных через разделяемую память является надежным способом связи. То, что мы сохранили в разделяемой

памяти, будет считано другими процессами в первозданном виде, если, конечно, не произойдет сбоя в питании компьютера. Для других средств коммуникации это не всегда верно.

Каким образом в вычислительных системах пытаются бороться с ненадежностью коммуникаций? Давайте рассмотрим возможные варианты на примере обмена данными через линию связи с помощью сообщений. Для обнаружения повреждения информации будем снабжать каждое передаваемое сообщение некоторой контрольной суммой, вычисленной по посланной информации. При приеме сообщения контрольную сумму будем вычислять заново и проверять ее соответствие пришедшему значению. Если данные не повреждены (контрольные суммы совпадают), то подтвердим правильность их получения. Если данные повреждены (контрольные суммы не совпадают), то сделаем вид, что сообщение к нам не поступило. Вместо контрольной суммы можно использовать специальное кодирование передаваемых данных с помощью кодов, исправляющих ошибки. Такое кодирование позволяет при числе искажений информации, не превышающем некоторого значения, восстановить первоначальные неискаженные данные. Если по прошествии некоторого интервала времени подтверждение правильности полученной информации не придет на передающий конец линии связи, будем считать информацию утерянной и пошлем ее повторно. Для того чтобы избежать двойного получения одной и той же информации, на приемном конце линии связи должен осуществляться соответствующий контроль. Для гарантии правильного порядка получения сообщений будем их нумеровать. При приеме сообщения с номером, не соответствующим ожидаемому, поступаем с ним как с утерянным и ждем сообщения с правильным номером.

Подобные действия могут быть возложены:

- на операционную систему;
- на процессы, обменивающиеся данными;
- совместно на систему и процессы, разделяя их ответственность.

Операционная система может обнаруживать ошибки при передаче данных и извещать об этом взаимодействующие процессы для принятия ими решения о дальнейшем поведении.

Как завершается связь?

Наконец, важным вопросом при изучении средств обмена данными является вопрос прекращения обмена. Здесь нужно выделить два аспекта: требуются ли от процесса какие-либо специальные дей-

ствия по прекращению использования средства коммуникации и влияет ли такое прекращение на поведение других процессов. Для способов связи, которые не подразумевали никаких инициализирующих действий, обычно ничего специального для окончания взаимодействия предпринимать не надо. Если же установление связи требовало некоторой инициализации, то, как правило, при ее завершении бывает необходимо выполнить ряд операций, например сообщить операционной системе об освобождении выделенного связного ресурса.

Если кооперативные процессы прекращают взаимодействие согласованно, то такое прекращение не влияет на их дальнейшее поведение. Иная картина наблюдается при несогласованном окончании связи одним из процессов. Если какой-либо из взаимодействующих процессов, не завершивших общение, находится в этот момент в состоянии ожидания получения данных либо попадает в такое состояние позже, то операционная система обязана предпринять некоторые действия для того, чтобы исключить вечное блокирование этого процесса. Обычно это либо прекращение работы ожидающего процесса, либо его извещение о том, что связи больше нет (например, с помощью передачи заранее определенного сигнала).

3.9. Нити исполнения

Рассмотренные выше аспекты логической реализации относятся к средствам связи, ориентированным на организацию взаимодействия различных процессов. Однако усилия, направленные на ускорение решения задач в рамках классических операционных систем, привели к появлению совершенно иных механизмов, к изменению самого понятия «процесс».

В свое время внедрение идеи мультипрограммирования позволило повысить пропускную способность компьютерных систем, т.е. уменьшить среднее время ожидания результатов работы процессов. Но любой отдельно взятый процесс в мультипрограммной системе никогда не может быть выполнен быстрее, чем при работе в однопрограммном режиме на том же вычислительном комплексе. Тем не менее, если алгоритм решения задачи обладает определенным внутренним параллелизмом, мы могли бы ускорить его работу, организовав взаимодействие нескольких процессов. Рассмотрим следующий пример. Пусть у нас есть следующая программа на псевдоязыке программирования:

Ввести массив a
Ввести массив b
Ввести массив c
$a = a + b$
$c = a + c$
Вывести массив c

При выполнении такой программы в рамках одного процесса этот процесс четырежды будет блокироваться, ожидая окончания операций ввода-вывода. Но наш алгоритм обладает внутренним параллелизмом. Вычисление суммы массивов $a + b$ можно было бы выполнять параллельно с ожиданием окончания операции ввода массива c.

Ввести массив a
Ожидание окончания операции ввода
Ввести массив b
Ожидание окончания операции ввода
Ввести массив c
Ожидание окончания операции ввода $a = a + b$
$c = a + c$
Вывести массив c
Ожидание окончания операции вывода

Такое совмещение операций по времени можно было бы реализовать, используя два взаимодействующих процесса. Для простоты будем полагать, что средством коммуникации между ними служит разделяемая память. Тогда наши процессы могут выглядеть следующим образом.

Процесс 1	*Процесс* 2
Ввести массив a	*Ожидание ввода массивов a и b*
Ожидание окончания операции ввода	
Ввести массив b	
Ожидание окончания операции ввода	
Ввести массив c	
Ожидание окончания операции ввода $a = a + b$	
$c = a + c$	
Вывести массив c	
Ожидание окончания операции вывода	

Казалось бы, мы предложили конкретный способ ускорения решения задачи. Однако в действительности дело обстоит не так просто. Второй процесс должен быть создан, оба процесса должны сообщить операционной системе, что им необходима память, кото-

рую они могли бы разделить с другим процессом, и, наконец, нельзя забывать о переключении контекста. Поэтому реальное поведение процессов будет выглядеть примерно так.

Процесс 1 — *Процесс* 2

Создать процесс 2

Переключение контекста

Выделение общей памяти

Ожидание ввода a и b

Переключение контекста

Выделение общей памяти

Ввести массив a

Ожидание окончания операции ввода

Ввести массив b

Ожидание окончания операции ввода

Ввести массив c

Ожидание окончания операции ввода

Переключение контекста

$a = a + b$

Переключение контекста

$c = a + c$

Вывести массив c

Ожидание окончания операции вывода

Очевидно, что мы можем не только не выиграть во времени при решении задачи, но даже и проиграть, так как временные потери на создание процесса, выделение общей памяти и переключение контекста могут превысить выигрыш, полученный за счет совмещения операций.

Для того чтобы реализовать нашу идею, введем новую абстракцию внутри понятия «процесс» — *нить исполнения* или просто *нить* (в англоязычной литературе используется термин *thread*). Нити процесса разделяют его программный код, глобальные переменные и системные ресурсы, но каждая нить имеет собственный программный счетчик, свое содержимое регистров и свой стек. Теперь процесс представляется как совокупность взаимодействующих нитей и выделенных ему ресурсов. Процесс, содержащий всего одну нить исполнения, идентичен процессу в том смысле, который мы употребляли ранее. Для таких процессов мы в дальнейшем будем использовать термин *традиционный процесс*. Иногда нити называют облегченными процессами или мини-процессами, так как во многих отношениях они подобны традиционным процессам. Нити, как и процессы, могут порождать нити-потомки, правда, только внутри своего процесса, и переходить из одного состояния в другое. Состояния нитей аналогичны состояниям традиционных процессов. Из состояния «рождение» процесс приходит содержащим всего одну нить исполнения. Другие нити процесса будут являться потомками этой нити-прародительницы. Мы можем считать, что процесс находится в состоянии «готовность», если хотя бы одна из его нитей находится в состоянии «готовность» и ни одна из нитей не находится в состоянии «исполнение». Мы можем считать, что процесс находится в состоянии «исполнение», если одна из его нитей находится в состоянии «исполнение». Процесс будет находиться в состоянии «ожидание», если все его нити находятся в состоянии «ожидание». Наконец, процесс находится в состоянии «закончил исполнение», если все его нити находятся в состоянии «закончила исполнение». Пока одна нить процесса заблокирована, другая нить того же процесса может выполняться. Нити разделяют процессор так же, как это делали традиционные процессы, в соответствии с рассмотренными алгоритмами планирования.

Поскольку нити одного процесса разделяют существенно больше ресурсов, чем различные процессы, то операции создания новой нити и переключения контекста между нитями одного процесса занимают значительно меньше времени, чем аналогичные операции для процессов в целом. Предложенная нами схема совмещения работы в терминах нитей одного процесса получает право на существование.

Нить 1 — *Нить 2*

Создать нить 2

Переключение контекста нитей

Ожидание ввода a и b

Переключение контекста нитей

Ввести массив a

Ожидание окончания операции ввода

Ввести массив b

Ожидание окончания операции ввода

Ввести массив c

Ожидание окончания операции ввода

Переключение контекста нитей

$a = a + b$

Переключение контекста нитей

$c = a + c$

Вывести массив c

Ожидание окончания операции вывода

Различают операционные системы, поддерживающие нити на уровне ядра и на уровне библиотек. Все сказанное выше справедливо для операционных систем, поддерживающих нити на уровне ядра. В них планирование использования процессора происходит в терминах нитей, а управление памятью и другими системными ресурсами остается в терминах процессов. В операционных системах, поддерживающих нити на уровне библиотек пользователей, и планирование процессора, и управление системными ресурсами осуществляются в терминах процессов. Распределение использования процессора по нитям в рамках выделенного процессу временного интервала осуществляется средствами библиотеки. В подобных системах блокирование одной нити приводит к блокированию всего процесса, ибо ядро операционной системы не имеет представления о существовании нитей. По сути дела, в таких вычислительных системах просто имитируется наличие нитей исполнения.

Глава 4
КРИТИЧЕСКИЕ СЕКЦИИ ПРОЦЕССОВ, ВЗАИМОИСКЛЮЧЕНИЯ И ОРГАНИЗАЦИЯ ПРАВИЛЬНОЙ ОЧЕРЕДНОСТИ

4.1. Interleaving, race condition и взаимоисключения

Давайте временно отвлечемся от операционных систем, процессов и нитей исполнения и поговорим о некоторых «активностях». Под активностями мы будем понимать последовательное выполнение ряда действий, направленных на достижение определенной цели. Активности могут иметь место в программном и техническом обеспечении, в обычной деятельности людей и животных. Мы будем разбивать активности на некоторые неделимые, или атомарные, операции. Например, активность «приготовление бутерброда» можно разбить на следующие атомарные операции:

1. Отрезать ломтик хлеба.
2. Отрезать ломтик колбасы.
3. Намазать ломтик хлеба маслом.
4. Положить ломтик колбасы на подготовленный ломтик хлеба.

Неделимые операции могут иметь внутренние невидимые действия (взять батон хлеба в левую руку, взять нож в правую руку, произвести отрезание). Мы же называем их неделимыми потому, что считаем выполняемыми за раз, без прерывания деятельности.

Пусть имеется две активности

$$P: a\ b\ c$$

$$Q: d\ e\ f$$

где a, b, c, d, e, f — атомарные операции. При последовательном выполнении активностей мы получаем такую последовательность атомарных действий:

$$PQ: a\ b\ c\ d\ e\ f$$

Что произойдет при исполнении этих активностей псевдопараллельно, в режиме разделения времени? Активности могут расслоиться на неделимые операции с различным чередованием, т.е. может

произойти то, что на английском языке принято называть словом «interleaving». Возможные варианты чередования:

a b c d e f
a b d c e f
a b d e c f
a b d e f c
a d b c e f
.....
d e f a b c

Атомарные операции активностей могут чередоваться всевозможными различными способами с сохранением порядка расположения внутри активностей. Так как псевдопараллельное выполнение двух активностей приводит к чередованию их неделимых операций, результат псевдопараллельного выполнения может отличаться от результата последовательного выполнения.

Рассмотрим пример. Пусть у нас имеются две активности P и Q, состоящие из двух атомарных операций каждая:

$$P: x = 2 \quad Q: x = 3$$
$$y = x - 1 \quad y = x + 1$$

Что мы получим в результате их псевдопараллельного выполнения, если переменные x и y являются для активностей общими? Очевидно, что возможны четыре разных набора значений для пары (x, y): (3, 4), (2, 1), (2, 3) и (3, 2). Мы будем говорить, что набор активностей (например, программ) детерминирован, если всякий раз при псевдопараллельном исполнении для одного и того же набора входных данных он дает одинаковые выходные данные. В противном случае он недетерминирован. Выше приведен пример недетерминированного набора программ. Понятно, что детерминированный набор активностей можно безбоязненно выполнять в режиме разделения времени. Для недетерминированного набора такое исполнение нежелательно.

Можно ли до получения результатов определить, является ли набор активностей детерминированным или нет? Для этого существуют достаточные условия Бернстайна. Изложим их применительно к программам с разделяемыми переменными.

Введем наборы входных и выходных переменных программы. Для каждой атомарной операции наборы входных и выходных переменных — это наборы переменных, которые атомарная операция считывает и записывает. Набор входных переменных программы $R(P)$

(R от слова «read») суть объединение наборов входных переменных для всех ее неделимых действий. Аналогично, набор выходных переменных программы $W(P)$ (W от слова «write») суть объединение наборов выходных переменных для всех ее неделимых действий. Например, для программы

$$P: x = u + v$$
$$y = x \cdot w$$

получаем $R(P) = \{u, v, x, w\}$, $W(P) = \{x, y\}$. Заметим, что переменная x присутствует как в $R(P)$, так и в $W(P)$.

Теперь сформулируем условия Бернстайна.

Если для двух данных активностей P и Q:

- пересечение $W(P)$ и $W(Q)$ пусто,
- пересечение $W(P)$ с $R(Q)$ пусто,
- пересечение $R(P)$ и $W(Q)$ пусто,

тогда выполнение P и Q детерминировано.

Если эти условия не соблюдены, возможно, параллельное выполнение P и Q детерминировано, а может быть, и нет.

Случай двух активностей естественным образом обобщается на их большее количество.

Условия Бернстайна информативны, но слишком жестки. По сути дела, они требуют практически невзаимодействующих процессов. А нам хотелось бы, чтобы детерминированный набор образовывали активности, совместно использующие информацию и обменивающиеся ею. Для этого нам необходимо ограничить число возможных чередований атомарных операций, исключив некоторые чередования с помощью механизмов синхронизации выполнения программ, обеспечив тем самым упорядоченный доступ программ к некоторым данным.

Про недетерминированный набор программ (и активностей вообще) говорят, что он имеет *race condition* (состояние гонки, состояние состязания). В приведенном выше примере процессы состязаются за вычисление значений переменных x и y.

Задачу упорядоченного доступа к разделяемым данным (устранение race condition) в том случае, когда нам не важна его очередность, можно решить, если обеспечить каждому процессу эксклюзивное право доступа к этим данным. Каждый процесс, обращающийся к разделяемым ресурсам, исключает для всех других процессов возможность одновременного общения с этими ресурсами, если это может привести к недетерминированному поведению набора процессов. Такой прием называется взаимоисключением

(mutual exclusion). Если очередность доступа к разделяемым ресурсам важна для получения правильных результатов, то одними взаимоисключениями уже не обойтись, нужна взаимосинхронизация поведения программ.

4.2. Критическая секция

Важным понятием при изучении способов синхронизации процессов является понятие критической секции (critical section) программы. Критическая секция — это часть программы, исполнение которой может привести к возникновению race condition для определенного набора программ. Чтобы исключить эффект гонок по отношению к некоторому ресурсу, необходимо организовать работу так, чтобы в каждый момент времени только один процесс мог находиться в своей критической секции, связанной с этим ресурсом. Иными словами, необходимо обеспечить реализацию взаимоисключения для критических секций программ. Реализация взаимоисключения для критических секций программ с практической точки зрения означает, что по отношению к другим процессам, участвующим во взаимодействии, критическая секция начинает выполняться как атомарная операция. Давайте рассмотрим следующий пример, в котором псевдопараллельные взаимодействующие процессы представлены действиями различных студентов (табл. 4.1).

Таблица 4.1

Пример взаимодействующих процессов

Время	Студент 1	Студент 2	Студент 3
17:05	Приходит в комнату		
17:07	Обнаруживает, что хлеба нет		
17:09	Уходит в магазин		
17:11		Приходит в комнату	
17:13		Обнаруживает, что хлеба нет	
17:15		Уходит в магазин	
17:17			Приходит в комнату
17:19			Обнаруживает, что хлеба нет
17:21			Уходит в магазин
17:23	Приходит в магазин		
17:25	Покупает 2 батона на всех		
17:27	Уходит из магазина		
17:29		Приходит в магазин	
17:31		Покупает 2 батона на всех	
17:33		Уходит из магазина	
17:35			Приходит в магазин
17:37			Покупает 2 батона на всех
17:39			Уходит из магазина
17:41	Возвращается в комнату		
17:43			
17:45			
17:47		Возвращается в комнату	
17:49			
17:51			
17:53			Возвращается в комнату

Окончание табл. 4.1

Здесь критический участок для каждого процесса — от операции «Обнаруживает, что хлеба нет» до операции «Возвращается в комнату» включительно. В результате отсутствия взаимоисключения мы из ситуации «Нет хлеба» попадаем в ситуацию «Слишком много хлеба». Если бы этот критический участок выполнялся как атомарная операция «Достает два батона хлеба», то проблема образования излишков была бы снята.

Сделать процесс добывания хлеба атомарной операцией можно было бы следующим образом: перед началом этого процесса закрыть дверь изнутри на засов и уходить добывать хлеб через окно, а по окончании процесса вернуться в комнату через окно и отодвинуть засов. Тогда пока один студент добывает хлеб, все остальные находятся в состоянии ожидания под дверью (табл. 4.2).

Таблица 4.2

Взаимодействие с закрытой дверью

Время	Студент 1	Студент 2	Студент 3
17:05	Приходит в комнату		
17:07	Достает 2 батона хлеба		
17:43		Приходит в комнату	
17:47			Приходит в комнату

Итак, для решения задачи необходимо, чтобы в том случае, когда процесс находится в своем критическом участке, другие процессы не могли войти в свои критические участки. Мы видим, что критический участок должен сопровождаться прологом (*entry section*) — «закрыть дверь изнутри на засов» — и эпилогом (*exit section*) — «отодвинуть засов», которые не имеют отношения к активности одиночного процесса. Во время выполнения пролога процесс должен, в частности, получить разрешение на вход в критический участок, а во время выполнения эпилога — сообщить другим процессам, что он покинул критическую секцию.

В общем случае структура процесса, участвующего во взаимодействии, может быть представлена следующим образом:

while (*some condition*) {
entry section
critical section
exit section
remainder section
}

Здесь под *remainder section* понимаются все атомарные операции, не входящие в критическую секцию.

Глава 5
АЛГОРИТМЫ СИНХРОНИЗАЦИИ ПРОЦЕССОВ

5.1. Требования, предъявляемые к алгоритмам

Организация взаимоисключения для критических участков, конечно, позволит избежать возникновения *race condition*, но не является достаточной для правильной и эффективной параллельной работы кооперативных процессов. Сформулируем пять условий, которые должны выполняться для хорошего программного алгоритма организации взаимодействия процессов, имеющих критические участки, если они могут проходить их в произвольном порядке.

1. Задача должна быть решена чисто программным способом на обычной машине, не имеющей специальных команд взаимоисключения. При этом предполагается, что основные инструкции языка программирования (такие примитивные инструкции, как *load, store, test*) являются атомарными операциями.

2. Не должно существовать никаких предположений об относительных скоростях выполняющихся процессов или числе процессоров, на которых они исполняются.

3. Если процесс P_i исполняется в своем критическом участке, то не существует никаких других процессов, которые исполняются в соответствующих критических секциях. Это условие получило название условия взаимоисключения (*mutual exclusion*).

4. Процессы, которые находятся вне своих критических участков и не собираются входить в них, не могут препятствовать другим процессам входить в их собственные критические участки. Если нет процессов в критических секциях и имеются процессы, желающие войти в них, то только те процессы, которые не исполняются в *remainder section*, должны принимать решение о том, какой процесс войдет в свою критическую секцию. Такое решение не должно приниматься бесконечно долго. Это условие получило название условия прогресса (*progress*).

5. Не должно возникать неограниченно долгого ожидания для входа одного из процессов в свой критический участок. От того момента, когда процесс запросил разрешение на вход в критическую секцию, и до того момента, когда он это разрешение получил, другие

процессы могут пройти через свои критические участки лишь ограниченное число раз. Это условие получило название условия ограниченного ожидания (*bound waiting*).

Надо заметить, что описание соответствующего алгоритма в нашем случае означает описание способа организации пролога и эпилога для критической секции.

5.2. Запрет прерываний

Наиболее простым решением поставленной задачи является следующая организация пролога и эпилога:

```
while (some condition) {
запретить все прерывания
critical section
разрешить все прерывания
remainder section
}
```

Поскольку выход процесса из состояния **исполнение** без его завершения осуществляется по прерыванию, внутри критической секции никто не может вмешаться в его работу. Однако такое решение может иметь далеко идущие последствия, поскольку позволяет процессу пользователя разрешать и запрещать прерывания во всей вычислительной системе. Допустим, что пользователь случайно или по злому умыслу запретил прерывания в системе и зациклил или завершил свой процесс. Без перезагрузки системы в такой ситуации не обойтись.

Тем не менее запрет и разрешение прерываний часто применяются как пролог и эпилог к критическим секциям внутри самой операционной системы, например при обновлении содержимого PCB.

5.3. Переменная-замок

В качестве следующей попытки решения задачи для пользовательских процессов рассмотрим другое предложение. Возьмем некоторую переменную, доступную всем процессам, с начальным значением равным 0. Процесс может войти в критическую секцию только тогда, когда значение этой переменной-замка равно 0, одновременно изменяя ее значение на 1 — закрывая замок. При выходе из критической секции процесс сбрасывает ее значение в 0 — замок открывается (как в случае с покупкой хлеба студентами в разделе «Критическая секция»).

```
shared int lock = 0;
/* shared означает, что */
/* переменная является разделяемой */
while (some condition) {
while(lock); lock = 1;
critical section
lock = 0;
remainder section
}
```

К сожалению, при внимательном рассмотрении мы видим, что такое решение не удовлетворяет условию взаимоисключения, так как действие *while*(*lock*); *lock* = 1; не является атомарным. Допустим, процесс *P*0 протестировал значение переменной *lock* и принял решение двигаться дальше. В этот момент, еще до присвоения переменной *lock* значения 1, планировщик передал управление процессу *P*1. Он тоже изучает содержимое переменной *lock* и тоже принимает решение войти в критический участок. Мы получаем два процесса, одновременно выполняющих свои критические секции.

5.4. Строгое чередование

Попробуем решить задачу сначала для двух процессов. Очередной подход будет также использовать общую для них обоих переменную с начальным значением 0. Только теперь она будет играть не роль замка для критического участка, а явно указывать, кто может следующим войти в него. Для *i*-го процесса это выглядит так:

```
shared int turn = 0;
while (some condition) {
while(turn != i);
critical section
turn = 1-i;
remainder section
}
```

Очевидно, что взаимоисключение гарантируется, процессы входят в критическую секцию строго по очереди: *P*0, *P*1, *P*0, *P*1, *P*0, ... Но наш алгоритм не удовлетворяет условию прогресса. Например, если значение *turn* равно 1 и процесс *P*0 готов войти в критический

участок, он не может сделать этого, даже если процесс *P*1 находится в *remainder section*.

5.5. Флаги готовности

Недостаток предыдущего алгоритма заключается в том, что процессы ничего не знают о состоянии друг друга в текущий момент времени. Давайте попробуем исправить эту ситуацию. Пусть два наших процесса имеют разделяемый массив флагов готовности входа процессов в критический участок

shared int ready[2] = {0, 0};

Когда *i*-й процесс готов войти в критическую секцию, он присваивает элементу массива *ready[i]* значение, равное 1. После выхода из критической секции он, естественно, сбрасывает это значение в 0. Процесс не входит в критическую секцию, если другой процесс уже готов к входу в критическую секцию или находится в ней.

```
while (some condition) {
ready[i] = 1;
while(ready[1-i]);
critical section
ready[i] = 0;
remainder section
}
```

Полученный алгоритм обеспечивает взаимоисключение, позволяет процессу, готовому к входу в критический участок, войти в него сразу после завершения эпилога в другом процессе, но все равно нарушает условие прогресса. Пусть процессы практически одновременно подошли к выполнению пролога. После выполнения присваивания *ready[0]* = 1 планировщик передал процессор от процесса 0 процессу 1, который также выполнил присваивание *ready[1] = 1*. После этого оба процесса бесконечно долго ждут друг друга на входе в критическую секцию. Возникает ситуация, которую принято называть тупиковой (*deadlock*).

5.6. Алгоритм Петерсона

Первое решение проблемы, удовлетворяющее всем требованиям и использующее идеи ранее рассмотренных алгоритмов, было предложено датским математиком Деккером (Dekker). В 1981 г. Петерсон (Peterson) предложил более изящное решение. Пусть оба процесса имеют доступ к массиву флагов готовности и к переменной очередности.

```
shared int ready[2] = {0, 0};
shared int turn;
while (some condition) {
ready[i] = 1;
turn = 1-i;
while(ready[1-i] && turn = = 1-i);
critical section
ready[i] = 0;
remainder section
}
```

При исполнении пролога критической секции процесс *Pi* заявляет о своей готовности выполнить критический участок и одновременно предлагает другому процессу приступить к его выполнению. Если оба процесса подошли к прологу практически одновременно, то они оба объявят о своей готовности и предложат выполняться друг другу. При этом одно из предложений всегда следует после другого. Тем самым работу в критическом участке продолжит процесс, которому было сделано последнее предложение.

Давайте докажем, что все пять наших требований к алгоритму действительно удовлетворяются.

Удовлетворение требований 1 и 2 очевидно.

Докажем выполнение условия взаимоисключения методом от противного. Пусть оба процесса одновременно оказались внутри своих критических секций. Заметим, что процесс *Pi* может войти в критическую секцию, только если *ready[1-i]* = = 0 или *turn* = = *i*. Заметим также, что если оба процесса выполняют свои критические секции одновременно, то значения флагов готовности для обоих процессов совпадают и равны 1. Могли ли оба процесса войти в критические секции из состояния, когда они оба одновременно находились в процессе выполнения цикла *while*? Нет, так как в этом случае переменная *turn* должна была бы одновременно иметь значения 0 и 1 (когда оба процесса выполняют цикл, значения переменных изменяться не могут). Пусть процесс *P*0 первым вошел в критический участок, тогда процесс *P*1 должен был выполнить перед вхождением в цикл *while* по крайней мере один предваряющий оператор (*turn = 0;*).

участок, он не может сделать этого, даже если процесс *P*1 находится в *remainder section*.

5.5. Флаги готовности

Недостаток предыдущего алгоритма заключается в том, что процессы ничего не знают о состоянии друг друга в текущий момент времени. Давайте попробуем исправить эту ситуацию. Пусть два наших процесса имеют разделяемый массив флагов готовности входа процессов в критический участок

```
shared int ready[2] = {0, 0};
```

Когда *i*-й процесс готов войти в критическую секцию, он присваивает элементу массива *ready[i]* значение, равное 1. После выхода из критической секции он, естественно, сбрасывает это значение в 0. Процесс не входит в критическую секцию, если другой процесс уже готов к входу в критическую секцию или находится в ней.

```
while (some condition) {
ready[i] = 1;
while(ready[1-i]);
critical section
ready[i] = 0;
remainder section
}
```

Полученный алгоритм обеспечивает взаимоисключение, позволяет процессу, готовому к входу в критический участок, войти в него сразу после завершения эпилога в другом процессе, но все равно нарушает условие прогресса. Пусть процессы практически одновременно подошли к выполнению пролога. После выполнения присваивания *ready[0]* = 1 планировщик передал процессор от процесса 0 процессу 1, который также выполнил присваивание *ready[1] = 1*. После этого оба процесса бесконечно долго ждут друг друга на входе в критическую секцию. Возникает ситуация, которую принято называть тупиковой (*deadlock*).

5.6. Алгоритм Петерсона

Первое решение проблемы, удовлетворяющее всем требованиям и использующее идеи ранее рассмотренных алгоритмов, было предложено датским математиком Деккером (Dekker). В 1981 г. Петерсон (Peterson) предложил более изящное решение. Пусть оба процесса имеют доступ к массиву флагов готовности и к переменной очередности.

```
shared int ready[2] = {0, 0};
shared int turn;
while (some condition) {
ready[i] = 1;
turn = 1-i;
while(ready[1-i] && turn = = 1-i);
critical section
ready[i] = 0;
remainder section
}
```

При исполнении пролога критической секции процесс *Pi* заявляет о своей готовности выполнить критический участок и одновременно предлагает другому процессу приступить к его выполнению. Если оба процесса подошли к прологу практически одновременно, то они оба объявят о своей готовности и предложат выполняться друг другу. При этом одно из предложений всегда следует после другого. Тем самым работу в критическом участке продолжит процесс, которому было сделано последнее предложение.

Давайте докажем, что все пять наших требований к алгоритму действительно удовлетворяются.

Удовлетворение требований 1 и 2 очевидно.

Докажем выполнение условия взаимоисключения методом от противного. Пусть оба процесса одновременно оказались внутри своих критических секций. Заметим, что процесс *Pi* может войти в критическую секцию, только если *ready[1-i]* = = 0 или *turn* = = *i*. Заметим также, что если оба процесса выполняют свои критические секции одновременно, то значения флагов готовности для обоих процессов совпадают и равны 1. Могли ли оба процесса войти в критические секции из состояния, когда они оба одновременно находились в процессе выполнения цикла *while*? Нет, так как в этом случае переменная *turn* должна была бы одновременно иметь значения 0 и 1 (когда оба процесса выполняют цикл, значения переменных изменяться не могут). Пусть процесс *P*0 первым вошел в критический участок, тогда процесс *P*1 должен был выполнить перед вхождением в цикл *while* по крайней мере один предваряющий оператор (*turn = 0;*).

Однако после этого он не может выйти из цикла до окончания критического участка процесса *P*0, так как при входе в цикл *ready[0]*= = 1 и *turn* = = *0*, и эти значения не могут измениться до тех пор, пока процесс *P*0 не покинет свой критический участок. Мы пришли к противоречию. Следовательно, имеет место взаимоисключение.

Докажем выполнение условия прогресса. Возьмем, без ограничения общности, процесс *P*0. Заметим, что он не может войти в свою критическую секцию только при совместном выполнении условий *ready[1]* = = *1* и *turn* = = *1*. Если процесс *P*1 не готов к выполнению критического участка, то *ready[1]* = = 0 и процесс *P*0 может осуществить вход. Если процесс *P*1 готов к выполнению критического участка, то *ready[1]* = = *1* и переменная *turn* имеет значение 0 либо 1, позволяя процессу *P*0 либо процессу *P*1 начать выполнение критической секции. Если процесс *P*1 завершил выполнение критического участка, то он сбросит свой флаг готовности *ready[1]* = = 0, разрешая процессу *P*0 приступить к выполнению критической работы. Таким образом, условие прогресса выполняется.

Отсюда же вытекает выполнение условия ограниченного ожидания. Так как в процессе ожидания разрешения на вход процесс *P*0 не изменяет значения переменных, он сможет начать исполнение своего критического участка после не более чем одного прохода по критической секции процесса *P*1.

5.7. Алгоритм булочной (Bakery algorithm)

Алгоритм Петерсона дает нам решение задачи корректной организации взаимодействия двух процессов.

Давайте рассмотрим теперь соответствующий алгоритм для *n* взаимодействующих процессов, который получил название алгоритма булочной, хотя применительно к нашим условиям его следовало бы скорее назвать алгоритмом регистратуры в поликлинике. Основная его идея выглядит так. Каждый вновь прибывающий клиент (он же процесс) получает талончик на обслуживание с номером. Клиент с наименьшим номером на талончике обслуживается следующим. К сожалению, из-за неатомарности операции вычисления следующего номера алгоритм булочной не гарантирует, что у всех процессов будут талончики с разными номерами. В случае равенства номеров на талончиках у двух или более клиентов первым обслуживается клиент с меньшим значением имени (имена можно сравнивать в лексикографическом порядке). Разделяемые структуры данных для алгоритма — это два массива

shared enum {false, true} choosing[n];
shared int number[n];

Изначально элементы этих массивов инициируются значениями *false* и 0 соответственно. Введем следующие обозначения

(*a*,*b*) < (*c*,*d*), если *a* < *c*
или если *a* = = *c* и *b* < *d*
max(*a*0, *a*1, ..., *an*) — это число *k* такое, что
k > = *ai* для всех *i* = 0, ...,*n*
Структура процесса *Pi* для алгоритма булочной приведена ниже:

```
while (some condition) {
choosing[i] = true;
number[i] = max(number[0], ...,
number[n-1]) + 1;
choosing[i] =false;
for(j = 0; j < n; j + + ){
while(choosing[j]);
while(number[j] ! = 0 && (number[j],j) <
(number[i],i));
}
critical section
number[i] = 0;
remainder section
}
```

5.8. Аппаратная поддержка взаимоисключений

Наличие аппаратной поддержки взаимоисключений позволяет упростить алгоритмы и повысить их эффективность точно так же, как это происходит и в других областях программирования. Мы уже обращались к общепринятому *hardware* для решения задачи реализации взаимоисключений, когда говорили об использовании механизма запрета/разрешения прерываний.

Многие вычислительные системы помимо этого имеют специальные команды процессора, которые позволяют проверить и изменить значение машинного слова или поменять местами значения двух машинных слов в памяти, выполняя эти действия как атомарные операции. Давайте обсудим, как концепции таких команд могут использоваться для реализации взаимоисключений.

5.9. Команда *Test-and-Set* (проверить и присвоить 1)

О выполнении команды *Test-and-Set*, осуществляющей проверку значения логической переменной с одновременной установкой ее значения в 1, можно думать как о выполнении функции

```
int Test_and_Set (int *target){
int tmp = *target;
*target = 1;
return tmp;
}
```

С использованием этой атомарной команды мы можем модифицировать наш алгоритм для переменной-замка, так чтобы он обеспечивал взаимоисключения.

```
shared int lock = 0;
while (some condition) {
while(Test_and_Set(&lock));
critical section
lock = 0;
remainder section
}
```

К сожалению, даже в таком виде полученный алгоритм не удовлетворяет условию ограниченного ожидания для алгоритмов.

5.10. Команда *Swap* (обменять значения)

Выполнение команды *Swap*, обменивающей два значения, находящихся в памяти, можно проиллюстрировать следующей функцией:

```
void Swap (int *a, int *b){
int tmp = *a;
*a = *b;
*b = tmp;
}
```

Применяя атомарную команду *Swap*, мы можем реализовать предыдущий алгоритм, введя дополнительную логическую переменную *key*, локальную для каждого процесса:

```
shared int lock = 0;
int key;
while (some condition) {
key = 1;
do Swap(&lock,&key);
while (key);
critical section
lock = 0;
remainder section
}
```

Глава 6
СЕМАФОРЫ, МОНИТОРЫ, СООБЩЕНИЯ И ИХ ЭКВИВАЛЕНТНОСТЬ

Для повышения производительности вычислительных систем и облегчения задачи программистов существуют специальные механизмы синхронизации. К ним относятся семафоры Дейкстры, мониторы Хора, очередей сообщений и др.

Алгоритмы синхронизации процессов хотя и являются корректными, но достаточно громоздки и не обладают элегантностью. Более того, процедура ожидания входа в критический участок предполагает достаточно длительное вращение процесса в пустом цикле, т.е. напрасную трату времени процессора. Существуют и другие серьезные недостатки у алгоритмов, построенных средствами обычных языков программирования. Допустим, что в вычислительной системе находятся два взаимодействующих процесса: один из них — *H* — с высоким приоритетом, другой — *L* — с низким приоритетом. Пусть планировщик устроен так, что процесс с высоким приоритетом вытесняет низкоприоритетный процесс всякий раз, когда он готов к исполнению, и занимает процессор на все время своего CPU burst (если не появится процесс с еще большим приоритетом). Тогда в случае, если процесс *L* находится в своей критической секции, а процесс *H*, получив процессор, подошел ко входу в критическую область, мы получаем тупиковую ситуацию. Процесс *H* не может войти в критическую область, находясь в цикле, а процесс *L* не получает управления, чтобы покинуть критический участок.

Для того чтобы не допустить возникновения подобных проблем, были разработаны различные механизмы синхронизации более высокого уровня.

6.1. Семафоры

Одним из первых механизмов, предложенных для синхронизации поведения процессов, стали семафоры, концепцию которых описал Дейкстра (Dijkstra) в 1965 г.

Концепция семафоров

Семафор представляет собой целую переменную, принимающую неотрицательные значения, доступ любого процесса к которой, за исключением момента ее инициализации, может осуществляться

только через две атомарные операции: *P* (от датского слова «proberen» — проверять) и *V* (от «verhogen» — увеличивать).

Классическое определение этих операций выглядит следующим образом:

P(S): пока S = = 0 процесс блокируется;
S = S — 1;
V(S): S = S + 1;

Эта запись означает следующее: при выполнении операции *P* над семафором *S* сначала проверяется его значение. Если оно больше 0, то из *S* вычитается 1. Если оно меньше или равно 0, то процесс блокируется до тех пор, пока *S* не станет больше 0, после чего из *S* вычитается 1. При выполнении операции *V* над семафором *S* к его значению просто прибавляется 1. В момент создания семафор может быть инициализирован любым неотрицательным значением.

Подобные переменные-семафоры могут с успехом применяться для решения различных задач организации взаимодействия процессов. В ряде языков программирования они были непосредственно введены в синтаксис языка (например, в ALGOL-68), в других случаях реализуются с помощью специальных системных вызовов. Соответствующая целая переменная располагается внутри адресного пространства ядра операционной системы. Операционная система обеспечивает атомарность операций *P* и *V*, используя, например, метод запрета прерываний на время выполнения соответствующих системных вызовов. Если при выполнении операции *P* заблокированными оказались несколько процессов, то порядок их разблокирования может быть произвольным, например *FIFO*.

Решение проблемы producer-consumer с помощью семафоров

Одной из типовых задач, требующих организации взаимодействия процессов, является задача producer-consumer (производитель-потребитель). Пусть два процесса обмениваются информацией через буфер ограниченного размера. Производитель закладывает информацию в буфер, а потребитель извлекает ее оттуда. На этом уровне деятельность потребителя и производителя можно описать следующим образом:

```
Producer: while(1) {
produce_item;
put_item;
}
```

```
Consumer: while(1) {
get_item;
consume_item;
}
```

Если буфер заполнен, то производитель должен ждать, пока в нем появится место, чтобы положить туда новую порцию информации. Если буфер пуст, то потребитель должен дожидаться нового сообщения.

Как можно реализовать эти условия с помощью семафоров? Возьмем три семафора: *empty*, *full* и *mutex*.

Семафор *full* будем использовать для гарантии того, что потребитель будет ждать, пока в буфере появится информация. Семафор *empty* будем использовать для организации ожидания производителя при заполненном буфере, а семафор *mutex* — для организации взаимоисключения на критических участках, которыми являются действия *put_item* и *get_item* (операции «положить информацию» и «взять информацию» не могут пересекаться, так как в этом случае возникнет опасность искажения информации). Тогда решение задачи на *C*-подобном языке выглядит так:

```
Semaphore mutex = 1;
Semaphore empty = N; /* где N — емкость буфера*/
Semaphore full = 0;
Producer:
while(1) {
produce_item;
P(empty);
P(mutex);
put_item;
V(mutex);
V(full);
}
Consumer:
while(1) {
P(full);
P(mutex);
get_item;
V(mutex);
V(empty);
consume_item;
}
```


Легко убедиться, что это действительно корректное решение поставленной задачи. Попутно заметим, что семафоры использовались здесь для достижения двух целей: организации взаимоисключения на критическом участке и взаимосинхронизации скорости работы процессов.

6.2. Мониторы

Хотя решение задачи producer-consumer с помощью семафоров выглядит достаточно изящно, программирование с их использованием требует повышенной осторожности и внимания, чем отчасти напоминает программирование на языке Ассемблера. Допустим, что в рассмотренном примере мы случайно поменяли местами операции *P*, сначала выполнив операцию для семафора *mutex*, а уже затем для семафоров *full* и *empty*. Допустим теперь, что потребитель, войдя в свой критический участок (*mutex* сброшен), обнаруживает, что буфер пуст. Он блокируется и начинает ждать появления сообщений. Но производитель не может войти в критический участок для передачи информации, так как тот заблокирован потребителем.

Получаем тупиковую ситуацию.

В сложных программах произвести анализ правильности использования семафоров с карандашом в руках становится очень непросто. В то же время обычные способы отладки программ зачастую не дают результата, поскольку возникновение ошибок зависит от *interleaving* атомарных операций, и ошибки могут быть трудновоспроизводимы. Для того чтобы облегчить работу программистов, в 1974 г. Хором (Hoare) был предложен механизм еще более высокого уровня, чем семафоры, получивший название мониторов. Рассмотрим конструкцию, несколько отличающуюся от оригинальной.

Мониторы представляют собой тип данных, который может быть с успехом внедрен в объектно-ориентированные языки программирования. Монитор обладает собственными переменными, определяющими его состояние. Значения этих переменных извне могут быть изменены только с помощью вызова функций-методов, принадлежащих монитору. В свою очередь, эти функции-методы могут использовать в работе только данные, находящиеся внутри монитора, и свои параметры. На абстрактном уровне можно описать структуру монитора следующим образом:

```
monitor monitor_name {
описание внутренних переменных ;
```

```
void m1(...){...
}
void m2(...){...
}
...
void mn(...){...
}
{
блок инициализации внутренних переменных;
}
}
```

Здесь функции *m*1, ..., *mn* представляют собой функции-методы монитора, а *блок инициализации внутренних переменных* содержит операции, которые выполняются один и только один раз: при создании монитора или при самом первом вызове какой-либо функции-метода до ее исполнения.

Важной особенностью мониторов является то, что в любой момент времени только один процесс может быть активен, т.е. находиться в состоянии **готовность** или **исполнение**, внутри данного монитора. Поскольку мониторы представляют собой особые конструкции языка программирования, компилятор может отличить вызов функции, принадлежащей монитору, от вызовов других функций и обработать его специальным образом, добавив к нему пролог и эпилог, реализующий взаимоисключение. Так как обязанность конструирования механизма взаимоисключений возложена на компилятор, а не на программиста, работа программиста при использовании мониторов существенно упрощается, а вероятность возникновения ошибок становится меньше.

Однако одних только взаимоисключений недостаточно для того, чтобы в полном объеме реализовать решение задач, возникающих при взаимодействии процессов. Нам нужны еще и средства организации очередности процессов, подобно семафорам *full* и *empty* в предыдущем примере. Для этого в мониторах было введено понятие условных переменных (*condition variables*), над которыми можно совершать две операции *wait* и *signal*, отчасти похожие на операции *P* и *V* над семафорами.

Если функция монитора не может выполняться дальше, пока не наступит некоторое событие, она выполняет операцию *wait* над какой-либо условной переменной. При этом процесс, выполнивший

операцию *wait*, блокируется, становится неактивным, и другой процесс получает возможность войти в монитор.

Когда ожидаемое событие происходит, другой процесс внутри функции-метода совершает операцию *signal* над той же самой условной переменной. Это приводит к пробуждению ранее заблокированного процесса, и он становится активным. Если несколько процессов дожидались операции *signal* для этой переменной, то активным становится только один из них. Что можно предпринять для того, чтобы у нас не оказалось двух процессов, разбудившего и пробужденного, одновременно активных внутри монитора?

Хор предложил, чтобы пробужденный процесс подавлял исполнение разбудившего процесса, пока он сам не покинет монитор. Несколько позже Хансен (Hansen) предложил другой механизм: разбудивший процесс покидает монитор немедленно после исполнения операции *signal*. Мы будем придерживаться подхода Хансена.

Необходимо отметить, что условные переменные, в отличие от семафоров Дейкстры, не умеют запоминать предысторию. Это означает, что операция *signal* всегда должна выполняться после операции *wait*. Если операция *signal* выполняется над условной переменной, с которой не связано ни одного заблокированного процесса, то информация о произошедшем событии будет утеряна. Следовательно, выполнение операции *wait* всегда будет приводить к блокированию процесса.

Давайте применим концепцию мониторов к решению задачи *производитель-потребитель*.

```
monitor ProducerConsumer {
condition full, empty;
int count;
void put() {
if(count = = N) full.wait;
put_item;
count + = 1;
if(count = = 1) empty.signal;
}
void get() {
if (count = = 0) empty.wait;
get_item();
count - = 1;
if(count = = N-1) full.signal;
}
```

```
{
count = 0;
}
}
Producer:
while(1) {
produce_item;
ProducerConsumer.put();
}
Consumer:
while(1) {
ProducerConsumer.get();
consume_item;
}
```

Легко убедиться, что приведенный пример действительно решает поставленную задачу.

Реализация мониторов требует разработки специальных языков программирования и компиляторов для них. Мониторы встречаются в таких языках, как параллельный Евклид, параллельный Паскаль, Java и т.д. Эмуляция мониторов с помощью системных вызовов для обычных широко используемых языков программирования не так проста, как эмуляция семафоров. Поэтому можно пользоваться еще одним механизмом со скрытыми взаимоисключениями, механизмом, о котором мы уже упоминали, — передачей сообщений.

6.3. Сообщения

Для прямой и непрямой адресации достаточно двух примитивов, чтобы описать передачу сообщений по линии связи — *send* и *receive*. В случае прямой адресации мы будем обозначать их так:

- *send*(*P, message*) — послать сообщение *message* процессу *P*;
- *receive*(*Q, message*) — получить сообщение *message* от процесса *Q*.

В случае непрямой адресации мы будем обозначать их так:

- *send*(*A, message*) — послать сообщение *message* в почтовый ящик *A*;
- *receive*(*A, message*) — получить сообщение *message* из почтового ящика *A*.

Примитивы *send* и *receive* уже имеют скрытый от наших глаз механизм взаимоисключения. Более того, в большинстве систем они уже имеют и скрытый механизм блокировки при чтении из пустого буфера и при записи в полностью заполненный буфер. Реализация

решения задачи producer-consumer для таких примитивов становится тривиальной. Надо отметить, что, несмотря на простоту использования, передача сообщений в пределах одного компьютера происходит существенно медленнее, чем работа с семафорами и мониторами.

6.4. Эквивалентность семафоров, мониторов и сообщений

Можно показать, что в рамках одной вычислительной системы, когда процессы имеют возможность использовать разделяемую память, все они эквивалентны. Это означает, что любые два из предложенных механизмов могут быть реализованы на базе третьего, оставшегося механизма.

Реализация мониторов и передачи сообщений с помощью семафоров

Рассмотрим сначала, как реализовать мониторы с помощью семафоров. Для этого нам нужно уметь реализовывать взаимоисключения при входе в монитор и условные переменные. Возьмем семафор *mutex* с начальным значением 1 для реализации взаимоисключения при входе в монитор и по одному семафору *ci* для каждой условной переменной. Кроме того, для каждой условной переменной заведем счетчик *fi* для индикации наличия ожидающих процессов. Когда процесс входит в монитор, компилятор будет генерировать вызов функции *monitor_enter*, которая выполняет операцию *P* над семафором *mutex* для данного монитора. При нормальном выходе из монитора (т.е. при выходе без вызова операции *signal* для условной переменной) компилятор будет генерировать вызов функции *monitor_exit*, которая выполняет операцию *V* над этим семафором.

Для выполнения операции *wait* над условной переменной компилятор будет генерировать вызов функции *wait*, которая выполняет операцию *V* для семафора *mutex*, разрешая другим процессам входить в монитор, и выполняет операцию *P* над соответствующим семафором *ci*, блокируя вызвавший процесс. Для выполнения операции *signal* над условной переменной компилятор будет генерировать вызов функции *signal_exit*, которая выполняет операцию *V* над ассоциированным семафором *ci* (если есть процессы, ожидающие соответствующего события), и выход из монитора, минуя функцию *monitor_exit*.

```
Semaphore mutex = 1;
void monitor_enter(){
```

```
P(mutex);
}
void monitor_exit(){
V(mutex);
}
Semaphore ci = 0;
int fi = 0;
void wait(i){
fi = fi + 1;
V(mutex);
P(ci);
fi = fi — 1;
}
void signal_exit(i){
if (fi)V(ci);
else V(mutex);
}
```

Заметим, что при выполнении функции *signal_exit*, если кто-либо ожидал этого события, процесс покидает монитор без увеличения значения семафора *mutex*, не разрешая тем самым всем процессам, кроме разбуженного, войти в монитор. Это увеличение совершит разбуженный процесс, когда покинет монитор обычным способом или когда выполнит новую операцию *wait* над какой-либо условной переменной.

Рассмотрим теперь, как реализовать передачу сообщений, используя семафоры. Для простоты опишем реализацию только одной очереди сообщений. Выделим в разделяемой памяти достаточно большую область под хранение сообщений, там же будем записывать, сколько пустых и заполненных ячеек находится в буфере, хранить ссылки на списки процессов, ожидающих чтения и памяти. Взаимоисключение при работе с разделяемой памятью будем обеспечивать семафором *mutex*. Также заведем по одному семафору *ci* на взаимодействующий процесс, для того чтобы обеспечивать блокирование процесса при попытке чтения из пустого буфера или при попытке записи в переполненный буфер. Посмотрим, как такой механизм будет работать. Начнем с процесса, желающего получить сообщение.

Процесс-получатель с номером *i* прежде всего выполняет операцию *P*(*mutex*), получая в монопольное владение разделяемую память. После чего он проверяет, есть ли в буфере сообщения. Если нет, то

он заносит себя в список процессов, ожидающих сообщения, выполняет *V*(*mutex*) и *P*(*ci*). Если сообщение в буфере есть, то он читает его, изменяет счетчики буфера и проверяет, есть ли процессы в списке процессов, жаждущих записи. Если таких процессов нет, то выполняется *V*(*mutex*), и процесс-получатель выходит из критической области. Если такой процесс есть (с номером *j*), то он удаляется из этого списка, выполняется *V* для его семафора *cj*, и мы выходим из критического района. Проснувшийся процесс начинает выполняться в критическом районе, так как *mutex* у нас имеет значение 0 и никто более не может попасть в критический район. При выходе из критического района именно разбуженный процесс произведет вызов *V*(*mutex*).

Как строится работа процесса-отправителя с номером *i*? Процесс, посылающий сообщение, тоже ждет, пока он не сможет иметь монополию на использование разделяемой памяти, выполнив операцию *P*(*mutex*). Далее он проверяет, есть ли место в буфере, и если да, то помещает сообщение в буфер, изменяет счетчики и смотрит, есть ли процессы, ожидающие сообщения. Если нет, выполняет *V*(*mutex*) и выходит из критической области, если есть, «будит» один из них (с номером *j*), вызывая *V*(*cj*), с одновременным удалением этого процесса из списка процессов, ожидающих сообщений, и выходит из критического региона без вызова *V*(*mutex*), предоставляя тем самым возможность разбуженному процессу прочитать сообщение. Если места в буфере нет, то процесс-отправитель заносит себя в очередь процессов, ожидающих возможности записи, и вызывает *V*(*mutex*) и *P*(*ci*).

Реализация семафоров и передачи сообщений с помощью мониторов

Нам достаточно показать, что с помощью мониторов можно реализовать семафоры, так как получать из семафоров сообщения мы уже умеем.

Самый простой способ такой реализации выглядит следующим образом. Заведем внутри монитора переменную-счетчик, связанный с эмулируемым семафором список блокируемых процессов и по одной условной переменной на каждый процесс. При выполнении операции *P* над семафором вызывающий процесс проверяет значение счетчика. Если оно больше нуля, уменьшает его на 1 и выходит из монитора.

Если оно равно 0, процесс добавляет себя в очередь процессов, ожидающих события, и выполняет операцию *wait* над своей условной

переменной. При выполнении операции *V* над семафором процесс увеличивает значение счетчика, проверяет, есть ли процессы, ожидающие этого события, и если есть, удаляет один из них из списка и выполняет операцию *signal* для условной переменной, соответствующей процессу.

Реализация семафоров и мониторов с помощью очередей сообщений

Покажем, наконец, как реализовать семафоры с помощью очередей сообщений. Для этого воспользуемся более хитрой конструкцией, введя новый синхронизирующий процесс. Этот процесс имеет счетчик и очередь для процессов, ожидающих включения семафора. Для того чтобы выполнить операции *P* и *V*, процессы посылают синхронизирующему процессу сообщения, в которых указывают свои потребности, после чего ожидают получения подтверждения от синхронизирующего процесса.

После получения сообщения синхронизирующий процесс проверяет значение счетчика, чтобы выяснить, можно ли совершить требуемую операцию. Операция *V* всегда может быть выполнена, в то время как операция *P* может потребовать блокирования процесса. Если операция может быть совершена, то она выполняется, и синхронизирующий процесс посылает подтверждающее сообщение. Если процесс должен быть блокирован, то его идентификатор заносится в очередь блокированных процессов, и подтверждение не посылается. Позднее, когда какой-либо из других процессов выполнит операцию *V*, один из блокированных процессов удаляется из очереди ожидания и получает соответствующее подтверждение.

Поскольку мы показали ранее, как из семафоров построить мониторы, мы доказали эквивалентность мониторов, семафоров и сообщений.

Глава 7
ТУПИКИ И БОРЬБА С НИМИ

В предыдущих главах мы рассматривали способы синхронизации процессов, которые позволяют процессам успешно кооперироваться. Однако в некоторых случаях могут возникнуть непредвиденные затруднения. Предположим, что несколько процессов конкурируют за обладание конечным числом ресурсов. Если запрашиваемый процессом ресурс недоступен, ОС переводит данный процесс в состояние ожидания. В случае когда требуемый ресурс удерживается другим ожидающим процессом, первый процесс не сможет сменить свое состояние. Такая ситуация называется тупиком (deadlock). Говорят, что в мультипрограммной системе процесс находится в состоянии тупика, если он ожидает события, которое никогда не произойдет. Системная тупиковая ситуация, или «зависание системы», является следствием того, что один или более процессов находятся в состоянии тупика. Иногда подобные ситуации называют взаимоблокировками. В общем случае проблема тупиков эффективного решения не имеет.

Рассмотрим пример. Предположим, что два процесса осуществляют вывод с ленты на принтер. Один из них успел монополизировать ленту и претендует на принтер, а другой — наоборот. После этого оба процесса оказываются заблокированными в ожидании второго ресурса (рис. 7.1).

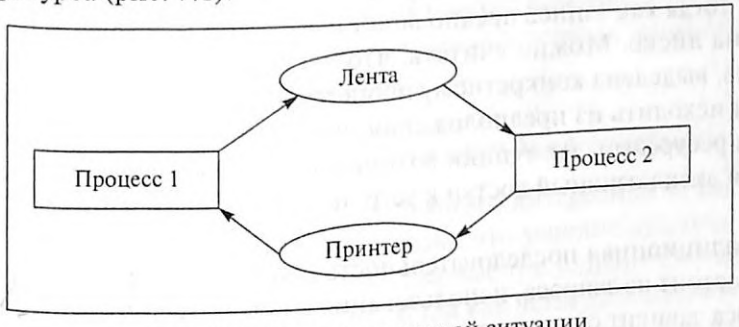

Рис. 7.1. Пример тупиковой ситуации

Множество процессов находится в тупиковой ситуации, если каждый процесс из множества ожидает события, которое может вызвать только другой процесс данного множества. Так как все процессы чего-то ожидают, то ни один из них не сможет инициировать

событие, которое разбудило бы другого члена множества и, следовательно, все процессы будут спать вместе.

Выше приведен пример взаимоблокировки, возникающей при работе с так называемыми выделенными устройствами. Тупики, однако, могут иметь место и в других ситуациях. Например, в системах управления базами данных записи могут быть локализованы процессами, чтобы избежать состояния гонок. В этом случае может получиться так, что один из процессов заблокировал записи, необходимые другому процессу, и наоборот. Таким образом, тупики могут иметь место как на аппаратных, так и на программных ресурсах.

Тупики также могут быть вызваны ошибками программирования. Например, процесс может напрасно ждать открытия семафора, потому что в некорректно написанном приложении эту операцию забыли предусмотреть. Другой причиной бесконечного ожидания может быть дискриминационная политика по отношению к некоторым процессам. Однако чаще всего событие, которого ждет процесс в тупиковой ситуации, — освобождение ресурса, поэтому в дальнейшем будут рассмотрены методы борьбы с тупиками ресурсного типа.

Ресурсами могут быть как устройства, так и данные. Некоторые ресурсы допускают разделение между процессами, т.е. являются **разделяемыми** ресурсами. Например, память, процессор, диски коллективно используются процессами. Другие не допускают разделения, т.е. являются **выделенными**, например лентопротяжное устройство. К взаимоблокировке может привести использование как выделенных, так и разделяемых ресурсов. Например, чтение с разделяемого диска может одновременно осуществляться несколькими процессами, тогда как запись предполагает исключительный доступ к данным на диске. Можно считать, что часть диска, куда происходит запись, выделена конкретному процессу. Поэтому в дальнейшем мы будем исходить из предположения, что тупики связаны с выделенными ресурсами, т.е. тупики возникают, когда процессу предоставляется эксклюзивный доступ к устройствам, файлам и другим ресурсам.

Традиционная последовательность событий при работе с ресурсом состоит из запроса, использования и освобождения ресурса. Тип запроса зависит от природы ресурса и от ОС. Запрос может быть явным, например специальный вызов *request*, или неявным — *open* для открытия файла. Обычно, если ресурс занят и запрос отклонен, запрашивающий процесс переходит в состояние ожидания.

Далее в данной главе будут рассматриваться вопросы обнаружения, предотвращения, обхода тупиков и восстановления после тупиков. Как правило, борьба с тупиками — очень дорогостоящее мероприятие.

Тем не менее, для ряда систем, например для систем реального времени, иного выхода нет.

7.1. Условия возникновения тупиков

Условия возникновения тупиков были сформулированы Коффманом, Элфиком и Шошани в 1970 г.

1. Условие взаимоисключения (*Mutual exclusion*). Одновременно использовать ресурс может только один процесс.
2. Условие ожидания ресурсов (*Hold and wait*). Процессы удерживают ресурсы, уже выделенные им, и могут запрашивать другие ресурсы.
3. Условие неперераспределяемости (*No preemtion*). Ресурс, выделенный ранее, не может быть принудительно забран у процесса. Освобождены они могут быть только процессом, который их удерживает.
4. Условие кругового ожидания (*Circular wait*). Существует кольцевая цепь процессов, в которой каждый процесс ждет доступа к ресурсу, удерживаемому другим процессом цепи.

Для образования тупика необходимым и достаточным является выполнение **всех четырех** условий.

Обычно тупик моделируется циклом в графе, состоящем из узлов двух видов: прямоугольников — процессов и эллипсов — ресурсов, наподобие того, что изображен на рис. 7.1. Стрелки, направленные от ресурса к процессу, показывают, что ресурс выделен данному процессу. Стрелки, направленные от процесса к ресурсу, означают, что процесс запрашивает данный ресурс.

7.2. Основные направления борьбы с тупиками

Проблема тупиков инициировала много интересных исследований в области информатики. Очевидно, что условие циклического ожидания отличается от остальных. Первые три условия формируют правила, существующие в системе, тогда как четвертое условие описывает ситуацию, которая может сложиться при определенной неблагоприятной последовательности событий. Поэтому методы предотвращения взаимоблокировок ориентированы главным образом на нарушение первых трех условий путем введения ряда ограничений на поведение процессов и способы распределения ресурсов. Методы

обнаружения и устранения менее консервативны и сводятся к поиску и разрыву цикла ожидания ресурсов.

Итак, основные направления борьбы с тупиками:

- игнорирование проблемы в целом;
- предотвращение тупиков;
- обнаружение тупиков;
- восстановление после тупиков.

Игнорирование проблемы тупиков

Простейший подход — не замечать проблему тупиков. Для того чтобы принять такое решение, необходимо оценить вероятность возникновения взаимоблокировки и сравнить ее с вероятностью ущерба от других отказов аппаратного и программного обеспечения. Проектировщики обычно не желают жертвовать производительностью системы или удобством пользователей для внедрения сложных и дорогостоящих средств борьбы с тупиками.

Любая ОС, имеющая в ядре ряд массивов фиксированной размерности, потенциально страдает от тупиков, даже если они не обнаружены. Таблица открытых файлов, таблица процессов, фактически каждая таблица являются ограниченными ресурсами. Заполнение всех записей таблицы процессов может привести к тому, что очередной запрос на создание процесса может быть отклонен. При неблагоприятном стечении обстоятельств несколько процессов могут выдать такой запрос одновременно и оказаться в тупике. Следует ли отказываться от вызова *CreateProcess*, чтобы решить эту проблему?

Подход большинства популярных ОС (Unix, Windows и др.) состоит в том, чтобы игнорировать данную проблему в предположении, что маловероятный случайный тупик предпочтительнее, чем нелепые правила, заставляющие пользователей ограничивать число процессов, открытых файлов и т.п. Сталкиваясь с нежелательным выбором между строгостью и удобством, трудно найти решение, которое устраивало бы всех.

Способы предотвращения тупиков

Цель предотвращения тупиков — обеспечить условия, исключающие возможность возникновения тупиковых ситуаций. Большинство методов связано с предотвращением одного из условий возникновения взаимоблокировки.

Система, предоставляя ресурс в распоряжение процесса, должна принять решение, безопасно это или нет. Возникает вопрос: есть ли такой алгоритм, который помогает всегда избегать тупиков и делать

правильный выбор. Ответ — да, мы можем избегать тупиков, но только если определенная информация известна заранее.

Способы предотвращения тупиков путем тщательного распределения ресурсов. Алгоритм банкира

Можно избежать взаимоблокировки, если распределять ресурсы, придерживаясь определенных правил.

Среди такого рода алгоритмов наиболее известен алгоритм банкира, предложенный Дейкстрой, который базируется на так называемых безопасных или надежных состояниях (*safe state*). Безопасное состояние — это такое состояние, для которого имеется по крайней мере одна последовательность событий, которая не приведет к взаимоблокировке. Модель алгоритма основана на действиях банкира, который, имея в наличии капитал, выдает кредиты.

Суть алгоритма состоит в следующем.

- Предположим, что у системы в наличии n устройств, например лент.
- ОС принимает запрос от пользовательского процесса, если его максимальная потребность не превышает n.
- Пользователь гарантирует, что если ОС в состоянии удовлетворить его запрос, то все устройства будут возвращены системе в течение конечного времени.
- Текущее состояние системы называется **надежным**, если ОС может обеспечить всем процессам их выполнение в течение конечного времени.
- В соответствии с алгоритмом банкира выделение устройств возможно, только если состояние системы остается надежным.

Рассмотрим пример надежного состояния для системы с 3 пользователями и 11 устройствами, где 9 устройств задействовано, а 2 имеется в резерве (табл. 7.1).

Таблица 7.1

Пример надежного состояния системы

Пользователи	Максимальная потребность в ресурсах	Выделенное пользователям количество ресурсов
Первый	9	6
Второй	10	2
Третий	3	1

Данное состояние надежно. Последующие действия системы могут быть таковы. Вначале удовлетворить запросы третьего пользова-

теля, затем дождаться, когда он закончит работу и освободит свои три устройства. Затем можно обслужить первого и второго пользователей. Т.е. система удовлетворяет только те запросы, которые оставляют ее в надежном состоянии, и отклоняет остальные.

Термин «**ненадежное** состояние» не предполагает, что обязательно возникнут тупики. Он лишь говорит о том, что в случае неблагоприятной последовательности событий система может зайти в тупик.

Данный алгоритм обладает тем достоинством, что при его использовании нет необходимости в перераспределении ресурсов и откате процессов назад. Однако использование этого метода требует выполнения ряда условий.

- Число пользователей и число ресурсов фиксировано.
- Число работающих пользователей должно оставаться постоянным.
- Алгоритм требует, чтобы клиенты гарантированно возвращали ресурсы.
- Должны быть заранее указаны максимальные требования процессов к ресурсам. Чаще всего данная информация отсутствует.

Наличие таких жестких и зачастую неприемлемых требований может склонить разработчиков к выбору других решений проблемы взаимоблокировки.

Предотвращение тупиков за счет нарушения условий возникновения тупиков

В отсутствие информации о будущих запросах единственный способ избежать взаимоблокировки — добиться невыполнения хотя бы одного из условий раздела «Условия возникновения тупиков».

Нарушение условия взаимоисключения

В общем случае избежать взаимоисключений невозможно. Доступ к некоторым ресурсам должен быть исключительным. Тем не менее, некоторые устройства удается обобществить. В качестве примера рассмотрим принтер. Известно, что пытаться осуществлять вывод на принтер могут несколько процессов.

Во избежание хаоса организуют промежуточное формирование всех выходных данных процесса на диске, т.е. разделяемом устройстве. Лишь один системный процесс, называемый сервисом или демоном принтера, отвечающий за вывод документов на печать по мере освобождения принтера, реально с ним взаимодействует. Эта схема называется спулингом (*spooling*). Таким образом, принтер становится разделяемым устройством, и тупик для него устранен.

К сожалению, не для всех устройств и не для всех данных можно организовать спулинг. Неприятным побочным следствием такой модели может быть потенциальная тупиковая ситуация из-за конкуренции за дисковое пространство для буфера спулинга. Тем не менее, в той или иной форме эта идея применяется часто.

Нарушение условия ожидания дополнительных ресурсов

Условия ожидания ресурсов можно избежать, потребовав выполнения стратегии двухфазного захвата.

- В первой фазе процесс должен запрашивать все необходимые ему ресурсы сразу. До тех пор пока они не предоставлены, процесс не может продолжать выполнение.
- Если в первой фазе некоторые ресурсы, которые были нужны данному процессу, уже заняты другими процессами, он освобождает все ресурсы, которые были ему выделены, и пытается повторить первую фазу.

В известном смысле этот подход напоминает требование захвата всех ресурсов заранее. Естественно, что только специально организованные программы могут быть приостановлены в течение первой фазы и рестартованы впоследствии.

Таким образом, один из способов — заставить все процессы затребовать нужные им ресурсы перед выполнением («все или ничего»). Если система в состоянии выделить процессу все необходимое, он может работать до завершения. Если хотя бы один из ресурсов занят, процесс будет ждать.

Данное решение применяется в пакетных мэйнфреймах (*mainframe*), которые требуют от пользователей перечислить все необходимые его программе ресурсы. Другим примером может служить механизм двухфазной локализации записей в СУБД. Однако в целом подобный подход не слишком привлекателен и приводит к неэффективному использованию компьютера. Как уже отмечалось, перечень будущих запросов к ресурсам редко удается спрогнозировать. Если такая информация есть, то можно воспользоваться алгоритмом банкира. Заметим также, что описываемый подход противоречит парадигме модульности в программировании, поскольку приложение должно знать о предполагаемых запросах к ресурсам во всех модулях.

Нарушение принципа отсутствия перераспределения

Если бы можно было отбирать ресурсы у удерживающих их процессов до завершения этих процессов, то удалось бы добиться невы-

полнения третьего условия возникновения тупиков. Перечислим минусы данного подхода.

Во-первых, отбирать у процессов можно только те ресурсы, состояние которых легко сохранить, а позже восстановить, например состояние процессора.

Во-вторых, если процесс в течение некоторого времени использует определенные ресурсы, а затем освобождает эти ресурсы, он может потерять результаты работы, проделанной до настоящего момента.

Наконец, следствием данной схемы может быть дискриминация отдельных процессов, у которых постоянно отбирают ресурсы.

Весь вопрос в цене подобного решения, которая может быть слишком высокой, если необходимость отбирать ресурсы возникает часто.

Нарушение условия кругового ожидания

Трудно предложить разумную стратегию, чтобы избежать последнего условия из раздела «Условия возникновения тупиков» — циклического ожидания.

Один из способов — упорядочить ресурсы. Например, можно присвоить всем ресурсам уникальные номера и потребовать, чтобы процессы запрашивали ресурсы в порядке их возрастания. Тогда круговое ожидание возникнуть не может. После последнего запроса и освобождения всех ресурсов можно разрешить процессу опять осуществить первый запрос. Очевидно, что практически невозможно найти порядок, который удовлетворит всех.

Один из немногих примеров упорядочивания ресурсов — создание иерархии спин-блокировок в Windows 2000. Спин-блокировка — простейший способ синхронизации. Спин-блокировка может быть захвачена и освобождена процессом. Классическая тупиковая ситуация возникает, когда процесс P_1 захватывает спин-блокировку S_1 и претендует на спин-блокировку S_2, а процесс P_2 захватывает спин-блокировку S_2 и хочет дополнительно захватить спин-блокировку S_1. Чтобы этого избежать, все спин-блокировки помещаются в упорядоченный список. Захват может осуществляться только в порядке, указанном в списке.

Другой способ атаки условия кругового ожидания — действовать в соответствии с правилом, согласно которому каждый процесс может иметь только один ресурс в каждый момент времени. Если нужен второй ресурс — освободи первый. Очевидно, что для многих процессов это неприемлемо.

Таким образом, технология предотвращения циклического ожидания, как правило, неэффективна и может без необходимости закрывать доступ к ресурсам.

Обнаружение тупиков

Обнаружение взаимоблокировки сводится к фиксации тупиковой ситуации и выявлению вовлеченных в нее процессов. Для этого производится проверка наличия циклического ожидания в случаях, когда выполнены первые три условия возникновения тупика. Методы обнаружения активно используют графы распределения ресурсов.

Рассмотрим модельную ситуацию.

- Процесс P_1 ожидает ресурс R_1.
- Процесс P_2 удерживает ресурс R_2 и ожидает ресурс R_1.
- Процесс P_3 удерживает ресурс R_1 и ожидает ресурс R_3.
- Процесс P_4 ожидает ресурс R_2.
- Процесс P_5 удерживает ресурс R_3 и ожидает ресурс R_2.

Вопрос состоит в том, является ли данная ситуация тупиковой, и если да, то какие процессы в ней участвуют. Для ответа на этот вопрос можно сконструировать граф ресурсов, как показано на рис. 7.2. Из рисунка видно, что имеется цикл, моделирующий условие кругового ожидания, и что процессы P_2, P_3, P_5, а может быть, и другие находятся в тупиковой ситуации.

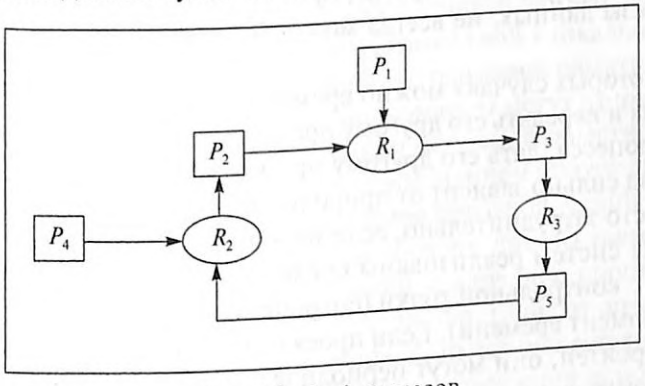

Рис. 7.2. Граф ресурсов

Визуально легко обнаружить наличие тупика, но нужны также формальные алгоритмы, реализуемые на компьютере.

Существуют и другие способы обнаружения тупиков, применимые также в ситуациях, когда имеется несколько ресурсов каждого типа.

Восстановление после тупиков

Обнаружив тупик, можно вывести из него систему, нарушив одно из условий существования тупика. При этом, возможно, несколько процессов частично или полностью потеряют результаты проделанной работы.

Сложность восстановления обусловлена рядом факторов.

- В большинстве систем нет достаточно эффективных средств, чтобы приостановить процесс, вывести его из системы и возобновить впоследствии с того места, где он был остановлен.
- Если даже такие средства есть, то их использование требует затрат и внимания оператора.
- Восстановление после тупика может потребовать значительных усилий.

Самый простой и наиболее распространенный способ устранить тупик — завершить выполнение одного или более процессов, чтобы впоследствии использовать его ресурсы. Тогда в случае удачи остальные процессы смогут выполняться. Если это не помогает, можно ликвидировать еще несколько процессов. После каждой ликвидации должен запускаться алгоритм обнаружения тупика.

По возможности лучше ликвидировать тот процесс, который может быть без ущерба возвращен к началу (такие процессы называются идемпотентными). Примером такого процесса может служить компиляция. С другой стороны, процесс, который изменяет содержимое базы данных, не всегда может быть корректно запущен повторно.

В некоторых случаях можно временно забрать ресурс у текущего владельца и передать его другому процессу. Возможность забрать ресурс у процесса, дать его другому процессу и затем без ущерба вернуть назад сильно зависит от природы ресурса. Подобное восстановление часто затруднительно, если не невозможно.

В ряде систем реализованы средства отката и перезапуска или рестарта с контрольной точки (сохранение состояния системы в какой-то момент времени). Если проектировщики системы знают, что тупик вероятен, они могут периодически организовывать для процессов контрольные точки. Иногда это приходится делать разработчикам прикладных программ.

Когда тупик обнаружен, видно, какие ресурсы вовлечены в цикл кругового ожидания. Чтобы осуществить восстановление, процесс, который владеет таким ресурсом, должен быть отброшен к моменту времени, предшествующему его запросу на этот ресурс.

Глава 8
ПРОСТЕЙШИЕ СХЕМЫ УПРАВЛЕНИЯ ПАМЯТЬЮ

Главная задача компьютерной системы — выполнять программы. Программы вместе с данными, к которым они имеют доступ, в процессе выполнения должны (по крайней мере частично) находиться в оперативной памяти. Операционной системе приходится решать задачу распределения памяти между пользовательскими процессами и компонентами ОС. Эта деятельность называется управлением памятью. Таким образом, память (*storage, memory*) является важнейшим ресурсом, требующим тщательного управления. В недавнем прошлом память была самым дорогим ресурсом.

Часть ОС, которая отвечает за управление памятью, называется менеджером памяти.

8.1. Физическая организация памяти компьютера

Запоминающие устройства компьютера разделяют как минимум на два уровня: основную (главную, оперативную, физическую) и вторичную (внешнюю) память.

Основная память представляет собой упорядоченный массив однобайтовых ячеек, каждая из которых имеет свой уникальный адрес (номер). Процессор извлекает команду из основной памяти, декодирует и выполняет ее. Для выполнения команды могут потребоваться обращения еще к нескольким ячейкам основной памяти. Обычно основная память изготавливается с применением полупроводниковых технологий и теряет свое содержимое при отключении питания.

Вторичную память (это главным образом диски) также можно рассматривать как одномерное линейное адресное пространство, состоящее из последовательности байтов. В отличие от оперативной памяти она является энергонезависимой, имеет существенно большую емкость и используется в качестве расширения основной памяти.

Эту схему можно дополнить еще несколькими промежуточными уровнями, как показано на рис. 8.1. Разновидности памяти могут быть объединены в иерархию по убыванию времени доступа, возрастанию цены и увеличению емкости.

Рис. 8.1. Иерархия памяти

Многоуровневую схему используют следующим образом. Информация, которая находится в памяти верхнего уровня, обычно хранится также на уровнях с большими номерами. Если процессор не обнаруживает нужную информацию на i-м уровне, он начинает искать ее на следующих уровнях. Когда нужная информация найдена, она переносится в более быстрые уровни.

8.2. Локальность

Оказывается, при таком способе организации по мере снижения скорости доступа к уровню памяти снижается также и частота обращений к нему.

Ключевую роль здесь играет свойство реальных программ, в течение ограниченного отрезка времени способных работать с небольшим набором адресов памяти. Это эмпирически наблюдаемое свойство известно как принцип локальности или локализации обращений.

Свойство локальности (соседние в пространстве и времени объекты характеризуются похожими свойствами) присуще не только функционированию ОС, но и природе вообще. В случае ОС свойство локальности объяснимо, если учесть, как пишутся программы и как хранятся данные, т.е. обычно в течение какого-то отрезка времени ограниченный фрагмент кода работает с ограниченным набором данных. Эту часть кода и данных удается разместить в памяти с быстрым доступом. В результате реальное время доступа к памяти определяется временем доступа к верхним уровням, что и обусловливает эффективность использования иерархической схемы. Надо сказать, что описываемая организация вычислительной системы во многом имитирует деятельность человеческого мозга при перера-

ботке информации. Действительно, решая конкретную проблему, человек работает с небольшим объемом информации, храня не относящиеся к делу сведения в своей памяти или во внешней памяти (например, в книгах).

Кэш процессора обычно является частью аппаратуры, поэтому менеджер памяти ОС занимается распределением информации главным образом в основной и внешней памяти компьютера. В некоторых схемах потоки между оперативной и внешней памятью регулируются программистом, однако это связано с затратами времени программиста, так что подобную деятельность стараются возложить на ОС.

Адреса в основной памяти, характеризующие реальное расположение данных в физической памяти, называются физическими адресами. Набор физических адресов, с которым работает программа, называют физическим адресным пространством.

8.3. Логическая память

Аппаратная организация памяти в виде линейного набора ячеек не соответствует представлениям программиста о том, как организовано хранение программ и данных. Большинство программ представляет собой набор модулей, созданных независимо друг от друга. Иногда все модули, входящие в состав процесса, располагаются в памяти один за другим, образуя линейное пространство адресов. Однако чаще модули помещаются в разные области памяти и используются по-разному.

Схема управления памятью, поддерживающая этот взгляд пользователя на то, как хранятся программы и данные, называется сегментацией. Сегмент — область памяти определенного назначения, внутри которой поддерживается линейная адресация. Сегменты содержат процедуры, массивы, стек или скалярные величины, но обычно не содержат информацию смешанного типа.

По-видимому, вначале сегменты памяти появились в связи с необходимостью обобществления процессами фрагментов программного кода (текстовый редактор, тригонометрические библиотеки и т.д.), без чего каждый процесс должен был хранить в своем адресном пространстве дублирующую информацию.

Эти отдельные участки памяти, хранящие информацию, которую система отображает в память нескольких процессов, получили название сегментов. Память, таким образом, перестала быть линейной

и превратилась в двумерную. Адрес состоит из двух компонентов: номер сегмента, смещение внутри сегмента.

Далее оказалось удобным размещать в разных сегментах различные компоненты процесса (код программы, данные, стек и т.д.). Попутно выяснилось, что можно контролировать характер работы с конкретным сегментом, приписав ему атрибуты, например права доступа или типы операций, которые разрешается производить с данными, хранящимися в сегменте.

Некоторые сегменты, описывающие адресное пространство процесса, показаны на рис. 8.2.

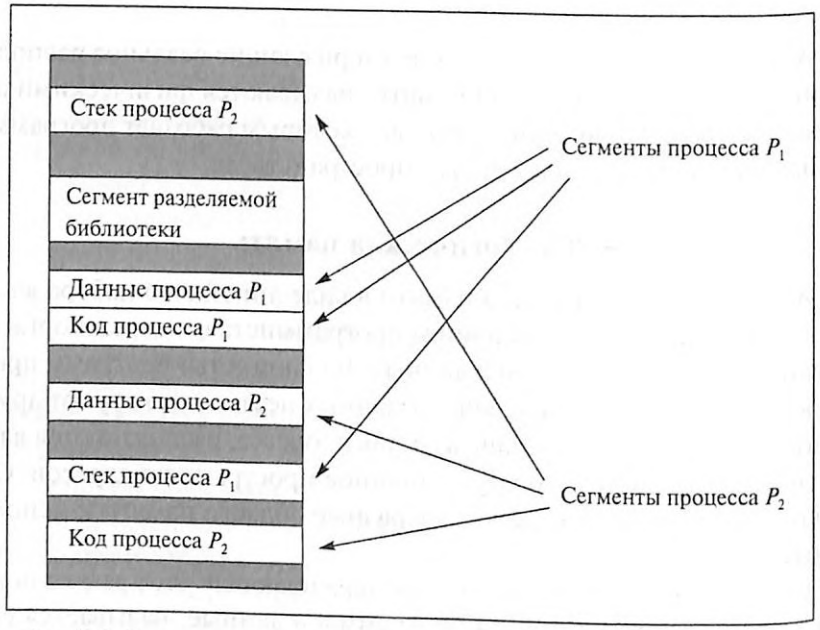

Рис. 8.2. Расположение сегментов процессов в памяти компьютера

Большинство современных ОС поддерживают сегментную организацию памяти. В некоторых архитектурах (Intel, например) сегментация поддерживается оборудованием.

Адреса, к которым обращается процесс, таким образом, отличаются от адресов, реально существующих в оперативной памяти. В каждом конкретном случае используемые программой адреса могут быть представлены различными способами. Например, адреса в исходных текстах обычно символические. Компилятор связывает эти символические адреса с перемещаемыми адресами (такими, как *n* байт от начала модуля). Подобный адрес, сгенерированный программой, обычно называют логическим (в системах с виртуальной памятью он часто называется виртуальным) адресом. Совокупность всех логических адресов называется логическим (виртуальным) адресным пространством.

8.4. Связывание адресов

Логические и физические адресные пространства ни по организации, ни по размеру не соответствуют друг другу. Максимальный размер логического адресного пространства обычно определяется разрядностью процессора (например, 2^{32}) и в современных системах значительно превышает размер физического адресного пространства. Следовательно, процессор и ОС должны быть способны отобразить ссылки в коде программы в реальные физические адреса, соответствующие текущему расположению программы в основной памяти. Такое отображение адресов называют трансляцией (привязкой) адреса или связыванием адресов (рис. 8.3).

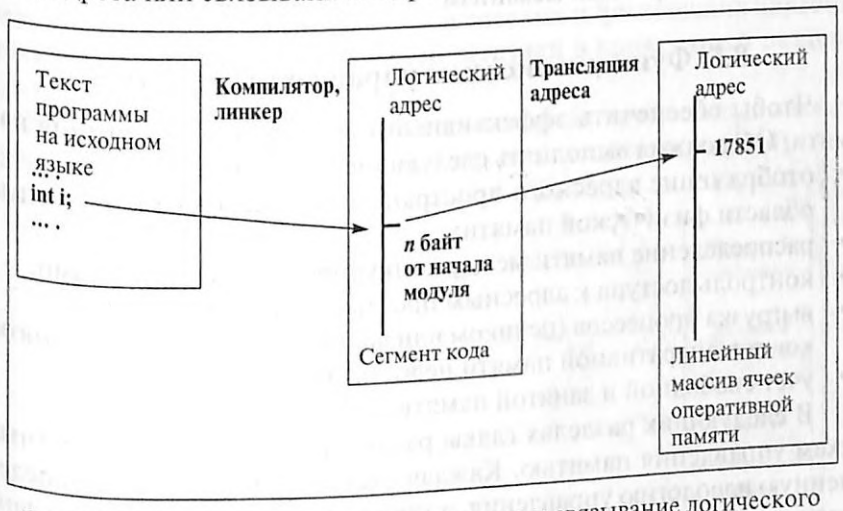

Рис. 8.3. Формирование логического адреса и связывание логического адреса с физическим

Связывание логического адреса, порожденного оператором программы, с физическим должно быть осуществлено до начала выполнения оператора или в момент его выполнения. Таким образом, привязка инструкций и данных к памяти в принципе может быть сделана на следующих шагах [Silberschatz]:

1. Этап компиляции (*Compile time*). Когда на стадии компиляции известно точное место размещения процесса в памяти, тогда непо-

средственно генерируются физические адреса. При изменении стартового адреса программы необходимо перекомпилировать ее код. В качестве примера можно привести com-программы MS-DOS, которые связывают ее с физическими адресами на стадии компиляции.

2. Этап загрузки (*Load time*). Если информация о размещении программы на стадии компиляции отсутствует, компилятор генерирует перемещаемый код. В этом случае окончательное связывание откладывается до момента загрузки. Если стартовый адрес меняется, нужно всего лишь перезагрузить код с учетом измененной величины.

3. Этап выполнения (*Execution time*). Если процесс может быть перемещен во время выполнения из одной области памяти в другую, связывание откладывается до стадии выполнения. Здесь желательно наличие специализированного оборудования, например регистров перемещения. Их значение прибавляется к каждому адресу, сгенерированному процессом. Большинство современных ОС осуществляет трансляцию адресов на этапе выполнения, используя для этого специальный аппаратный механизм.

8.5. Функции системы управления памятью

Чтобы обеспечить эффективный контроль использования памяти, ОС должна выполнять следующие функции:

- отображение адресного пространства процесса на конкретные области физической памяти;
- распределение памяти между конкурирующими процессами;
- контроль доступа к адресным пространствам процессов;
- выгрузка процессов (целиком или частично) во внешнюю память, когда в оперативной памяти недостаточно места;
- учет свободной и занятой памяти.

В следующих разделах главы рассматривается ряд конкретных схем управления памятью. Каждая схема включает в себя определенную идеологию управления, а также алгоритмы и структуры данных и зависит от архитектурных особенностей используемой системы. Вначале будут рассмотрены простейшие схемы.

8.6. Простейшие схемы управления памятью

Первые ОС применяли очень простые методы управления памятью. Вначале каждый процесс пользователя должен был полностью поместиться в основной памяти, занимать непрерывную область памяти, а система принимала к обслуживанию дополнительные пользовательские процессы до тех пор, пока все они одновременно помещались в основной памяти. Затем появился «простой свопинг» (система по-прежнему размещает каждый процесс в основной памяти целиком, но иногда на основании некоторого критерия целиком сбрасывает образ некоторого процесса из основной памяти во внешнюю и заменяет его в основной памяти образом другого процесса). Такого рода схемы имеют не только историческую ценность. В настоящее время они применяются в учебных и научно-исследовательских модельных ОС, а также в ОС для встроенных (embedded) компьютеров.

Схема с фиксированными разделами

Самым простым способом управления оперативной памятью является ее предварительное (обычно на этапе генерации или в момент загрузки системы) разбиение на несколько разделов фиксированной величины. Поступающие процессы помещаются в тот или иной раздел. При этом происходит условное разбиение физического адресного пространства. Связывание логических и физических адресов процесса происходит на этапе его загрузки в конкретный раздел, иногда — на этапе компиляции.

Каждый раздел может иметь свою очередь процессов, а может существовать и глобальная очередь для всех разделов (рис. 8.4).

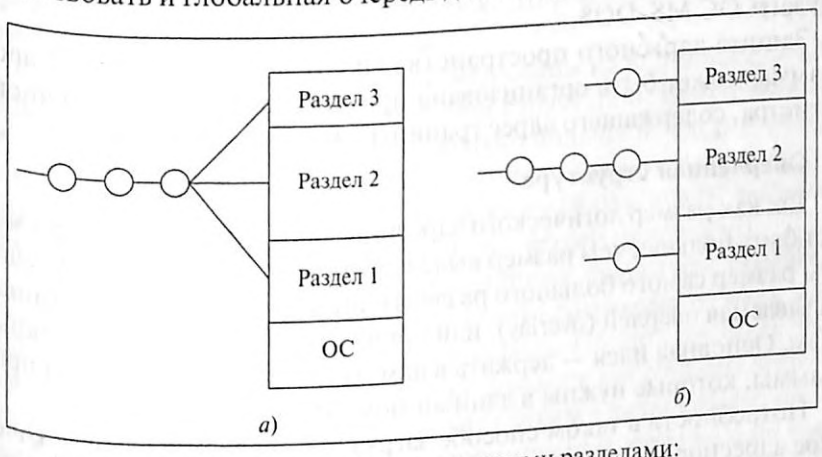

Рис. 8.4. Схема с фиксированными разделами:
а — с общей очередью процессов; *б* — с отдельными очередями процессов

Эта схема была реализована в IBM OS/360 (MFT), DEC RSX-11 и ряде других систем.

Подсистема управления памятью оценивает размер поступившего процесса, выбирает подходящий для него раздел, осуществляет загрузку процесса в этот раздел и настройку адресов.

Очевидный недостаток этой схемы — число одновременно выполняемых процессов ограничено числом разделов.

Другим существенным недостатком является то, что предлагаемая схема сильно страдает от внутренней фрагментации — потери части памяти, выделенной процессу, но не используемой им. Фрагментация возникает потому, что процесс не полностью занимает выделенный ему раздел или потому, что некоторые разделы слишком малы для выполняемых пользовательских программ.

Один процесс в памяти

Частный случай схемы с фиксированными разделами — работа менеджера памяти однозадачной ОС. В памяти размещается один пользовательский процесс. Остается определить, где располагается пользовательская программа по отношению к ОС — в верхней части памяти, в нижней или в средней. Причем часть ОС может быть в ROM (например, BIOS, драйверы устройств). Главный фактор, влияющий на это решение, — расположение вектора прерываний, который обычно локализован в нижней части памяти, поэтому ОС также размещают в нижней. Примером такой организации может служить ОС MS-DOS.

Защита адресного пространства ОС от пользовательской программы может быть организована при помощи одного граничного регистра, содержащего адрес границы ОС.

Оверлейная структура

Так как размер логического адресного пространства процесса может быть больше, чем размер выделенного ему раздела (или больше, чем размер самого большого раздела), иногда используется техника, называемая оверлей (overlay), или организация структуры с перекрытием. Основная идея — держать в памяти только те инструкции программы, которые нужны в данный момент.

Потребность в таком способе загрузки появляется, если логическое адресное пространство системы мало, например 1 Мбайт (MS-DOS) или даже всего 64 Кбайта (PDP-11), а программа относительно велика. На современных 32-разрядных системах, где виртуальное адресное пространство измеряется гигабайтами, проблемы с нехваткой памяти решаются другими способами.

Коды ветвей оверлейной структуры программы находятся на диске как абсолютные образы памяти и считываются драйвером оверлеев при необходимости. Для описания оверлейной структуры обычно используется специальный несложный язык (overlay description language). Совокупность файлов исполняемой программы дополняется файлом (обычно с расширением .odl), описывающим дерево вызовов внутри программы. Для примера, приведенного на рис. 8.5, текст этого файла может выглядеть так:

A-(*B*,*C*)
C-(D,E)

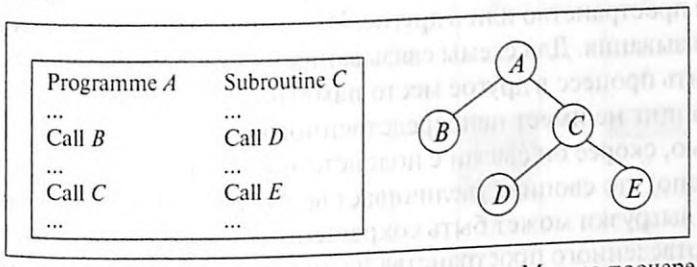

Рис. 8.5. Организация структуры с перекрытием. Можно поочередно загружать в память ветви *A–B*, *A–C–D* и *A–C–E* программы

Синтаксис подобного файла может распознаваться загрузчиком. Привязка к физической памяти происходит в момент очередной загрузки одной из ветвей программы.

Оверлеи могут быть полностью реализованы на пользовательском уровне в системах с простой файловой структурой. Операционная система при этом лишь делает несколько больше операций ввода-вывода. Типовое решение — порождение линкером специальных команд, которые включают загрузчик каждый раз, когда требуется обращение к одной из перекрывающихся ветвей программы.

Тщательное проектирование оверлейной структуры отнимает много времени и требует знания устройства программы, ее кода, данных и языка описания оверлейной структуры. По этой причине применение оверлеев ограничено компьютерами с небольшим логическим адресным пространством. Как мы увидим в дальнейшем, проблема оверлейных сегментов, контролируемых программистом, отпадает благодаря появлению систем виртуальной памяти.

Заметим, что возможность организации структур с перекрытиями во многом обусловлена свойством локальности, которое позволяет хранить в памяти только ту информацию, которая необходима в конкретный момент вычислений.

Динамическое распределение. Свопинг

Имея дело с пакетными системами, можно обходиться фиксированными разделами и не использовать ничего более сложного. В системах с разделением времени возможна ситуация, когда память не в состоянии содержать все пользовательские процессы. Приходится прибегать к *свопингу* (*swapping*) — перемещению процессов из главной памяти на диск и обратно целиком. Частичная выгрузка процессов на диск осуществляется в системах со страничной организацией (*paging*) и будет рассмотрена ниже.

Выгруженный процесс может быть возвращен в то же самое адресное пространство или в другое. Это ограничение диктуется методом связывания. Для схемы связывания на этапе выполнения можно загрузить процесс в другое место памяти.

Свопинг не имеет непосредственного отношения к управлению памятью, скорее он связан с подсистемой планирования процессов. Очевидно, что свопинг увеличивает время переключения контекста. Время выгрузки может быть сокращено за счет организации специально отведенного пространства на диске (раздел для свопинга). Обмен с диском при этом осуществляется блоками большего размера, т.е. быстрее, чем через стандартную файловую систему. Во многих версиях Unix свопинг начинает работать только тогда, когда возникает необходимость в снижении загрузки системы.

Схема с переменными разделами

В принципе система свопинга может базироваться на фиксированных разделах. Более эффективной, однако, представляется схема динамического распределения или схема с переменными разделами, которая может использоваться и в тех случаях, когда все процессы целиком помещаются в памяти, т.е. в отсутствие свопинга. В этом случае вначале вся память свободна и не разделена заранее на разделы. Вновь поступающей задаче выделяется строго необходимое количество памяти, не более. После выгрузки процесса память временно освобождается. По истечении некоторого времени память представляет собой переменное число разделов разного размера (рис. 8.6). Смежные свободные участки могут быть объединены.

В какой раздел помещать процесс? Наиболее распространены три стратегии.

- Стратегия первого подходящего (First fit). Процесс помещается в первый подходящий по размеру раздел.

- Стратегия наиболее подходящего (Best fit). Процесс помещается в тот раздел, где после его загрузки останется меньше всего свободного места.
- Стратегия наименее подходящего (Worst fit). При помещении в самый большой раздел в нем остается достаточно места для возможного размещения еще одного процесса.

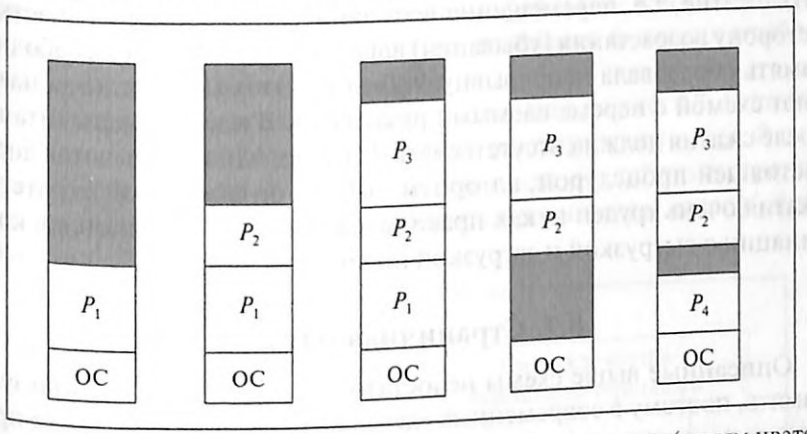

Рис. 8.6. Динамика распределения памяти между процессами (серым цветом показана неиспользуемая память)

Моделирование показало, что доля полезно используемой памяти в первых двух случаях больше, при этом первый способ несколько быстрее. Попутно заметим, что перечисленные стратегии широко применяются и другими компонентами операционной системы, например, для размещения файлов на диске.

Типовой цикл работы менеджера памяти состоит в анализе запроса на выделение свободного участка (раздела), выборе его среди имеющихся в соответствии с одной из стратегий (первого подходящего, наиболее подходящего и наименее подходящего), загрузке процесса в выбранный раздел и последующих изменениях таблиц свободных и занятых областей. Аналогичная корректировка необходима и после завершения процесса. Связывание адресов может осуществляться на этапах загрузки и выполнения.

Этот метод более гибок по сравнению с методом фиксированных разделов, однако ему присуща внешняя фрагментация — наличие большого числа участков неиспользуемой памяти, не выделенной ни одному процессу. Выбор стратегии размещения процесса между первым подходящим и наиболее подходящим слабо влияет на величину фрагментации. Любопытно, что метод наиболее подходящего может

оказаться наихудшим, так как он оставляет множество мелких незанятых блоков.

Статистический анализ показывает, что пропадает в среднем $^1/_3$ памяти! Это известное правило 50% (два соседних свободных участка в отличие от двух соседних процессов могут быть объединены).

Одно из решений проблемы внешней фрагментации — организовать сжатие, т.е. перемещение всех занятых (свободных) участков в сторону возрастания (убывания) адресов, так, чтобы вся свободная память образовала непрерывную область. Этот метод иногда называют схемой с перемещаемыми разделами. В идеале фрагментация после сжатия должна отсутствовать. Сжатие, однако, является дорогостоящей процедурой, алгоритм выбора оптимальной стратегии сжатия очень труден и, как правило, сжатие осуществляется в комбинации с выгрузкой и загрузкой по другим адресам.

8.7. Страничная память

Описанные выше схемы недостаточно эффективно используют память, поэтому в современных схемах управления памятью не принято размещать процесс в оперативной памяти одним непрерывным блоком.

В самом простом и наиболее распространенном случае *страничной организации памяти* (или *paging*) как логическое адресное пространство, так и физическое представляются состоящими из наборов блоков или страниц одинакового размера. При этом образуются *логические страницы* (page), а соответствующие единицы в физической памяти называют физическими страницами или *страничными кадрами* (page frames). Страницы (и страничные кадры) имеют фиксированную длину, обычно являющуюся степенью числа 2, и не могут перекрываться. Каждый кадр содержит одну страницу данных. При такой организации внешняя фрагментация отсутствует, а потери из-за внутренней фрагментации, поскольку процесс занимает целое число страниц, ограничены частью последней страницы процесса.

Логический адрес в страничной системе — упорядоченная пара (p, d), где p — номер страницы в виртуальной памяти, а d — смещение в рамках страницы p, на которой размещается адресуемый элемент. Заметим, что разбиение адресного пространства на страницы осуществляется вычислительной системой незаметно для программиста. Поэтому адрес является двумерным лишь с точки зрения операционной системы, а с точки зрения программиста адресное пространство процесса остается линейным.

Описываемая схема позволяет загрузить процесс, даже если нет непрерывной области кадров, достаточной для размещения процесса целиком. Но одного базового регистра для осуществления трансляции адреса в данной схеме недостаточно. Система отображения логических адресов в физические сводится к системе отображения логических страниц в физические и представляет собой таблицу страниц, которая хранится в оперативной памяти. Иногда говорят, что *таблица страниц* — это кусочно-линейная функция отображения, заданная в табличном виде.

Интерпретация логического адреса показана на рис. 8.7. Если выполняемый процесс обращается к логическому адресу $v = (p,d)$, механизм отображения ищет номер страницы p в таблице страниц и определяет, что эта страница находится в страничном кадре p', формируя реальный адрес из p' и d.

Рис. 8.7. Связь логического и физического адресов при страничной организации памяти

Таблица страниц (page table) адресуется при помощи специального регистра процессора и позволяет определить номер кадра по логическому адресу. Помимо этой основной задачи, при помощи атрибутов, записанных в строке таблицы страниц, можно организовать контроль доступа к конкретной странице и ее защиту.

Отметим еще раз различие точек зрения пользователя и системы на используемую память. С точки зрения пользователя, его память — единое непрерывное пространство, содержащее только одну прог-

рамму. Реальное отображение скрыто от пользователя и контролируется операционной системой. Заметим, что процессу пользователя чужая память недоступна. Он не имеет возможности адресовать память за пределами своей таблицы страниц, которая включает только его собственные страницы.

Для управления физической памятью операционная система поддерживает структуру таблицы кадров. Она имеет одну запись на каждый физический кадр, показывающий его состояние.

Отображение адресов должно быть осуществлено корректно даже в сложных случаях и обычно реализуется аппаратно. Для ссылки на таблицу процессов используется специальный регистр. При переключении процессов необходимо найти таблицу страниц нового процесса, указатель на которую входит в контекст процесса.

8.8. Сегментная и сегментно-страничная организация памяти

Существуют две другие схемы организации управления памятью: сегментная и сегментно-страничная. Сегменты, в отличие от страниц, могут иметь переменный размер. При сегментной организации виртуальный адрес является двумерным как для программиста, так и для операционной системы и состоит из двух полей — номера сегмента и смещения внутри сегмента.

Подчеркнем, что в отличие от страничной организации, где линейный адрес преобразован в двумерный операционной системой для удобства отображения, здесь двумерность адреса является следствием представления пользователя о процессе не в виде линейного массива байтов, а как набор сегментов переменного размера (данные, код, стек...).

Программисты, пишущие на языках низкого уровня, должны иметь представление о сегментной организации, явным образом меняя значения сегментных регистров (это хорошо видно по текстам программ, написанных на Ассемблере). Логическое адресное пространство — набор сегментов. Каждый сегмент имеет имя, размер и другие параметры (уровень привилегий, разрешенные виды обращений, флаги присутствия). В отличие от страничной схемы, где пользователь задает только один адрес, который разбивается на номер страницы и смещение прозрачным для программиста образом, в сегментной схеме пользователь специфицирует каждый адрес двумя величинами: именем сегмента и смещением.

Каждый сегмент — линейная последовательность адресов, начинающаяся с 0. Максимальный размер сегмента определяется разрядностью процессора (при 32-разрядной адресации это 2^{32} байт или 4 Гбайт). Размер сегмента может меняться динамически (например, сегмент стека). В элементе таблицы сегментов помимо физического адреса начала сегмента обычно содержится и длина сегмента. Если размер смещения в виртуальном адресе выходит за пределы размера сегмента, возникает исключительная ситуация.

Логический адрес — упорядоченная пара $v = (s,d)$, номер сегмента и смещение внутри сегмента.

В системах, где сегменты поддерживаются аппаратно, эти параметры обычно хранятся в таблице дескрипторов сегментов, а программа обращается к этим дескрипторам по номерам-селекторам. При этом в контекст каждого процесса входит набор сегментных регистров, содержащих селекторы текущих сегментов кода, стека, данных и т.д. и определяющих, какие сегменты будут использоваться при разных видах обращений к памяти. Это позволяет процессору уже на аппаратном уровне определять допустимость обращений к памяти, упрощая реализацию защиты информации от повреждения и несанкционированного доступа (рис. 8.8).

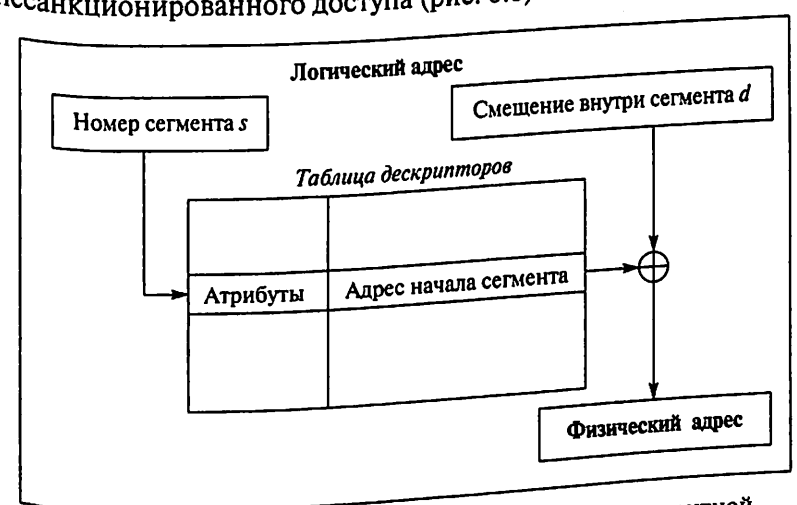

Рис. 8.8. Преобразование логического адреса при сегментной организации памяти

Аппаратная поддержка сегментов распространена мало (главным образом на процессорах Intel). В большинстве операционных систем сегментация реализуется на уровне, не зависящем от аппаратуры.

Хранить в памяти сегменты большого размера целиком так же неудобно, как и хранить процесс непрерывным блоком. Напрашивается идея разбиения сегментов на страницы. При сегментно-страничной организации памяти происходит двухуровневая трансляция виртуального адреса в физический. В этом случае логический адрес состоит из трех полей: номера сегмента логической памяти, номера страницы внутри сегмента и смещения внутри страницы. Соответственно, используются две таблицы отображения — таблица сегментов, связывающая номер сегмента с таблицей страниц, и отдельная таблица страниц для каждого сегмента (рис. 8.9).

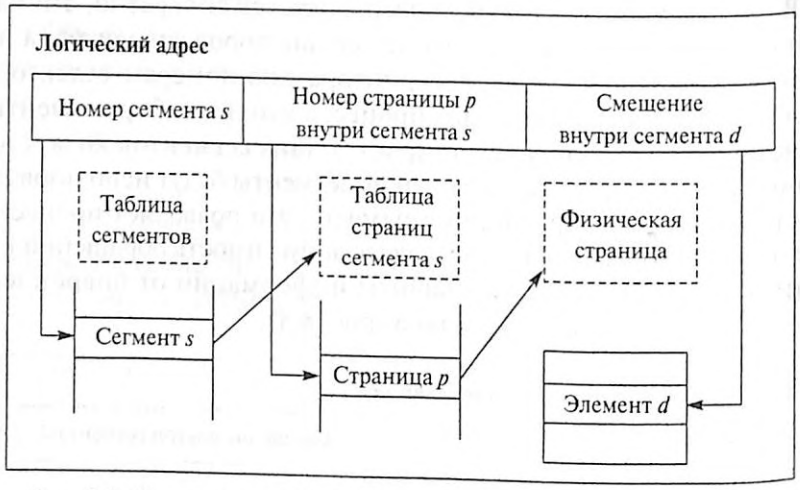

Рис. 8.9. Упрощенная схема формирования физического адреса при сегментно-страничной организации памяти

Сегментно-страничная и страничная организация памяти позволяет легко организовать совместное использование одних и тех же данных и программного кода разными задачами. Для этого различные логические блоки памяти разных процессов отображают в один и тот же блок физической памяти, где размещается разделяемый фрагмент кода или данных.

Глава 9
ВИРТУАЛЬНАЯ ПАМЯТЬ

9.1. Понятие виртуальной памяти

Разработчикам программного обеспечения часто приходится решать проблему размещения в памяти больших программ, размер которых превышает объем доступной оперативной памяти. Один из вариантов решения данной проблемы — организация структур с перекрытием. При этом предполагается активное участие программиста в процессе формирования перекрывающихся частей программы. Развитие архитектуры компьютеров и расширение возможностей операционной системы по управлению памятью позволило переложить решение этой задачи на компьютер. Одним из главных достижений стало появление виртуальной памяти (*virtual memory*). Впервые она была реализована в 1959 г. на компьютере «Атлас», разработанном в Манчестерском университете.

Суть концепции виртуальной памяти заключается в следующем. Информация, с которой работает активный процесс, должна располагаться в оперативной памяти. В схемах виртуальной памяти у процесса создается иллюзия того, что вся необходимая ему информация имеется в основной памяти. Для этого, во-первых, занимаемая процессом память разбивается на несколько частей, например страниц. Во-вторых, логический адрес (логическая страница), к которому обращается процесс, динамически транслируется в физический адрес (физическую страницу). И наконец, в тех случаях, когда страница, к которой обращается процесс, не находится в физической памяти, нужно организовать ее подкачку с диска. Для контроля наличия страницы в памяти вводится специальный бит присутствия, входящий в состав атрибутов страницы в таблице страниц.

Таким образом, в наличии всех компонентов процесса в основной памяти необходимости нет. Важным следствием такой организации является то, что размер памяти, занимаемой процессом, может быть больше, чем размер оперативной памяти. Принцип локальности обеспечивает этой схеме нужную эффективность.

Возможность выполнения программы, находящейся в памяти лишь частично, имеет ряд вполне очевидных преимуществ.

Программа не ограничена объемом физической памяти. Упрощается разработка программ, поскольку можно задействовать большие виртуальные пространства, не заботясь о размере используемой памяти.

Поскольку появляется возможность частичного помещения программы (процесса) в память и гибкого перераспределения памяти между программами, можно разместить в памяти больше программ, что увеличивает загрузку процессора и пропускную способность системы.

Объем ввода-вывода для выгрузки части программы на диск может быть меньше, чем в варианте классического свопинга, в итоге каждая программа будет работать быстрее.

Таким образом, возможность обеспечения (при поддержке операционной системы) для программы «видимости» практически неограниченной (характерный размер для 32-разрядных архитектур 2^{32} = 4 Гбайт) адресуемой пользовательской памяти (логическое адресное пространство) при наличии основной памяти существенно меньших размеров (физическое адресное пространство) — очень важный аспект.

Но введение виртуальной памяти позволяет решать другую, не менее важную задачу — обеспечение контроля доступа к отдельным сегментам памяти и, в частности, защиту пользовательских программ друг от друга и защиту операционной системы от пользовательских программ. Каждый процесс работает со своими виртуальными адресами, трансляцию которых в физические выполняет аппаратура компьютера. Таким образом, пользовательский процесс лишен возможности напрямую обратиться к страницам основной памяти, занятым информацией, относящейся к другим процессам.

Например, 16-разрядный компьютер PDP-11/70 с 64 Кбайт логической памяти мог иметь до 2 Мбайт оперативной памяти. Операционная система этого компьютера тем не менее поддерживала виртуальную память, которая обеспечивала защиту и перераспределение основной памяти между пользовательскими процессами.

Напомним, что в системах с виртуальной памятью те адреса, которые генерирует программа (логические адреса), называются *виртуальными* и они формируют виртуальное адресное пространство. Термин «виртуальная память» означает, что программист имеет дело с памятью, отличной от реальной, размер которой потенциально больше, чем размер оперативной памяти.

Хотя известны и чисто программные реализации виртуальной памяти, это направление получило наиболее широкое развитие после соответствующей аппаратной поддержки.

Следует отметить, что оборудование компьютера принимает участие в трансляции адреса практически во всех схемах управления памятью. Но в случае виртуальной памяти это становится более сложным вследствие разрывности отображения и многомерности логического адресного пространства. Может быть, наиболее существенным вкладом аппаратуры в реализацию описываемой схемы является автоматическая генерация исключительных ситуаций при отсутствии в памяти нужных страниц (page fault).

Любая из трех ранее рассмотренных схем управления памятью — страничной, сегментной и сегментно-страничной — пригодна для организации виртуальной памяти. Чаще всего используется сегментно-страничная модель, которая является синтезом страничной модели и идеи сегментации. Причем для тех архитектур, в которых сегменты не поддерживаются аппаратно, их реализация — задача архитектурно-независимого компонента менеджера памяти.

Сегментная организация в чистом виде встречается редко.

9.2. Архитектурные средства поддержки виртуальной памяти

Очевидно, что невозможно создать полностью машинно-независимый компонент управления виртуальной памятью. С другой стороны, имеются существенные части программного обеспечения, связанного с управлением виртуальной памятью, для которых детали аппаратной реализации совершенно не важны. Одним из достижений современных операционных систем является грамотное и эффективное разделение средств управления виртуальной памятью на аппаратно-независимую и аппаратно-зависимую части. Коротко рассмотрим, что и каким образом входит в аппаратно-зависимую часть подсистемы управления виртуальной памятью.

В самом распространенном случае необходимо отобразить большое виртуальное адресное пространство в физическое адресное пространство существенно меньшего размера. Пользовательский процесс или операционная система должны иметь возможность осуществить запись по виртуальному адресу, а задача операционной системы — сделать так, чтобы записанная информация оказалась в физической памяти (впоследствии при нехватке оперативной памяти она может быть вытеснена во внешнюю память). В случае вир-

туальной памяти система отображения адресных пространств помимо трансляции адресов должна предусматривать ведение таблиц, показывающих, какие области виртуальной памяти в данный момент находятся в физической памяти и где именно размещаются.

Страничная виртуальная память

Как и в случае простой страничной организации, страничная виртуальная память и физическая память представляются состоящими из наборов блоков или страниц одинакового размера. Виртуальные адреса делятся на страницы (page), соответствующие единицы в физической памяти образуют страничные кадры (page frames), а в целом система поддержки страничной виртуальной памяти называется пейджингом (paging). Передача информации между памятью и диском всегда осуществляется целыми страницами.

После разбиения менеджером памяти виртуального адресного пространства на страницы виртуальный адрес преобразуется в упорядоченную пару (p,d), где p — номер страницы в виртуальной памяти, а d — смещение в рамках страницы p, внутри которой размещается адресуемый элемент. Процесс может выполняться, если его текущая страница находится в оперативной памяти. Если текущей страницы в главной памяти нет, она должна быть переписана (подкачана) из внешней памяти. Поступившую страницу можно поместить в любой свободный страничный кадр.

Поскольку число виртуальных страниц велико, таблица страниц принимает специфический вид, структура записей становится более сложной, среди атрибутов страницы появляются биты присутствия, модификации и другие управляющие биты.

При отсутствии страницы в памяти в процессе выполнения команды возникает исключительная ситуация, называемая страничным нарушением (page fault) или страничным отказом. Обработка страничного нарушения заключается в том, что выполнение команды прерывается, затребованная страница подкачивается из конкретного места вторичной памяти в свободный страничный кадр физической памяти и попытка выполнения команды повторяется. При отсутствии свободных страничных кадров на диск выгружается редко используемая страница.

Для управления физической памятью операционная система поддерживает структуру таблицы кадров. Она имеет одну запись на каждый физический кадр, показывающую его состояние.

В большинстве современных компьютеров со страничной организацией в основной памяти хранится лишь часть таблицы страниц,

а быстрота доступа к элементам таблицы текущей виртуальной памяти достигается за счет использования сверхбыстродействующей памяти, размещенной в кэше процессора.

Сегментно-страничная организации виртуальной памяти

Как и в случае простой сегментации, в схемах виртуальной памяти сегмент — это линейная последовательность адресов, начинающаяся с 0. При организации виртуальной памяти размер сегмента может быть велик, например, может превышать размер оперативной памяти. Повторяя все ранее приведенные рассуждения о размещении в памяти больших программ, приходим к разбиению сегментов на страницы и необходимости поддержки своей таблицы страниц для каждого сегмента.

На практике, однако, появления в системе большого количества таблиц страниц стараются избежать, организуя неперекрывающиеся сегменты в одном виртуальном пространстве, для описания которого хватает одной таблицы страниц. Таким образом, одна таблица страниц отводится для всего процесса. Например, в операционных системах Linux и Windows 2000 все сегменты процесса, а также область памяти ядра ограничены виртуальным адресным пространством объемом 4 Гбайт. При этом ядро операционной системы располагается по фиксированным виртуальным адресам вне зависимости от выполняемого процесса.

Структура таблицы страниц

Организация таблицы страниц — один из ключевых элементов отображения адресов в страничной и сегментно-страничной схемах. Рассмотрим структуру таблицы страниц для случая страничной организации более подробно.

Итак, виртуальный адрес состоит из виртуального номера страницы и смещения. Номер записи в таблице страниц соответствует номеру виртуальной страницы. Размер записи колеблется от системы к системе, но чаще всего он составляет 32 бита. Из этой записи в таблице страниц находится номер кадра для данной виртуальной страницы, затем прибавляется смещение и формируется физический адрес. Помимо этого запись в таблице страниц содержит информацию об атрибутах страницы. Это биты присутствия и защиты (например, 0 — read/write, 1 — read only...). Также могут быть указаны: бит модификации, который устанавливается, если содержимое страницы модифицировано, и позволяет контролировать необходимость переписи страницы на диск; бит ссылки, который помогает выделить

малоиспользуемые страницы; бит, разрешающий кэширование, и другие управляющие биты. Заметим, что адреса страниц на диске не являются частью таблицы страниц.

Основную проблему для эффективной реализации таблицы страниц создают большие размеры виртуальных адресных пространств современных компьютеров, которые обычно определяются разрядностью архитектуры процессора. Самыми распространенными на сегодня являются 32-разрядные процессоры, позволяющие создавать виртуальные адресные пространства размером 4 Гбайт (для 64-разрядных компьютеров эта величина равна 2^{64} байт). Кроме того, существует проблема скорости отображения, которая решается за счет использования так называемой ассоциативной памяти.

Подсчитаем примерный размер таблицы страниц. В 32-битном адресном пространстве при размере страницы 4 Кбайт (Intel) получаем $2^{32}/2^{12} = 2^{20}$, т.е. приблизительно миллион страниц, а в 64-битном и того более. Таким образом, таблица должна иметь примерно миллион строк (entry), причем запись в строке состоит из нескольких байтов. Заметим, что каждый процесс нуждается в своей таблице страниц (а в случае сегментно-страничной схемы желательно иметь по одной таблице страниц на каждый сегмент).

Понятно, что количество памяти, отводимое таблицам страниц, не может быть так велико. Для того чтобы избежать размещения в памяти огромной таблицы, ее разбивают на ряд фрагментов. В оперативной памяти хранят лишь некоторые, необходимые для конкретного момента исполнения фрагменты таблицы страниц. В силу свойства локальности число таких фрагментов относительно невелико. Выполнить разбиение таблицы страниц на части можно по-разному. Наиболее распространенный способ разбиения — организация так называемой многоуровневой таблицы страниц. Для примера рассмотрим двухуровневую таблицу с размером страниц 4 Кбайт, реализованную в 32-разрядной архитектуре Intel.

Таблица, состоящая из 2^{20} строк, разбивается на 2^{10} таблиц второго уровня по 2^{10} строк. Эти таблицы второго уровня объединены в общую структуру при помощи одной таблицы первого уровня, состоящей из 2^{10} строк. 32-разрядный адрес делится на 10-разрядное поле p_1, 10-разрядное поле p_2 и 12-разрядное смещение d. Поле p_1 указывает на нужную строку в таблице первого уровня, поле p_2 — второго, а поле d локализует нужный байт внутри указанного страничного кадра (рис. 9.1).

При помощи всего лишь одной таблицы второго уровня можно охватить 4 Мбайт (4 Кбайт × 1024) оперативной памяти. Таким образом, для размещения процесса с большим объемом занимаемой памяти достаточно иметь в оперативной памяти одну таблицу первого уровня и несколько таблиц второго уровня. Очевидно, что суммарное количество строк в этих таблицах много меньше 220. Такой подход естественным образом обобщается на три и более уровней таблицы.

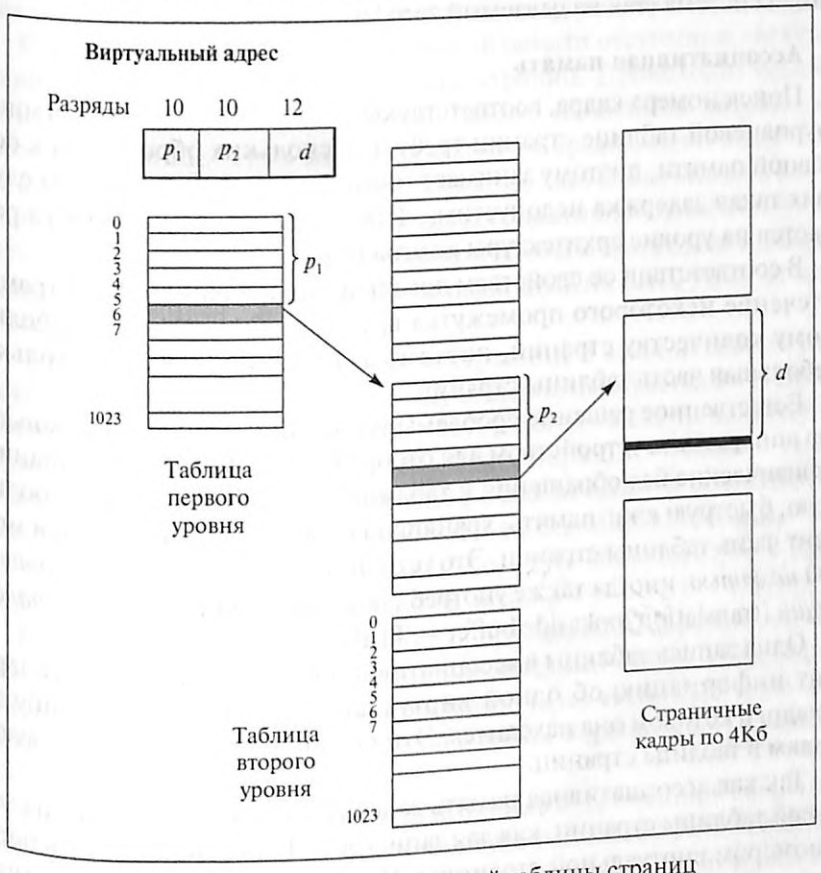

Рис. 9.1. Пример двухуровневой таблицы страниц

Наличие нескольких уровней, естественно, снижает производительность менеджера памяти. Несмотря на то что размеры таблиц на каждом уровне подобраны так, чтобы таблица помещалась целиком внутри одной страницы, обращение к каждому уровню — это отдельное обращение к памяти. Таким образом, трансляция адреса может потребовать нескольких обращений к памяти.

Количество уровней в таблице страниц зависит от конкретных особенностей архитектуры. Можно привести примеры реализации одноуровневого (DEC PDP-11), двухуровневого (Intel, DEC VAX),

трехуровневого (Sun SPARC, DEC Alpha) пейджинга, а также пейджинга с заданным количеством уровней (Motorola). Функционирование RISC-процессора MIPS R2000 осуществляется вообще без таблицы страниц. Здесь поиск нужной страницы, если эта страница отсутствует в ассоциативной памяти, должна взять на себя операционная система (так называемый zero level paging).

Ассоциативная память

Поиск номера кадра, соответствующего нужной странице, в многоуровневой таблице страниц требует нескольких обращений к основной памяти, поэтому занимает много времени. В некоторых случаях такая задержка недопустима. Проблема ускорения поиска решается на уровне архитектуры компьютера.

В соответствии со свойством локальности большинство программ в течение некоторого промежутка времени обращаются к небольшому количеству страниц, поэтому активно используется только небольшая часть таблицы страниц.

Естественное решение проблемы ускорения — снабдить компьютер аппаратным устройством для отображения виртуальных страниц в физические без обращения к таблице страниц, т.е. иметь небольшую, быструю кэш-память, хранящую необходимую на данный момент часть таблицы страниц. Это устройство называется *ассоциативной памятью*, иногда также употребляют термин *буфер поиска трансляции* (translation lookaside buffer — TLB).

Одна запись таблицы в ассоциативной памяти (один вход) содержит информацию об одной виртуальной странице: ее атрибуты и кадр, в котором она находится. Эти поля в точности соответствуют полям в таблице страниц.

Так как ассоциативная память содержит только некоторые из записей таблицы страниц, каждая запись в TLB должна включать поле с номером виртуальной страницы. Память называется ассоциативной, потому что в ней происходит одновременное сравнение номера отображаемой виртуальной страницы с соответствующим полем во всех строках этой небольшой таблицы. Поэтому данный вид памяти достаточно дорого стоит. В строке, поле виртуальной страницы которой совпало с искомым значением, находится номер страничного кадра. Обычное число записей в TLB от 8 до 4096. Рост количества записей в ассоциативной памяти должен осуществляться с учетом таких факторов, как размер кэша основной памяти и количества обращений к памяти при выполнении одной команды.

Рассмотрим функционирование менеджера памяти при наличии ассоциативной памяти.

Вначале информация об отображении виртуальной страницы в физическую отыскивается в ассоциативной памяти. Если нужная запись найдена — все нормально, за исключением случаев нарушения привилегий, когда запрос на обращение к памяти отклоняется.

Если нужная запись в ассоциативной памяти отсутствует, отображение осуществляется через таблицу страниц. Происходит замена одной из записей в ассоциативной памяти найденной записью из таблицы страниц. Здесь мы сталкиваемся с традиционной для любого кэша проблемой замещения (а именно какую из записей в кэше необходимо изменить). Конструкция ассоциативной памяти должна организовывать записи таким образом, чтобы можно было принять решение о том, какая из старых записей должна быть удалена при внесении новых.

Число удачных поисков номера страницы в ассоциативной памяти по отношению к общему числу поисков называется *hit* (совпадение) *ratio* (пропорция, отношение). Иногда также используется термин *«процент попаданий в кэш»*. Таким образом, *hit ratio* — часть ссылок, которая может быть сделана с использованием ассоциативной памяти. Обращение к одним и тем же страницам повышает hit ratio. Чем больше hit ratio, тем меньше среднее время доступа к данным, находящимся в оперативной памяти.

Предположим, например, что для определения адреса в случае кэш-промаха через таблицу страниц необходимо 100 нс, а для определения адреса в случае кэш-попадания через ассоциативную память — 20 нс. С 90% hit ratio среднее время определения адреса — $0{,}9 \cdot 20 + 0{,}1 \cdot 100 = 28$ нс.

Вполне приемлемая производительность современных операционных систем доказывает эффективность использования ассоциативной памяти. Высокое значение вероятности нахождения данных в ассоциативной памяти связано с наличием у данных объективных свойств: пространственной и временной локальности.

Необходимо обратить внимание на следующий факт. При переключении контекста процессов нужно добиться того, чтобы новый процесс «не видел» в ассоциативной памяти информацию, относящуюся к предыдущему процессу, например очищать ее. Таким образом, использование ассоциативной памяти увеличивает время переключения контекста.

Рассмотренная двухуровневая (ассоциативная память + таблица страниц) схема преобразования адреса является ярким примером

иерархии памяти, основанной на использовании принципа локальности.

Инвертированная таблица страниц

Несмотря на многоуровневую организацию, хранение нескольких таблиц страниц большого размера по-прежнему представляет собой проблему. Ее значение особенно актуально для 64-разрядных архитектур, где число виртуальных страниц очень велико. Вариантом решения является применение *инвертированной таблицы страниц* (inverted page table). Этот подход применяется на машинах PowerPC, некоторых рабочих станциях Hewlett-Packard, IBM RT, IBM AS/400 и ряде других.

В этой таблице содержится по одной записи на каждый страничный кадр физической памяти. Существенно, что достаточно одной таблицы для всех процессов. Таким образом, для хранения функции отображения требуется фиксированная часть основной памяти, независимо от разрядности архитектуры, размера и количества процессов. Например, для компьютера Pentium с 256 Мбайт оперативной памяти нужна таблица размером 64 Кбайт строк.

Несмотря на экономию оперативной памяти, применение инвертированной таблицы имеет существенный минус — записи в ней (как и в ассоциативной памяти) не отсортированы по возрастанию номеров виртуальных страниц, что усложняет трансляцию адреса. Один из способов решения данной проблемы — использование хеш-таблицы виртуальных адресов. При этом часть виртуального адреса, представляющая собой номер страницы, отображается в хеш-таблицу с использованием функции хеширования. Каждой странице физической памяти здесь соответствует одна запись в хеш-таблице и инвертированной таблице страниц. Виртуальные адреса, имеющие одно значение хеш-функции, сцепляются друг с другом. Обычно длина цепочки не превышает двух записей.

Размер страницы

Разработчики операционных систем для существующих машин редко имеют возможность влиять на размер страницы. Однако для вновь создаваемых компьютеров решение относительно оптимального размера страницы является актуальным. Как и следовало ожидать, нет одного наилучшего размера. Скорее есть набор факторов, влияющих на размер. Обычно размер страницы — это степень двойки от 2^9 до 2^{14} байт.

Чем больше размер страницы, тем меньше будет размер структур данных, обслуживающих преобразование адресов, но тем больше будут потери, связанные с тем, что память можно выделять только постранично.

Как следует выбирать размер страницы? Во-первых, нужно учитывать размер таблицы страниц, здесь желателен большой размер страницы (страниц меньше, соответственно и таблица страниц меньше). С другой стороны, память лучше утилизируется с маленьким размером страницы. В среднем половина последней страницы процесса пропадает. Необходимо также учитывать объем ввода-вывода для взаимодействия с внешней памятью и другие факторы. Проблема не имеет идеального решения. Историческая тенденция состоит в увеличении размера страницы.

Как правило, размер страниц задается аппаратно, например в DEC PDP-11 — 8 Кбайт, в DEC VAX — 512 байт, в других архитектурах, таких как Motorola 68030, размер страниц может быть задан программно. Учитывая все обстоятельства, в ряде архитектур возникают множественные размеры страниц, например в Pentium размер страницы колеблется от 4 Кбайт до 8 Кбайт. Тем не менее большинство коммерческих операционных систем ввиду сложности перехода на множественный размер страниц поддерживают только один размер страниц.

9.3. Исключительные ситуации при работе с памятью

Для каждой виртуальной страницы запись в таблице страниц содержит номер соответствующего страничного кадра в оперативной памяти, а также атрибуты страницы для контроля обращений к памяти.

Что же происходит, когда нужной страницы в памяти нет или операция обращения к памяти недопустима? Естественно, что операционная система должна быть как-то оповещена о происшедшем. Обычно для этого используется механизм исключительных ситуаций (exceptions). При попытке выполнить подобное обращение к виртуальной странице возникает исключительная ситуация «страничное нарушение» (page fault), приводящая к вызову специальной последовательности команд для обработки конкретного вида страничного нарушения.

Страничное нарушение может происходить в самых разных случаях: при отсутствии страницы в оперативной памяти, при попытке записи в страницу с атрибутом «только чтение» или при

попытке чтения или записи страницы с атрибутом «только выполнение». В любом из этих случаев вызывается обработчик страничного нарушения, являющийся частью операционной системы. Ему обычно передается причина возникновения исключительной ситуации и виртуальный адрес, обращение к которому вызвало нарушение.

Нас будет интересовать конкретный вариант страничного нарушения — обращение к отсутствующей странице, поскольку именно его обработка во многом определяет производительность страничной системы. Когда программа обращается к виртуальной странице, отсутствующей в основной памяти, операционная система должна выделить страницу основной памяти, переместить в нее копию виртуальной страницы из внешней памяти и модифицировать соответствующий элемент таблицы страниц.

Повышение производительности вычислительной системы может быть достигнуто за счет уменьшения частоты страничных нарушений, а также за счет увеличения скорости их обработки. Время эффективного доступа к отсутствующей в оперативной памяти странице складывается из:

- обслуживания исключительной ситуации (page fault);
- чтения (подкачки) страницы из вторичной памяти (иногда, при недостатке места в основной памяти, необходимо вытолкнуть одну из страниц из основной памяти во вторичную, т.е. осуществить замещение страницы);
- возобновления выполнения процесса, вызвавшего данный page fault.

Для решения первой и третьей задач операционная система выполняет до нескольких сот машинных инструкций в течение нескольких десятков микросекунд. Время подкачки страницы близко к нескольким десяткам миллисекунд. Проведенные исследования показывают, что вероятности page fault $5 \cdot 10^{-7}$ оказывается достаточно, чтобы снизить производительность страничной схемы управления памятью на 10%. Таким образом, уменьшение частоты page faults является одной из ключевых задач системы управления памятью. Ее решение обычно связано с правильным выбором алгоритма замещения страниц.

9.4. Стратегии управления страничной памятью

Программное обеспечение подсистемы управления памятью связано с реализацией следующих стратегий:

Стратегия выборки (fetch policy) — в какой момент следует переписать страницу из вторичной памяти в первичную. Существует два основных варианта выборки — по запросу и с упреждением. Алгоритм выборки по запросу вступает в действие в тот момент, когда процесс обращается к отсутствующей странице, содержимое которой находится на диске. Его реализация заключается в загрузке страницы с диска в свободную физическую страницу и коррекции соответствующей записи таблицы страниц.

Алгоритм выборки с упреждением осуществляет опережающее чтение, т.е. кроме страницы, вызвавшей исключительную ситуацию, в память также загружается несколько страниц, окружающих ее (обычно соседние страницы располагаются во внешней памяти последовательно и могут быть считаны за одно обращение к диску). Такой алгоритм призван уменьшить накладные расходы, связанные с большим количеством исключительных ситуаций, возникающих при работе со значительными объемами данных или кода; кроме того, оптимизируется работа с диском.

Стратегия размещения (placement policy) — в какой участок первичной памяти поместить поступающую страницу. В системах со страничной организацией все просто — в любой свободный страничный кадр. В случае систем с сегментной организацией необходима стратегия, аналогичная стратегии с динамическим распределением.

Стратегия замещения (replacement policy) — какую страницу нужно вытолкнуть во внешнюю память, чтобы освободить место в оперативной памяти. Разумная стратегия замещения, реализованная в соответствующем алгоритме замещения страниц, позволяет хранить в памяти самую необходимую информацию и тем самым снизить частоту страничных нарушений. Замещение должно происходить с учетом выделенного каждому процессу количества кадров. Кроме того, нужно решить, должна ли замещаемая страница принадлежать процессу, который инициировал замещение, или она должна быть выбрана среди всех кадров основной памяти.

9.5. Алгоритмы замещения страниц

Итак, наиболее ответственным действием менеджера памяти является выделение кадра оперативной памяти для размещения в ней виртуальной страницы, находящейся во внешней памяти. Напомним, что мы рассматриваем ситуацию, когда размер виртуальной памяти для каждого процесса может существенно превосходить размер основной памяти. Это означает, что при выделении страницы

основной памяти с большой вероятностью не удастся найти свободный страничный кадр. В этом случае операционная система в соответствии с заложенными в нее критериями должна:

- найти некоторую занятую страницу основной памяти;
- переместить в случае надобности ее содержимое во внешнюю память;
- переписать в этот страничный кадр содержимое нужной виртуальной страницы из внешней памяти;
- должным образом модифицировать необходимый элемент соответствующей таблицы страниц;
- продолжить выполнение процесса, которому эта виртуальная страница понадобилась.

Заметим, что при замещении приходится дважды передавать страницу между основной и вторичной памятью. Процесс замещения может быть оптимизирован за счет использования бита модификации (один из атрибутов страницы в таблице страниц). Бит модификации устанавливается компьютером, если хотя бы один байт был записан на страницу. При выборе кандидата на замещение проверяется бит модификации. Если бит не установлен, нет необходимости переписывать данную страницу на диск, ее копия на диске уже имеется. Подобный метод также применяется к read-only-страницам, они никогда не модифицируются. Эта схема уменьшает время обработки page fault.

Существует большое количество разнообразных алгоритмов замещения страниц. Все они делятся на локальные и глобальные. Локальные алгоритмы, в отличие от глобальных, распределяют фиксированное или динамически настраиваемое число страниц для каждого процесса. Когда процесс израсходует все предназначенные ему страницы, система будет удалять из физической памяти одну из его страниц, а не из страниц других процессов. Глобальный же алгоритм замещения в случае возникновения исключительной ситуации удовлетворится освобождением любой физической страницы, независимо от того, какому процессу она принадлежала.

Глобальные алгоритмы имеют ряд недостатков. Во-первых, они делают одни процессы чувствительными к поведению других процессов. Например, если один процесс в системе одновременно использует большое количество страниц памяти, то все остальные приложения будут в результате ощущать сильное замедление из-за недостатка кадров памяти для своей работы. Во-вторых, некорректно работающее приложение может подорвать работу всей системы (если, конечно, в системе не предусмотрено ограничение на размер

памяти, выделяемой процессу), пытаясь захватить больше памяти. Поэтому в многозадачной системе иногда приходится использовать более сложные локальные алгоритмы. Применение локальных алгоритмов требует хранения в операционной системе списка физических кадров, выделенных каждому процессу. Этот список страниц иногда называют *резидентным множеством* процесса.

Эффективность алгоритма обычно оценивается на конкретной последовательности ссылок к памяти, для которой подсчитывается число возникающих page faults. Эта последовательность называется *строкой обращений* (reference string). Мы можем генерировать строку обращений искусственным образом при помощи датчика случайных чисел или трассируя конкретную систему. Последний метод дает слишком много ссылок, для уменьшения числа которых можно сделать две вещи:

- для конкретного размера страниц можно запоминать только их номера, а не адреса, на которые идет ссылка;
- несколько подряд идущих ссылок на одну страницу можно фиксировать один раз.

Как уже говорилось, большинство процессоров имеют простейшие аппаратные средства, позволяющие собирать некоторую статистику обращений к памяти. Эти средства обычно включают два специальных флага на каждый элемент таблицы страниц. *Флаг ссылки* (reference бит) автоматически устанавливается, когда происходит любое обращение к этой странице, а уже рассмотренный выше *флаг изменения* (modify бит) устанавливается, если производится запись в эту страницу. Операционная система периодически проверяет установку таких флагов, для того чтобы выделить активно используемые страницы, после чего значения этих флагов сбрасываются.

Рассмотрим ряд алгоритмов замещения страниц.

Алгоритм FIFO. Выталкивание первой пришедшей страницы

Простейший алгоритм. Каждой странице присваивается временная метка. Реализуется это просто созданием очереди страниц, в конец которой страницы попадают, когда загружаются в физическую память, а из начала берутся, когда требуется освободить память. Для замещения выбирается старейшая страница. К сожалению, эта стратегия с достаточной вероятностью будет приводить к замещению активно используемых страниц, например страниц кода текстового процессора при редактировании файла. Заметим, что при замещении активных страниц все работает корректно, но page fault происходит немедленно.

Аномалия Билэди (Belady)

На первый взгляд кажется очевидным, что чем больше в памяти страничных кадров, тем реже будут иметь место page faults. Удивительно, но это не всегда так. Как установил Билэди с коллегами, определенные последовательности обращений к страницам в действительности приводят к увеличению числа страничных нарушений при увеличении кадров, выделенных процессу. Это явление носит название *«аномалии Билэди»* или *«аномалии FIFO»*.

Система с тремя кадрами (9 faults) оказывается более производительной, чем с четырьмя кадрами (10 faults), для строки обращений к памяти 012301401234 при выборе стратегии FIFO.

	0	1	2	3	0	1	4	0	1	2	3	4
Самая старая страница	0	1	2	3	0	1	4	4	4	2	3	3
		0	1	2	3	0	1	1	1	4	2	2
Самая новая страница			0	1	2	3	0	0	0	1	4	4
	p	p	p	p	p	p	p			p	p	9 page faults

а)

	0	1	2	3	0	1	4	0	1	2	3	4
Самая старая страница	0	1	2	3	3	3	4	0	1	2	3	4
		0	1	2	2	2	3	4	0	1	2	3
			0	1	1	1	2	3	4	0	1	2
Самая новая страница				0	0	0	1	2	3	4	0	1
	p	p	p	p			p	p	p	p	p	p 10 page faults

б)

Рис. 9.2. Аномалия Билэди: FIFO с тремя страничными кадрами (*а*); FIFO с четырьмя страничными кадрами (*б*)

Аномалию Билэди следует считать скорее курьезом, чем фактором, требующим серьезного отношения, который иллюстрирует сложность операционных систем, где интуитивный подход не всегда приемлем.

Оптимальный алгоритм (OPT)

Одним из последствий открытия аномалии Билэди стал поиск оптимального алгоритма, который при заданной строке обращений имел бы минимальную частоту page faults среди всех других алгоритмов. Такой алгоритм был найден. Он прост: замещай страницу, которая не будет использоваться в течение самого длительного периода времени.

Каждая страница должна быть помечена числом инструкций, которые будут выполнены, прежде чем на эту страницу будет сделана первая ссылка. Выталкиваться должна страница, для которой это число наибольшее.

Этот алгоритм легко описать, но реализовать невозможно. Операционная система не знает, к какой странице будет следующее обращение. (Ранее такие проблемы возникали при планировании процессов — алгоритм SJF.)

Зато мы можем сделать вывод, что для того, чтобы алгоритм замещения был максимально близок к идеальному алгоритму, система должна как можно точнее предсказывать обращения процессов к памяти. Данный алгоритм применяется для оценки качества реализуемых алгоритмов.

Выталкивание дольше всего не использовавшейся страницы. Алгоритм LRU

Одним из приближений к алгоритму OPT является алгоритм, исходящий из эвристического правила, что недавнее прошлое — хороший ориентир для прогнозирования ближайшего будущего.

Ключевое отличие между FIFO и оптимальным алгоритмом заключается в том, что один смотрит назад, а другой вперед. Если использовать прошлое для аппроксимации будущего, имеет смысл замещать страницу, которая не использовалась в течение самого долгого времени. Такой подход называется *least recently used алгоритм* (LRU). Работа алгоритма проиллюстрирована на рис. 9.3. Сравнивая рис. 9.2, *б* и 9.3, можно увидеть, что использование LRU-алгоритма позволяет сократить количество страничных нарушений.

	0	1	2	3	0	1	4	0	1	2	3	4
				3	3	3	3	3	3	2	2	2
		2	2	2	2	4	4	4	4	3	3	
	1	1	1	1	1	1	1	1	1	1	1	
0	0	0	0	0	0	0	0	0	0	0	4	
p	p	p	p			p	0		p	p	p	8 page faults

Рис. 9.3. Пример работы алгоритма LRU

LRU — хороший, но труднореализуемый алгоритм. Необходимо иметь связанный список всех страниц в памяти, в начале которого будут храниться недавно использованные страницы. Причем этот список должен обновляться при каждом обращении к памяти. Много времени нужно и на поиск страниц в таком списке.

В [Таненбаум, 2002] рассмотрен вариант реализации алгоритма LRU со специальным 64-битным указателем, который автоматически увеличивается на единицу после выполнения каждой инструкции, а в таблице страниц имеется соответствующее поле, в которое заносится значение указателя при каждой ссылке на страницу. При возникновении page fault выгружается страница с наименьшим значением этого поля.

Как оптимальный алгоритм, так и LRU не страдают от аномалии Билэди. Существует класс алгоритмов, для которых при одной и той же строке обращений множество страниц в памяти для n кадров всегда является подмножеством страниц для $n + 1$ кадра. Эти алгоритмы не проявляют аномалии Билэди и называются *стековыми* (*stack*) *алгоритмами*.

Выталкивание редко используемой страницы. Алгоритм NFU

Поскольку большинство современных процессоров не предоставляют соответствующей аппаратной поддержки для реализации алгоритма LRU, хотелось бы иметь алгоритм, достаточно близкий к LRU, но не требующий специальной поддержки.

Программная реализация алгоритма, близкого к LRU, — *алгоритм NFU* (Not Frequently Used).

Для него требуются программные счетчики, по одному на каждую страницу, которые сначала равны нулю. При каждом прерывании по времени (а не после каждой инструкции) операционная система сканирует все страницы в памяти и у каждой страницы с установленным флагом обращения увеличивает на единицу значение счетчика, а флаг обращения сбрасывает.

Таким образом, кандидатом на освобождение оказывается страница с наименьшим значением счетчика, как страница, к которой реже всего обращались. Главный недостаток алгоритма NFU состоит в том, что он ничего не забывает. Например, страница, к которой очень часто обращались в течение некоторого времени, а потом обращаться перестали, все равно не будет удалена из памяти, потому что ее счетчик содержит большую величину. Например, в многопроходных компиляторах страницы, которые активно использовались

во время первого прохода, могут надолго сохранить большие значения счетчика, мешая загрузке полезных в дальнейшем страниц.

К счастью, возможна небольшая модификация алгоритма, которая позволяет ему «забывать». Достаточно, чтобы при каждом прерывании по времени содержимое счетчика сдвигалось вправо на 1 бит, а уже затем производилось бы его увеличение для страниц с установленным флагом обращения.

Другим, уже более устойчивым недостатком алгоритма является длительность процесса сканирования таблиц страниц.

Другие алгоритмы

Для полноты картины можно упомянуть еще несколько алгоритмов.

Например, алгоритм Second-Chance — модификация алгоритма FIFO, которая позволяет избежать потери часто используемых страниц с помощью анализа флага обращений (бита ссылки) для самой старой страницы. Если флаг установлен, то страница, в отличие от алгоритма FIFO, не выталкивается, а ее флаг сбрасывается, и страница переносится в конец очереди. Если первоначально флаги обращений были установлены для всех страниц (на все страницы ссылались), алгоритм Second-Chance превращается в алгоритм FIFO. Данный алгоритм использовался в Multics и BSD Unix.

В компьютере Macintosh использован алгоритм NRU (Not Recently-Used), где страница-«жертва» выбирается на основе анализа битов модификации и ссылки. Интересные стратегии, основанные на буферизации страниц, реализованы в VAX/VMS и Mach.

Имеется также и много других алгоритмов замещения. Подробное описание различных алгоритмов замещения можно найти в монографиях [Дейтел, 1987], [Цикритис, 1977], [Таненбаум, 2002] и др.

9.6. Управление количеством страниц, выделенным процессу. Модель рабочего множества

В стратегиях замещения прослеживается предположение о том, что количество кадров, принадлежащих процессу, нельзя увеличить. Это приводит к необходимости выталкивания страницы. Рассмотрим более общий подход, базирующийся на концепции рабочего множества, сформулированной Деннингом [Denning, 1996].

Итак, что делать, если в распоряжении процесса имеется недостаточное число кадров? Нужно ли его приостановить с освобожде-

нием всех кадров? Что следует понимать под достаточным количеством кадров?

Трешинг (Thrashing)

Хотя теоретически возможно уменьшить число кадров процесса до минимума, существует какое-то число активно используемых страниц, без которого процесс часто генерирует page faults. Высокая частота страничных нарушений называется *трешинг* (thrashing, иногда употребляется русский термин «пробуксовка», рис. 9.4). Процесс находится в состоянии трешинга, если при его работе больше времени уходит на подкачку страниц, нежели на выполнение команд. Такого рода критическая ситуация возникает вне зависимости от конкретных алгоритмов замещения.

Рис. 9.4. Частота page faults в зависимости от количества кадров, выделенных процессу

Часто результатом трешинга является снижение производительности вычислительной системы. Один из нежелательных сценариев развития событий может выглядеть следующим образом. При глобальном алгоритме замещения процесс, которому не хватает кадров, начинает отбирать кадры у других процессов, которые в свою очередь начинают заниматься тем же. В результате все процессы попадают в очередь запросов к устройству вторичной памяти (находятся в состоянии ожидания), а очередь процессов в состоянии готовности пустеет. Загрузка процессора снижается. Операционная система реагирует на это увеличением степени мультипрограммирования, что приводит к еще большему трешингу и дальнейшему снижению загрузки процессора. Таким образом, пропускная способность системы падает из-за трешинга.

Эффект трешинга, возникающий при использовании глобальных алгоритмов, может быть ограничен за счет применения локальных

алгоритмов замещения. При локальных алгоритмах замещения если даже один из процессов попал в трешинг, это не сказывается на других процессах. Однако он много времени проводит в очереди к устройству выгрузки, затрудняя подкачку страниц остальных процессов.

Критическая ситуация типа трешинга возникает вне зависимости от конкретных алгоритмов замещения. Единственным алгоритмом, теоретически гарантирующим отсутствие трешинга, является рассмотренный выше не реализуемый на практике оптимальный алгоритм.

Итак, *трешинг* — это высокая частота страничных нарушений. Необходимо ее контролировать. Когда она высока, процесс нуждается в кадрах. Можно, устанавливая желаемую частоту page faults, регулировать размер процесса, добавляя или отнимая у него кадры. Может оказаться целесообразным выгрузить процесс целиком. Освободившиеся кадры выделяются другим процессам с высокой частотой page faults.

Для предотвращения трешинга требуется выделять процессу столько кадров, сколько ему нужно. Но как узнать, сколько ему нужно? Необходимо попытаться выяснить, как много кадров процесс реально использует. Для решения этой задачи Деннинг использовал модель рабочего множества, которая основана на применении принципа локальности.

Модель рабочего множества

Рассмотрим поведение реальных процессов.

Процессы начинают работать, не имея в памяти необходимых страниц. В результате при выполнении первой же машинной инструкции возникает page fault, требующий подкачки порции кода. Следующий page fault происходит при локализации глобальных переменных и еще один — при выделении памяти для стека. После того как процесс собрал большую часть необходимых ему страниц, page faults возникают редко.

Таким образом, существует набор страниц (P_1, P_2, ... P_n), активно используюшихся вместе, который позволяет процессу в момент времени t в течение некоторого периода T производительно работать, избегая большого количества page faults. Этот набор страниц называется *рабочим множеством* $W(t, T)$ (*working set*) *процесса*. Число страниц в рабочем множестве определяется параметром T, является невозрастающей функцией T и относительно невелико. Иногда T назы-

вают *размером окна рабочего множества*, через которое ведется наблюдение за процессом (рис. 9.5).

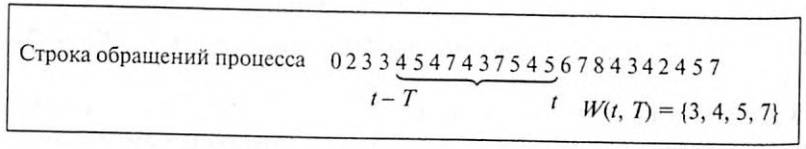

Рис. 9.5. Пример рабочего множества процесса

Легко написать тестовую программу, которая систематически работает с большим диапазоном адресов, но, к счастью, большинство реальных процессов не ведут себя подобным образом, а проявляют свойство локальности. В течение любой фазы вычислений процесс работает с небольшим количеством страниц.

Когда процесс выполняется, он двигается от одного рабочего множества к другому. Программа обычно состоит из нескольких рабочих множеств, которые могут перекрываться. Например, когда вызвана процедура, она определяет новое рабочее множество, состоящее из страниц, содержащих инструкции процедуры, ее локальные и глобальные переменные. После ее завершения процесс покидает это рабочее множество, но может вернуться к нему при новом вызове процедуры. Таким образом, рабочее множество определяется кодом и данными программы. Если процессу выделять меньше кадров, чем ему требуется для поддержки рабочего множества, он будет находиться в состоянии трешинга.

Принцип локальности ссылок препятствует частым изменениям рабочих наборов процессов. Формально это можно выразить следующим образом. Если в период времени $(t\text{-}T, t)$ программа обращалась к страницам $W(t, T)$, то при надлежащем выборе T с большой вероятностью эта программа будет обращаться к тем же страницам в период времени $(t, t + T)$. Другими словами, принцип локальности утверждает, что если не слишком далеко заглядывать в будущее, то можно достаточно точно его прогнозировать исходя из прошлого. Понятно, что с течением времени рабочий набор процесса может изменяться (как по составу страниц, так и по их числу).

Наиболее важное свойство рабочего множества — его размер. Операционная система должна выделить каждому процессу достаточное число кадров, чтобы поместилось его рабочее множество. Если кадры еще остались, то может быть инициирован другой процесс. Если рабочие множества процессов не помещаются в память и начинается трешинг, то один из процессов можно выгрузить на диск.

Решение о размещении процессов в памяти должно, следовательно, базироваться на размере его рабочего множества. Для впервые инициируемых процессов это решение может быть принято эвристически. Во время работы процесса система должна уметь определять: расширяет процесс свое рабочее множество или перемещается на новое рабочее множество. Если в состав атрибутов страницы включить время последнего использования t_i (для страницы с номером i), то принадлежность i-й страницы к рабочему набору, определяемому параметром T в момент времени t будет выражаться неравенством: $t\text{-}T < t_i < t$. Алгоритм выталкивания страниц WSClock, использующий информацию о рабочем наборе процесса, описан в [Таненбаум, 2002].

Другой способ реализации данного подхода может быть основан на отслеживании количества страничных нарушений, вызываемых процессом. Если процесс часто генерирует page faults и память не слишком заполнена, то система может увеличить число выделенных ему кадров. Если же процесс не вызывает исключительных ситуаций в течение некоторого времени и уровень генерации ниже какого-то порога, то число кадров процесса может быть урезано. Этот способ регулирует лишь размер множества страниц, принадлежащих процессу, и должен быть дополнен какой-либо стратегией замещения страниц. Несмотря на то что система при этом может пробуксовывать в моменты перехода от одного рабочего множества к другому, предлагаемое решение в состоянии обеспечить наилучшую производительность для каждого процесса, не требуя никакой дополнительной настройки системы.

9.7. Страничные демоны

Подсистема виртуальной памяти работает производительно при наличии резерва свободных страничных кадров. Алгоритмы, обеспечивающие поддержку системы в состоянии отсутствия трешинга, реализованы в составе фоновых процессов (их часто называют *демонами* или *сервисами*), которые периодически «просыпаются» и инспектируют состояние памяти. Если свободных кадров оказывается мало, они могут сменить стратегию замещения. Их задача — поддерживать систему в состоянии наилучшей производительности.

Примером такого рода процесса может быть фоновый процесс — сборщик страниц, реализующий облегченный вариант алгоритма откачки, основанный на использовании рабочего набора и применяемый во многих клонах операционной системы Unix [Bach, 1986].

Данный демон производит откачку страниц, не входящих в рабочие наборы процессов. Он начинает активно работать, когда количество страниц в списке свободных страниц достигает установленного нижнего порога, и пытается выталкивать страницы в соответствии с собственной стратегией.

Но если возникает требование страницы в условиях, когда список свободных страниц пуст, то начинает работать механизм свопинга, поскольку простое отнятие страницы у любого процесса (включая тот, который затребовал бы страницу) потенциально вело бы к ситуации thrashing и разрушало бы рабочий набор некоторого процесса. Любой процесс, затребовавший страницу не из своего текущего рабочего набора, становится в очередь на выгрузку в расчете на то, что после завершения выгрузки хотя бы одного из процессов свободной памяти уже может быть достаточно.

В операционной системе Windows 2000 аналогичную роль играет менеджер балансного набора (Working set manager), который вызывается раз в секунду или тогда, когда размер свободной памяти опускается ниже определенного предела, и отвечает за суммарную политику управления памятью и поддержку рабочих множеств.

Программная поддержка сегментной модели памяти процесса

Реализация функций операционной системы, связанных с поддержкой памяти, — ведение таблиц страниц, трансляция адреса, обработка страничных ошибок, управление ассоциативной памятью и др. — тесно связана со структурами данных, обеспечивающими удобное представление адресного пространства процесса. Формат этих структур сильно зависит от аппаратуры и особенностей конкретной операционной системы.

Чаще всего виртуальная память процесса операционной системы разбивается на сегменты пяти типов: кода программы, данных, стека, разделяемый сегмент файлов, отображаемых в память (рис. 9.6).

Сегмент программного кода содержит только команды. Сегмент программного кода не модифицируется в ходе выполнения процесса, обычно страницы данного сегмента имеют атрибут read-only. Следствием этого является возможность использования одного экземпляра кода для разных процессов.

Сегмент данных, содержащий переменные программы и сегмент стека, содержащий автоматические переменные, могут динамически менять свой размер (обычно данные в сторону увеличения адресов, а стек — в сторону уменьшения) и содержимое, должны быть доступны по чтению и записи и являются приватными сегментами процесса.

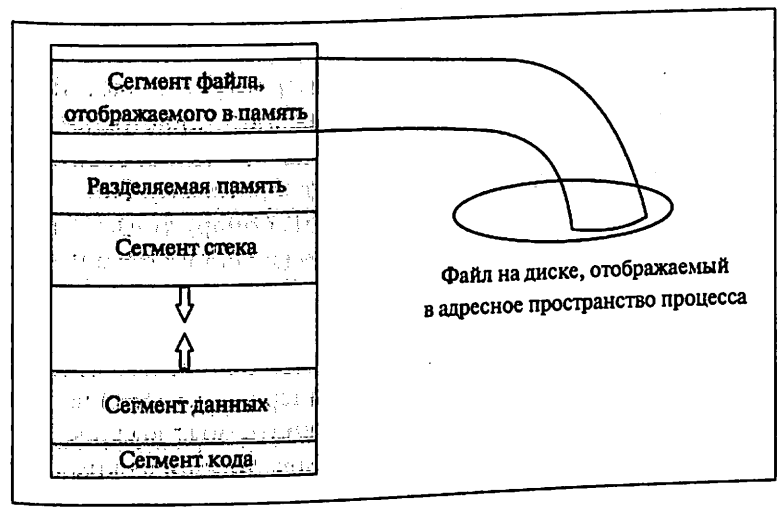

Рис. 9.6. Образ процесса в памяти

С целью обобществления памяти между несколькими процессами создаются *разделяемые сегменты*, допускающие доступ по чтению и записи. Вариантом разделяемого сегмента может быть *сегмент файла, отображаемого в память*. Специфика таких сегментов состоит в том, что из них откачка осуществляется не в системную область выгрузки, а непосредственно в отображаемый файл. Реализация разделяемых сегментов основана на том, что логические страницы различных процессов связываются с одними и теми же страничными кадрами.

Сегменты представляют собой непрерывные области (в Linux они так и называются — *области*) в виртуальном адресном пространстве процесса, выровненные по границам страниц. Каждая область состоит из набора страниц с одним и тем же режимом защиты. Между областями в виртуальном пространстве могут быть свободные участки. Естественно, что подобные объекты описаны соответствующими структурами (например, структуры mm_struct и vm_area_struct в Linux).

Часть работы по организации сегментов может происходить с участием программиста. Особенно это заметно при низкоуровневом программировании. В частности, отдельные области памяти могут быть поименованы и использоваться для обмена данными между процессами. Два процесса могут общаться через разделяемую

область памяти при условии, что им известно ее имя (пароль). Обычно это делается при помощи специальных вызовов (например, map и unmap), входящих в состав интерфейса виртуальной памяти.

Загрузка исполняемого файла (системный вызов exec) осуществляется обычно через отображение (mapping) его частей (кода, данных) в соответствующие сегменты адресного пространства процесса. Например, сегмент кода является сегментом отображаемого в память файла, содержащего исполняемую программу. При попытке выполнить первую же инструкцию система обнаруживает, что нужной части кода в памяти нет, генерирует page fault и подкачивает эту часть кода с диска. Далее процедура повторяется до тех пор, пока вся программа не окажется в оперативной памяти.

Как уже говорилось, размер сегмента данных динамически меняется. Рассмотрим, как организована поддержка сегментов данных в Unix. Пользователь, запрашивая (библиотечные вызовы malloc, new) или освобождая (free, delete) память для динамических данных, фактически изменяет границу выделенной процессу памяти через системный вызов brk (от слова «break»), который модифицирует значение переменной brk из структуры данных процесса. В результате происходит выделение физической памяти, граница brk смещается в сторону увеличения виртуальных адресов, а соответствующие строки таблиц страниц получают осмысленные значения. При помощи того же вызова brk пользователь может уменьшить размер сегмента данных. На практике освобожденная пользователем виртуальная память (библиотечные вызовы free, delete) системе не возвращается. На эт.е. две причины. Во-первых, для уменьшения размеров сегмента данных необходимо организовать его уплотнение или «сборку мусора». А во-вторых, незанятые внутри сегмента данных области естественным образом будут вытолкнуты из оперативной памяти вследствие того, что к ним не будет обращений. Ведение списков занятых и свободных областей памяти в сегменте данных пользователя осуществляется на уровне системных библиотек.

Более подробно информация об адресных пространствах процессов в Unix изложена в [Кузнецов], [Bach, 1986].

9.8. Отдельные аспекты функционирования менеджера памяти

Корректная работа менеджера памяти помимо принципиальных вопросов, связанных с выбором абстрактной модели виртуальной памяти и ее аппаратной поддержкой, обеспечивается также множеством нюансов и мелких деталей. В качестве примера такого рода компонента рассмотрим более подробно *локализацию страниц в памяти*, которая применяется в тех случаях, когда поддержка страничной системы приводит к необходимости разрешить определенным страницам, хранящим буферы ввода-вывода, другие важные данные и код, быть блокированными в памяти.

Рассмотрим случай, когда система виртуальной памяти может вступить в конфликт с подсистемой ввода-вывода. Например, процесс может запросить ввод в буфер и ожидать его завершения. Управление передастся другому процессу, который может вызвать page fault и, с отличной от нуля вероятностью, спровоцировать выгрузку той страницы, куда должен быть осуществлен ввод первым процессом. Подобные ситуации нуждаются в дополнительном контроле, особенно если ввод-вывод реализован с использованием механизма прямого доступа к памяти (DMA). Одно из решений данной проблемы — вводить данные в невытесняемый буфер в пространстве ядра, а затем копировать их в пользовательское пространство.

Второе решение — локализовать страницы в памяти, используя специальный бит локализации, входящий в состав атрибутов страницы. Локализованная страница замещению не подлежит. Бит локализации сбрасывается после завершения операции ввода-вывода.

Другое использование бита локализации может иметь место и при нормальном замещении страниц. Рассмотрим следующую цепь событий. Низкоприоритетный процесс после длительного ожидания получил в свое распоряжение процессор и подкачал с диска нужную ему страницу. Если он сразу после этого будет вытеснен высокоприоритетным процессом, последний может легко заместить вновь подкачанную страницу низкоприоритетного, так как на нее не было ссылок. Имеет смысл вновь загруженные страницы помечать битом локализации до первой ссылки, иначе низкоприоритетный процесс так и не начнет работать.

Использование бита локализации может быть опасным, если забыть его отключить. Если такая ситуация имеет место, страница становится неиспользуемой. SunOS разрешает использование данного бита в качестве подсказки, которую можно игнорировать, когда пул свободных кадров становится слишком маленьким.

Другим важным применением локализации является ее использование в системах мягкого *реального времени*. Рассмотрим процесс или нить реального времени. Вообще говоря, виртуальная память — антитеза вычислений реального времени, так как дает непредсказуемые задержки при подкачке страниц. Поэтому системы реального

времени почти не используют виртуальную память. ОС Solaris поддерживает как реальное время, так и разделение времени. Для решения проблемы page faults, Solaris разрешает процессам сообщать системе, какие страницы важны для процесса, и локализовать их в памяти. В результате возможно выполнение процесса, реализующего задачу реального времени, содержащего локализованные страницы, где временные задержки страничной системы будут минимизированы.

Помимо системы локализации страниц есть и другие интересные проблемы, возникающие в процессе управления памятью. Так, например, бывает непросто осуществить повторное выполнение инструкции, вызвавшей page fault. Представляют интерес и алгоритмы отложенного выделения памяти (копирование при записи и др.).

Глава 10
ФАЙЛОВЫЕ СИСТЕМЫ С ТОЧКИ ЗРЕНИЯ ПОЛЬЗОВАТЕЛЯ

История систем управления данными во внешней памяти начинается еще с магнитных лент, но современный облик они приобрели с появлением магнитных дисков. До этого каждая прикладная программа сама решала проблемы именования данных и их структуризации во внешней памяти. Это затрудняло поддержание на внешнем носителе нескольких архивов долговременно хранящейся информации. Историческим шагом стал переход к использованию централизованных систем управления файлами. Система управления файлами берет на себя распределение внешней памяти, отображение имен файлов в адреса внешней памяти и обеспечение доступа к данным.

Файловая система — это часть операционной системы, назначение которой состоит в том, чтобы организовать эффективную работу с данными, хранящимися во внешней памяти, и обеспечить пользователю удобный интерфейс при работе с такими данными. Организовать хранение информации на магнитном диске непросто. Это требует, например, хорошего знания устройства контроллера диска, особенностей работы с его регистрами. Непосредственное взаимодействие с диском — прерогатива компонента системы ввода-вывода операционной системы, называемого драйвером диска. Для того чтобы избавить пользователя компьютера от сложностей взаимодействия с аппаратурой, была придумана ясная абстрактная модель файловой системы. Операции записи или чтения файла концептуально проще, чем низкоуровневые операции работы с устройствами.

Основная идея использования внешней памяти состоит в следующем. Операционная система делит память на блоки фиксированного размера, например 4096 байт. Файл, обычно представляющий собой неструктурированную последовательность однобайтовых записей, хранящихся в виде последовательности блоков (не обязательно смежных); каждый блок хранит целое число записей. В некоторых операционных системах (MS-DOS) адреса блоков, содержащих данные файла, могут быть организованы в связный список и вынесены в отдельную таблицу в памяти. В других операционных системах (Unix) адреса блоков данных файла хранятся в отдельном блоке внешней памяти (так называемом индексе или индексном

узле). Этот прием, называемый индексацией, является наиболее распространенным для приложений, требующих произвольного доступа к записям файлов. Индекс файла состоит из списка элементов, каждый из которых содержит номер блока в файле и сведения о местоположении данного блока. Считывание очередного байта осуществляется с так называемой *текущей* позиции, которая характеризуется смещением от начала файла. Зная размер блока, легко вычислить номер блока, содержащего текущую позицию. Адрес же нужного блока диска можно затем извлечь из индекса файла. Базовой операцией, выполняемой по отношению к файлу, является чтение блока с диска и перенос его в буфер, находящийся в основной памяти.

Файловая система позволяет при помощи системы справочников (каталогов, директорий) связать уникальное имя файла с блоками вторичной памяти, содержащими данные файла. Иерархическая структура каталогов, используемая для управления файлами, может служить другим примером индексной структуры. В этом случае каталоги или папки играют роль индексов, каждый из которых содержит ссылки на свои подкаталоги. С этой точки зрения вся файловая система компьютера представляет собой большой индексированный файл. Помимо собственно файлов и структур данных, используемых для управления файлами (каталоги, дескрипторы файлов, различные таблицы распределения внешней памяти), понятие «файловая система» включает программные средства, реализующие различные операции над файлами.

Перечислим *основные функции файловой системы*.

1. Идентификация файлов. Связывание имени файла с выделенным ему пространством внешней памяти.
2. Распределение внешней памяти между файлами. Для работы с конкретным файлом пользователю не требуется иметь информацию о местоположении этого файла на внешнем носителе информации. Например, для того чтобы загрузить документ в редактор с жесткого диска, нам не нужно знать, на какой стороне какого магнитного диска, на каком цилиндре и в каком секторе находится данный документ.
3. Обеспечение надежности и отказоустойчивости. Стоимость информации может во много раз превышать стоимость компьютера.
4. Обеспечение защиты от несанкционированного доступа.
5. Обеспечение совместного доступа к файлам, так чтобы пользователю не приходилось прилагать специальных усилий по обеспечению синхронизации доступа.

6. Обеспечение высокой производительности.

Иногда говорят, что *файл* — это поименованный набор связанной информации, записанной во вторичную память. Для большинства пользователей файловая система — наиболее видимая часть операционной системы. Она предоставляет механизм для онлайнового хранения и доступа как к данным, так и к программам для всех пользователей системы. С точки зрения пользователя, *файл* — единица внешней памяти, т.е. данные, записанные на диск, должны быть в составе какого-нибудь файла.

Важный аспект организации файловой системы — учет стоимости операций взаимодействия с вторичной памятью. Процесс считывания блока диска состоит из позиционирования считывающей головки над дорожкой, содержащей требуемый блок, ожидания, пока требуемый блок сделает оборот и окажется под головкой, и собственно считывания блока. Для этого требуется значительное время (десятки миллисекунд). В современных компьютерах обращение к диску осуществляется примерно в 100 000 раз медленнее, чем обращение к оперативной памяти. Таким образом, критерием вычислительной сложности алгоритмов, работающих с внешней памятью, является количество обращений к диску.

10.1. Общие сведения о файлах

Имена файлов

Файлы представляют собой абстрактные объекты. Их задача — хранить информацию, скрывая от пользователя детали работы с устройствами. Когда процесс создает файл, он дает ему имя. После завершения процесса файл продолжает существовать и через свое имя может быть доступен другим процессам.

Правила именования файлов зависят от операционной системы. Многие операционные системы поддерживают имена из *двух частей* (имя + расширение), например progr.c (файл, содержащий текст программы на языке Си) или autoexec.bat (файл, содержащий команды интерпретатора командного языка). Тип расширения файла позволяет операционной системе организовать работу с ним различных прикладных программ в соответствии с заранее оговоренными соглашениями. Обычно операционные системы накладывают некоторые ограничения, как на используемые в имени символы, так и на длину имени файла. В соответствии со стандартом POSIX, популяр-

ные операционные системы оперируют удобными для пользователя длинными именами (до 255 символов).

Типы файлов

Важный аспект организации файловой системы и операционной системы — следует ли поддерживать и распознавать типы файлов. Если да, то это может помочь правильному функционированию операционной системы, например не допустить вывода на принтер бинарного файла.

Основные типы файлов: регулярные (обычные) файлы и директории (справочники, каталоги). Обычные файлы содержат пользовательскую информацию. *Директории* — системные файлы, поддерживающие структуру файловой системы. В каталоге содержится перечень входящих в него файлов и устанавливается соответствие между файлами и их характеристиками (атрибутами).

Напомним, что хотя внутри подсистемы управления файлами обычный файл представляется в виде набора блоков внешней памяти, для пользователей обеспечивается представление файла в виде линейной последовательности байтов. Такое представление позволяет использовать абстракцию файла при работе с внешними устройствами, при организации межпроцессных взаимодействий и т.д. Так, например, клавиатура обычно рассматривается как текстовый файл, из которого компьютер получает данные в символьном формате. Поэтому иногда к файлам приписывают другие объекты операционной системы, например специальные символьные файлы и специальные блочные файлы, именованные каналы и сокеты, имеющие файловый интерфейс.

Далее речь пойдет главным образом об обычных файлах.

Обычные (или регулярные) *файлы* реально представляют собой набор блоков (возможно, пустой) на устройстве внешней памяти, на котором поддерживается файловая система. Такие файлы могут содержать как текстовую информацию (обычно в формате ASCII), так и произвольную двоичную (бинарную) информацию.

Текстовые файлы содержат символьные строки, которые можно распечатать, увидеть на экране или редактировать обычным текстовым редактором.

Другой тип файлов — нетекстовые, или бинарные, файлы. Обычно они имеют некоторую внутреннюю структуру. Например, исполняемый файл в операционной системе Unix имеет пять секций: заголовок, текст, данные, биты реаллокации и символьную таблицу. Операционная система выполняет файл, только если он имеет нужный формат. Другим примером бинарного файла может быть архивный файл. Типизация файлов не слишком строгая.

Обычно прикладные программы, работающие с файлами, распознают тип файла по его имени в соответствии с общепринятыми соглашениями. Например, файлы с расширениями .c, .pas, .txt — ASCII-файлы, файлы с расширениями .exe — исполняемые, файлы с расширениями .obj, .zip — бинарные и т.д.

Атрибуты файлов

Кроме имени операционные системы часто связывают с каждым файлом и другую информацию, например дату модификации, размер и т.д. Эти другие характеристики файлов называются *атрибутами*. Список атрибутов в разных операционных системах может варьироваться. Обычно он содержит следующие элементы: основную информацию (имя, тип файла), адресную информацию (устройство, начальный адрес, размер), информацию об управлении доступом (владелец, допустимые операции) и информацию об использовании (даты создания, последнего чтения, модификации и др.).

Список атрибутов обычно хранится в структуре директорий или других структурах, обеспечивающих доступ к данным файла.

10.2. Организация файлов и доступ к ним

Программист воспринимает файл в виде набора однородных записей. *Запись* — это наименьший элемент данных, который может быть обработан как единое целое прикладной программой при обмене с внешним устройством. Причем в большинстве операционных систем размер записи равен одному байту. В то время как приложения оперируют записями, физический обмен с устройством осуществляется большими единицами (обычно блоками). Поэтому записи объединяются в блоки для вывода и разблокируются — для ввода. Операционные системы поддерживают несколько вариантов структуризации файлов.

Последовательный файл

Простейший вариант — так называемый последовательный файл. Т.е. файл является последовательностью записей. Поскольку записи, как правило, однобайтовые, файл представляет собой *неструктурированную последовательность байтов*.

Обработка подобных файлов предполагает последовательное чтение записей от начала файла, причем конкретная запись определя-

ется ее положением в файле. Такой способ доступа называется *последовательным* (модель ленты). Если в качестве носителя файла используется магнитная лента, то так и делается. Текущая позиция считывания может быть возвращена к началу файла (rewind).

Файл прямого доступа

В реальной практике файлы хранятся на устройствах прямого (random) доступа, например на дисках, поэтому содержимое файла может быть разбросано по разным блокам диска, которые можно считывать в произвольном порядке. Причем номер блока однозначно определяется позицией внутри файла.

Здесь имеется в виду относительный номер, специфицирующий данный блок среди блоков диска, принадлежащих файлу.

Естественно, что в этом случае для доступа к середине файла просмотр всего файла с самого начала не обязателен. Для специфицирования места, с которого надо начинать чтение, используются два способа: с начала или с текущей позиции, которую дает операция seek. Файл, байты которого могут быть считаны в произвольном порядке, называется *файлом прямого доступа*.

Таким образом, файл, состоящий из однобайтовых записей на устройстве прямого доступа, — наиболее распространенный способ организации файла. Базовыми операциями для такого рода файлов являются считывание или запись символа в текущую позицию. В большинстве языков высокого уровня предусмотрены операторы посимвольной пересылки данных в файл или из него.

Подобную логическую структуру имеют файлы во многих файловых системах, например в файловых системах операционных систем Unix и MS-DOS. Операционная система не осуществляет никакой интерпретации содержимого файла. Эта схема обеспечивает максимальную гибкость и универсальность. С помощью базовых системных вызовов (или функций библиотеки ввода/вывода) пользователи могут как угодно структурировать файлы. В частности, многие СУБД хранят свои базы данных в обычных файлах.

Другие формы организации файлов

Известны как другие формы организации файла, так и другие способы доступа к ним, которые использовались в ранних операционных системах, а также применяются сегодня в больших мэйнфреймах (mainframe), ориентированных на коммерческую обработку данных.

Первый шаг в структурировании — хранение файла в виде *последовательности записей фиксированной длины*, каждая из которых имеет внутреннюю структуру. Операция чтения производится над записью, а операция записи переписывает или добавляет запись целиком. Ранее использовались записи по 80 байт (это соответствовало числу позиций в перфокарте) или по 132 символа (ширина принтера). В операционной система CP/M файлы были последовательностями 128-символьных записей. С введением CRT-терминалов данная идея утратила популярность.

Другой способ представления файлов — *последовательность записей переменной длины*, каждая из которых содержит ключевое поле в фиксированной позиции внутри записи (рис. 10.1). Базисная операция в данном случае — считать запись с каким-либо значением ключа. Записи могут располагаться в файле последовательно (например, отсортированные по значению ключевого поля) или в более сложном порядке. Метод доступа по значению ключевого поля к записям последовательного файла называется *индексно-последовательным*.

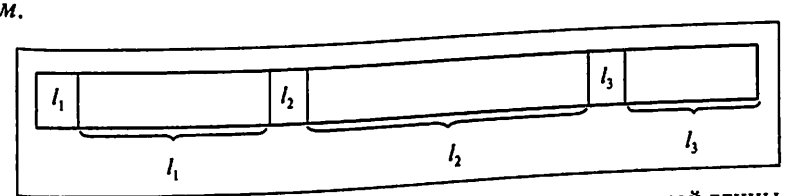

Рис. 10.1. Файл как последовательность записей переменной длины

В некоторых системах ускорение доступа к файлу обеспечивается конструированием *индекса* файла. Индекс обычно хранится на том же устройстве, что и сам файл, и состоит из списка элементов, каждый из которых содержит идентификатор записи, за которым следует указание о местоположении данной записи. Для поиска записи вначале происходит обращение к индексу, где находится указатель на нужную запись. Такие файлы называются *индексированными*, а метод доступа к ним — *доступ с использованием индекса*.

Предположим, у нас имеется большой несортированный файл, содержащий разнообразные сведения о студентах, состоящие из записей с несколькими полями, и возникает задача организации быстрого поиска по одному из полей, например по фамилии студента. Рисунок 10.2 иллюстрирует решение данной проблемы — организацию метода доступа к файлу с использованием индекса.

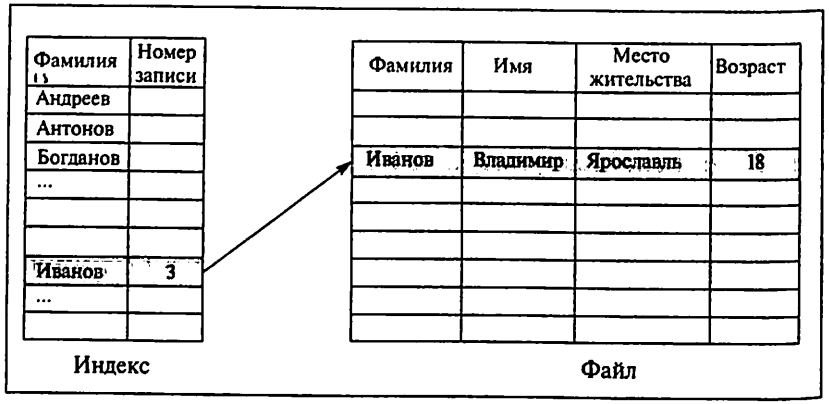

Рис. 10.2. Пример организации индекса для последовательного файла

Следует отметить, что почти всегда главным фактором увеличения скорости доступа является *избыточность* данных.

Способ выделения дискового пространства при помощи индексных узлов, применяемый в ряде операционных систем (Unix и некоторых других), может служить другим примером организации индекса.

В этом случае операционная система использует древовидную организацию блоков, при которой блоки, составляющие файл, являются листьями дерева, а каждый внутренний узел содержит указатели на множество блоков файла. Для больших файлов индекс может быть слишком велик. В этом случае создают индекс для индексного файла (блоки промежуточного уровня или блоки косвенной адресации).

10.3. Операции над файлами

Операционная система должна предоставить в распоряжение пользователя набор операций для работы с файлами, реализованных через системные вызовы. Чаще всего при работе с файлом пользователь выполняет не одну, а несколько операций. Во-первых, нужно найти данные файла и его атрибуты по символьному имени, во-вторых, считать необходимые атрибуты файла в отведенную область оперативной памяти и проанализировать права пользователя на выполнение требуемой операции. Затем следует выполнить операцию, после чего освободить занимаемую данными файла область памяти. Рассмотрим в качестве примера основные файловые операции операционной системы Unix [Таненбаум, 2002].

- Создание файла, не содержащего данных. Смысл данного вызова — объявить, что файл существует, и присвоить ему ряд атрибутов. При этом выделяется место для файла на диске и вносится запись в каталог.
- Удаление файла и освобождение занимаемого им дискового пространства.
- Открытие файла. Перед использованием файла процесс должен его открыть. Цель данного системного вызова — разрешить системе проанализировать атрибуты файла и проверить права доступа к нему, а также считать в оперативную память список адресов блоков файла для быстрого доступа к его данным. Открытие файла является процедурой создания *дескриптора* или управляющего блока файла. Дескриптор (описатель) файла хранит всю информацию о нем. Иногда, в соответствии с парадигмой, принятой в языках программирования, под дескриптором понимается альтернативное имя файла или указатель на описание файла в таблице открытых файлов, используемый при последующей работе с файлом. Например, на языке Си операция открытия файла *fd = open (pathname, flags, modes)*; возвращает дескриптор *fd*, который может быть задействован при выполнении операций чтения (*read (fd, buffer ,count)*;) или записи.
- Закрытие файла. Если работа с файлом завершена, его атрибуты и адреса блоков на диске больше не нужны. В этом случае файл нужно закрыть, чтобы освободить место во внутренних таблицах файловой системы.
- Позиционирование. Дает возможность специфицировать место внутри файла, откуда будет производиться считывание (или запись) данных, т.е. задать *текущую* позицию.
- Чтение данных из файла. Обычно это делается с текущей позиции. Пользователь должен задать объем считываемых данных и предоставить для них буфер в оперативной памяти.
- Запись данных в файл с текущей позиции. Если текущая позиция находится в конце файла, его размер увеличивается, в противном случае запись осуществляется на место имеющихся данных, которые, таким образом, теряются.

Есть и другие операции, например переименование файла, получение атрибутов файла и т.д.

Существует два способа выполнить последовательность действий над файлами [Олифер, 2001].

В первом случае для каждой операции выполняются как универсальные, так и уникальные действия (схема *stateless*). Например, по-

следовательность операций может быть такой: *open, read*1*, close, ... open*$_n$*, read*2*, close, ... open, read*3*, close.*

Альтернативный способ — это когда универсальные действия выполняются в начале и в конце последовательности операций, а для каждой промежуточной операции выполняются только уникальные действия. В этом случае последовательность вышеприведенных операций будет выглядеть так: *open, read*1*, ... read*2*, ... read*3*, close.*

Большинство операционных систем использует второй способ, более экономичный и быстрый. Первый способ более устойчив к сбоям, поскольку результаты каждой операции становятся независимыми от результатов предыдущей операции; поэтому он иногда применяется в распределенных файловых системах (например, Sun NFS).

10.4. Директории. Логическая структура файлового архива

Количество файлов на компьютере может быть большим. Отдельные системы хранят тысячи файлов, занимающие сотни гигабайтов дискового пространства. Эффективное управление этими данными подразумевает наличие в них четкой логической структуры. Все современные файловые системы поддерживают многоуровневое именование файлов за счет наличия во внешней памяти дополнительных файлов со специальной структурой — каталогов (или директорий).

Каждый каталог содержит список каталогов и/или файлов, содержащихся в данном каталоге. Каталоги имеют один и тот же внутренний формат, где каждому файлу соответствует одна запись в файле директории (рис. 10.3).

Имя файла (каталога)	Тип файла (обычный или каталог)	
Anti	K	Атрибуты
Games	K	Атрибуты
Autoexec.bat	O	Атрибуты
Mouse.com	O	Атрибуты

Рис. 10.3. Директории

Число директорий зависит от системы. В ранних операционных системах имелась только одна корневая директория, затем появились

директории для пользователей (по одной директории на пользователя). В современных операционных системах используется произвольная структура дерева директорий.

Таким образом, файлы на диске образуют иерархическую древовидную структуру (рис. 10.4).

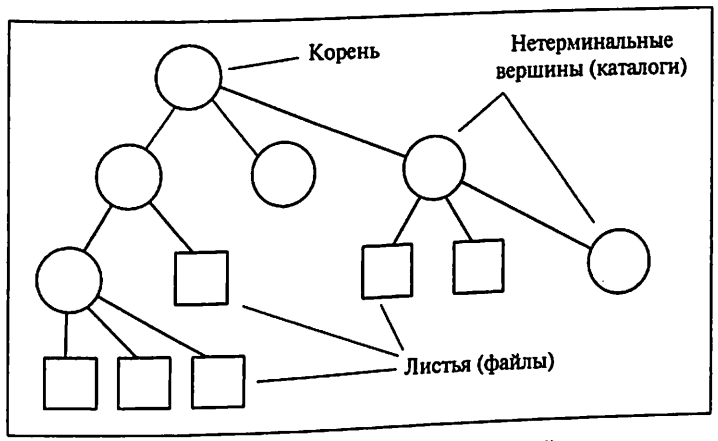

Рис. 10.4. Древовидная структура файловой системы

Существует несколько эквивалентных способов изображения дерева. Структура перевернутого дерева, приведенного на рис. 10.4, наиболее распространена. Верхнюю вершину называют корнем. Если элемент дерева не может иметь потомков, он называется терминальной вершиной или листом (в данном случае является файлом). Нелистовые вершины — справочники или каталоги — содержат списки листовых и нелистовых вершин. Путь от корня к файлу *однозначно* определяет файл.

Подобные древовидные структуры являются графами, не имеющими циклов. Можно считать, что ребра графа направлены вниз, а корень — вершина, не имеющая входящих ребер. Связывание файлов, которое практикуется в ряде операционных систем, приводит к образованию циклов в графе.

Внутри одного каталога имена листовых файлов уникальны. Имена файлов, находящихся в разных каталогах, могут совпадать. Для того чтобы однозначно определить файл по его имени (избежать коллизии имен), принято именовать файл так называемым *абсолютным или полным именем* (*pathname*), состоящим из списка имен вложенных каталогов, по которому можно найти путь от корня к файлу плюс имя файла в каталоге, непосредственно содержащем данный файл. Т.е. полное имя включает цепочку имен — путь к файлу, на-

пример /usr/games/doom. Такие имена уникальны. Компоненты пути разделяют различными символами: «/» (слэш) в Unix или обратными слэшем в MS-DOS (в Multics — «>»). Таким образом, использование древовидных каталогов минимизирует сложность назначения уникальных имен.

Указывать полное имя не всегда удобно, поэтому применяют другой способ задания имени — *относительный* путь к файлу. Он использует концепцию рабочей или текущей директории, которая обычно входит в состав атрибутов процесса, работающего с данным файлом. Тогда на файлы в такой директории можно ссылаться только по имени, при этом поиск файла будет осуществляться в рабочем каталоге. Это удобнее, но, по существу, то же самое, что и абсолютная форма.

Для получения доступа к файлу и локализации его блоков система должна выполнить навигацию по каталогам. Рассмотрим для примера путь /usr/linux/progr.c. Алгоритм одинаков для всех иерархических систем. Сначала в фиксированном месте на диске находится корневая директория. Затем находится компонент пути usr, т.е. в корневой директории ищется файл /usr. Исследуя этот файл, система понимает, что данный файл является каталогом, и блоки его данных рассматривает как список файлов и ищет следующий компонент linux в нем. Из строки для linux находится файл, соответствующий компоненту usr/linux/. Затем находится компонент progr.c, который открывается, заносится в таблицу открытых файлов и сохраняется в ней до закрытия файла.

Отклонение от типовой обработки компонентов pathname может возникнуть в том случае, когда этот компонент является не обычным каталогом с соответствующим ему индексным узлом и списком файлов, а служит точкой связывания (принято говорить «точкой монтирования») двух файловых архивов.

Многие прикладные программы работают с файлами, находящимися в текущей директории, не указывая явным образом ее имени. Это дает пользователю возможность произвольным образом именовать каталоги, содержащие различные программные пакеты. Для реализации этой возможности в большинстве операционных систем, поддерживающих иерархическую структуру директорий, используется обозначение «.» — для текущей директории и «..» — для родительской.

Разделы диска. Организация доступа к архиву файлов

Задание пути к файлу в файловых системах некоторых операционных систем отличается тем, с чего начинается эта цепочка имен.

В современных операционных системах принято разбивать диски на *логические диски* (это низкоуровневая операция), иногда называемые *разделами* (partitions). Бывает, что, наоборот, объединяют несколько физических дисков в один логический диск (например, это можно сделать в операционной системе Windows NT). Поэтому в дальнейшем изложении мы будем игнорировать проблему физического выделения пространства для файлов и считать, что каждый раздел представляет собой отдельный (виртуальный) диск. Диск содержит иерархическую древовидную структуру, состоящую из набора файлов, каждый из которых является хранилищем данных пользователя, и каталогов или директорий (т.е. файлов, которые содержат перечень других файлов, входящих в состав каталога), необходимых для хранения информации о файлах системы.

В некоторых системах управления файлами требуется, чтобы каждый архив файлов целиком располагался на одном диске (разделе диска). В этом случае полное имя файла начинается с имени дискового устройства, на котором установлен соответствующий диск (буквы диска). Например, c:\util\nu\ndd.exe. Такой способ именования используется в файловых системах DEC и Microsoft.

В других системах (Multics) вся совокупность файлов и каталогов представляет собой единое дерево. Сама система, выполняя поиск файлов по имени, начиная с корня, требовала установки необходимых дисков.

В операционной системе Unix предполагается наличие нескольких архивов файлов, каждый на своем разделе, один из которых считается корневым. После запуска системы можно *«смонтировать»* корневую файловую систему и ряд изолированных файловых систем в одну общую файловую систему.

Технически это осуществляется с помощью создания в корневой файловой системе специальных пустых каталогов. Специальный системный вызов *mount* операционной системы Unix позволяет подключить к одному из этих пустых каталогов корневой каталог указанного архива файлов. После монтирования общей файловой системы именование файлов производится так же, как если бы она с самого начала была централизованной. Задачей операционной системы является беспрепятственный проход точки монтирования при получении доступа к файлу по цепочке имен. Если учесть, что обычно монтирование файловой системы производится при загрузке системы, пользователи операционной системы Unix обычно и не задумываются о происхождении общей файловой системы.

10.5. Операции над директориями

Как и в случае с файлами, система обязана обеспечить пользователя набором операций, необходимых для работы с директориями, реализованных через системные вызовы. Несмотря на то что директории — это файлы, логика работы с ними отличается от логики работы с обычными файлами и определяется природой этих объектов, предназначенных для поддержки структуры файлового архива. Совокупность системных вызовов для управления директориями зависит от особенностей конкретной операционной системы. Напомним, что операции над каталогами являются прерогативой операционной системы, т.е. пользователь не может, например, выполнить запись в каталог начиная с текущей позиции. Рассмотрим в качестве примера некоторые системные вызовы, необходимые для работы с каталогами [Таненбаум, 2002].

- Создание директории. Вновь созданная директория включает записи с именами '.' и '..', однако считается пустой.
- Удаление директории. Удалена может быть только пустая директория.
- Открытие директории для последующего чтения. Например, чтобы перечислить файлы, входящие в директорию, процесс должен открыть директорию и считать имена всех файлов, которые она включает.
- Закрытие директории после ее чтения для освобождения места во внутренних системных таблицах.
- Поиск. Данный системный вызов возвращает содержимое текущей записи в открытой директории. Вообще говоря, для этих целей может использоваться системный вызов *Read*, но в этом случае от программиста потребуется знание внутренней структуры директории.
- Получение списка файлов в каталоге.
- Переименование. Имена директорий можно менять, как и имена файлов.
- Создание файла. При создании нового файла необходимо добавить в каталог соответствующий элемент.
- Удаление файла. Удаление из каталога соответствующего элемента. Если удаляемый файл присутствует только в одной директории, то он вообще удаляется из файловой системы, в противном случае система ограничивается только удалением специфицируемой записи.

Очевидно, что создание и удаление файлов предполагает также выполнение соответствующих файловых операций. Имеется еще ряд других системных вызовов, например связанных с защитой информации.

10.6. Защита файлов

Информация в компьютерной системе должна быть защищена как от физического *разрушения* (reliability), так и от несанкционированного *доступа* (protection).

Здесь мы коснемся отдельных аспектов защиты, связанных с контролем доступа к файлам.

Контроль доступа к файлам

Наличие в системе многих пользователей предполагает организацию контролируемого доступа к файлам. Выполнение любой операции над файлом должно быть разрешено только в случае наличия у пользователя соответствующих привилегий. Обычно контролируются следующие операции: чтение, запись и выполнение. Другие операции, например копирование файлов или их переименование, также могут контролироваться. Однако они чаще реализуются через перечисленные. Так, операцию копирования файлов можно представить как операцию чтения и последующую операцию записи.

Списки прав доступа

Наиболее общий подход к защите файлов от несанкционированного использования — сделать доступ зависящим от идентификатора пользователя, т.е. связать с каждым файлом или директорией *список прав доступа* (access control list), где перечислены имена пользователей и типы разрешенных для них способов доступа к файлу. Любой запрос на выполнение операции сверяется с таким списком. Основная проблема реализации данного способа — список может быть длинным. Чтобы разрешить всем пользователям читать файл, необходимо всех их внести в список. У такой техники есть два нежелательных следствия.

- Конструирование подобного списка может оказаться сложной задачей, особенно если мы не знаем заранее пользователей системы.
- Запись в директории должна иметь переменный размер (включать список потенциальных пользователей).

Для решения этих проблем создают классификации пользователей, например, в операционной системе Unix все пользователи разделены на три группы.

- Владелец (Owner).
- Группа (Group). Набор пользователей, разделяющих файл и нуждающихся в типовом способе доступа к нему.
- Остальные (Univers).

Это позволяет реализовать конденсированную версию списка прав доступа. В рамках такой ограниченной классификации задаются только три поля (по одному для каждой группы) для каждой контролируемой операции. В итоге в Unix операции чтения, записи и исполнения контролируются при помощи 9 бит (rwxrwxrwx).

Глава 11
РЕАЛИЗАЦИЯ ФАЙЛОВОЙ СИСТЕМЫ И ДИРЕКТОРИЙ

Как уже говорилось, файловая система должна организовать эффективную работу с данными, хранящимися во внешней памяти, и предоставить пользователю возможности для запоминания и выборки этих данных.

Для организации хранения информации на диске пользователь вначале обычно выполняет его форматирование, выделяя на нем место для структур данных, которые описывают состояние файловой системы в целом. Затем пользователь создает нужную ему структуру каталогов (или директорий), которые, по существу, являются списками вложенных каталогов и собственно файлов. И наконец, он заполняет дисковое пространство файлами, приписывая их тому или иному каталогу. Таким образом, операционная система должна предоставить в распоряжение пользователя совокупность системных вызовов, которые обеспечивают его необходимыми сервисами.

Кроме того, файловые службы могут решать проблемы проверки и сохранения целостности файловой системы, проблемы повышения производительности и ряд других.

11.1. Общая структура файловой системы

Система хранения данных на дисках может быть структурирована следующим образом (рис. 11.1).

Нижний уровень — оборудование. Это в первую очередь магнитные диски с подвижными головками — основные устройства внешней памяти, представляющие собой пакеты магнитных пластин (поверхностей), между которыми на одном рычаге двигается пакет магнитных головок. Шаг движения пакета головок является дискретным, и каждому положению пакета головок логически соответствует цилиндр магнитного диска. Цилиндры делятся на дорожки (треки), а каждая дорожка размечается на одно и то же количество блоков (секторов) таким образом, что в каждый блок можно записать по максимуму одно и то же число байтов. Следовательно, для обмена с магнитным диском на уровне аппаратуры нужно указать номер цилиндра, номер поверхности, номер блока на соответствующей дорожке и число байтов, которое нужно записать или прочитать от на-

чала этого блока. Таким образом, диски могут быть разбиты на блоки фиксированного размера и можно непосредственно получить доступ к любому блоку (организовать прямой доступ к файлам).

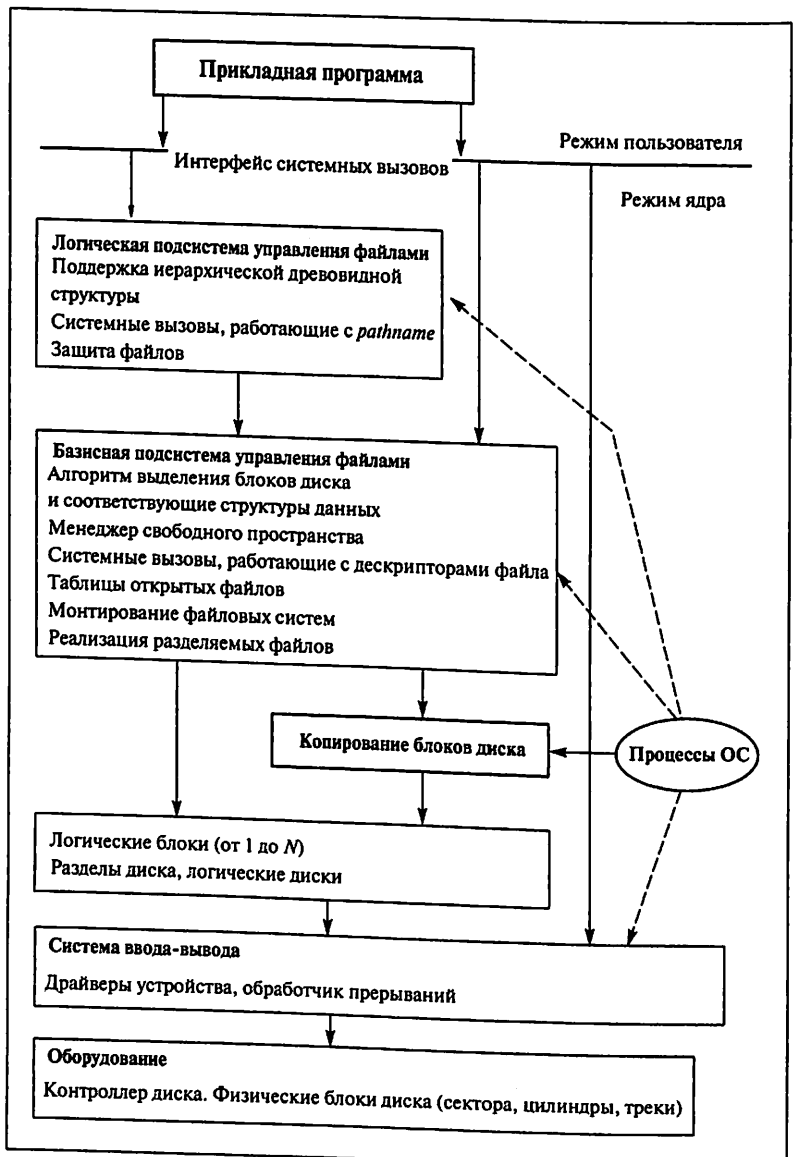

Рис. 11.1. Блок-схема файловой системы

Непосредственно с устройствами (дисками) взаимодействует часть операционной системы, называемая *системой ввода-вывода* (см. гл. 12). Система ввода-вывода предоставляет в распоряжение более высокоуровневого компонента операционной системы — файловой системы — используемое дисковое пространство в виде *непрерывной последовательности блоков фиксированного размера*. Система ввода-вывода имеет дело с *физическими* блоками диска, которые характеризуются адресом, например диск 2, цилиндр 75, сектор 11. Файловая система имеет дело с *логическими* блоками, каждый из которых имеет номер (от 0 или 1 до *N*). Размер логических блоков файла совпадает или является кратным размеру физического блока диска и может быть задан равным размеру страницы виртуальной памяти, поддерживаемой аппаратурой компьютера совместно с операционной системой.

В структуре системы управления файлами можно выделить базисную подсистему, которая отвечает за выделение дискового пространства конкретным файлам, и более высокоуровневую логическую подсистему, которая использует структуру дерева директорий для предоставления модулю базисной подсистемы необходимой ей информации, исходя из символического имени файла. Она также ответственна за авторизацию доступа к файлам.

Стандартный запрос на открытие (*open*) или создание (*create*) файла поступает от прикладной программы к логической подсистеме. Логическая подсистема, используя структуру директорий, проверяет права доступа и вызывает базовую подсистему для получения доступа к блокам файла. После этого файл считается открытым, он содержится в таблице открытых файлов, и прикладная программа получает в свое распоряжение дескриптор (или *handle* в системах Microsoft) этого файла. Дескриптор файла является ссылкой на файл в таблице открытых файлов и используется в запросах прикладной программы на чтение-запись из этого файла. Запись в таблице открытых файлов указывает через систему выделения блоков диска на блоки данного файла. Если к моменту открытия файл уже используется другим процессом, т.е. содержится в таблице открытых файлов, то после проверки прав доступа к файлу может быть организован совместный доступ. При этом новому процессу также возвращается дескриптор — ссылка на файл в таблице открытых файлов. Далее подробно проанализирована работа наиболее важных системных вызовов.

11.2. Управление внешней памятью

Прежде чем описывать структуру данных файловой системы на диске, необходимо рассмотреть алгоритмы выделения дискового пространства и способы учета свободной и занятой дисковой памяти. Эти задачи связаны между собой.

Методы выделения дискового пространства

Ключевым, безусловно, является вопрос, какой тип структур используется для учета отдельных блоков файла, т.е. способ связывания файлов с блоками диска. В операционных системах используется несколько методов выделения файлу дискового пространства. Для каждого из методов запись в директории, соответствующая символьному имени файла, содержит указатель, следуя которому, можно найти все блоки данного файла.

Выделение непрерывной последовательностью блоков

Простейший способ — хранить каждый файл как непрерывную последовательность блоков диска. При непрерывном расположении файл характеризуется адресом и длиной (в блоках). Файл, стартующий с блока b, занимает затем блоки $b + 1$, $b + 2$, ... $b + n - 1$.

Эта схема имеет два преимущества. Во-первых, ее легко реализовать, так как выяснение местонахождения файла сводится к вопросу, где находится первый блок. Во-вторых, она обеспечивает хорошую производительность, так как целый файл может быть считан за одну дисковую операцию.

Непрерывное выделение используется в операционной системе IBM/CMS, в операционной системе RSX-11 (для выполняемых файлов) и в ряде других.

Этот способ распространен мало, и вот почему. В процессе эксплуатации диск представляет собой некоторую совокупность свободных и занятых фрагментов. Не всегда имеется подходящий по размеру свободный фрагмент для нового файла. Проблема непрерывного расположения может рассматриваться как частный случай более общей проблемы выделения блока нужного размера из списка свободных блоков. Типовыми решениями этой задачи являются стратегии первого подходящего, наиболее подходящего и наименее подходящего (сравните с проблемой выделения памяти в методе с динамическим распределением). Как и в случае выделения нужного объема оперативной памяти в схеме с динамическими разделами, метод страдает от *внешней фрагментации*, в большей или меньшей степени, в зависимости от размера диска и среднего размера файла.

Кроме того, непрерывное распределение внешней памяти неприменимо до тех пор, пока неизвестен максимальный размер файла. Иногда размер выходного файла оценить легко (при копировании). Чаще, однако, это трудно сделать, особенно в тех случаях, когда размер файла меняется. Если места не хватило, то пользовательская программа может быть приостановлена с учетом выделения дополнительного места для файла при последующем рестарте. Некоторые операционные системы используют модифицированный вариант непрерывного выделения — основные блоки файла + резервные блоки. Однако с выделением блоков из резерва возникают те же проблемы, так как приходится решать задачу выделения непрерывной последовательности блоков диска теперь уже из совокупности резервных блоков.

Единственным приемлемым решением перечисленных проблем является периодическое уплотнение содержимого внешней памяти, или «сборка мусора», цель которой состоит в объединении свободных участков в один большой блок. Но это дорогостоящая операция, которую невозможно осуществлять слишком часто.

Таким образом, когда содержимое диска постоянно изменяется, данный метод нерационален. Однако для *стационарных* файловых систем, например для файловых систем компакт-дисков, он вполне пригоден.

Связный список

Внешняя фрагментация — основная проблема рассмотренного выше метода — может быть устранена за счет представления файла в виде связного списка блоков диска. Запись в директории содержит указатель на первый и последний блоки файла (иногда в качестве варианта используется специальный знак конца файла — EOF). Каждый блок содержит указатель на следующий блок (рис. 11.2).

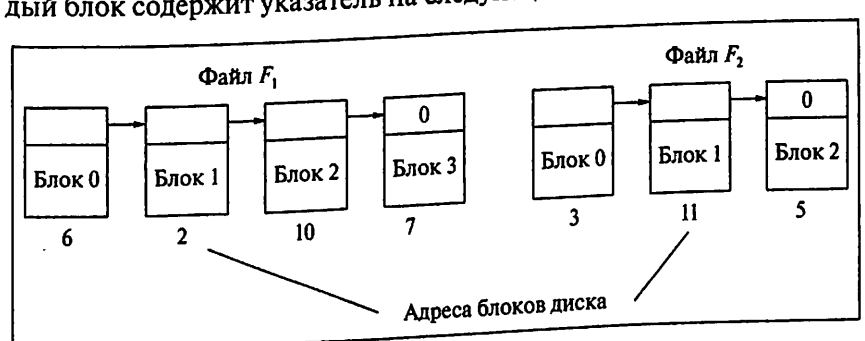

Рис. 11.2. Хранение файла в виде связного списка дисковых блоков

Внешняя фрагментация для данного метода отсутствует. Любой свободный блок может быть использован для удовлетворения запроса. Заметим, что нет необходимости декларировать размер файла в момент создания. Файл может расти неограниченно.

Связное выделение имеет, однако, несколько существенных недостатков.

Во-первых, при прямом доступе к файлу для поиска i-го блока нужно осуществить несколько обращений к диску, последовательно считывая блоки от 1 до $i - 1$, т.е. выборка логически смежных записей, которые занимают физически несмежные секторы, может требовать много времени. Здесь мы теряем все преимущества прямого доступа к файлу.

Во-вторых, данный способ не очень надежен. Наличие дефектного блока в списке приводит к потере информации в оставшейся части файла и потенциально к потере дискового пространства, отведенного под этот файл.

Наконец, для указателя на следующий блок внутри блока нужно выделить место, что не всегда удобно. Емкость блока, традиционно являющаяся степенью двойки (многие программы читают и пишут блоками по степеням двойки), таким образом, перестает быть степенью двойки, так как указатель отбирает несколько байтов.

Поэтому метод связного списка обычно в чистом виде не используется.

Таблица отображения файлов

Одним из вариантов предыдущего способа является хранение указателей не в дисковых блоках, а в индексной таблице в памяти, которая называется таблицей отображения файлов (*FAT — file allocation table*) (рис. 11.3). Этой схемы придерживаются многие операционные системы (MS-DOS, OS/2, MS Windows и др.).

По-прежнему существенно, что запись в директории содержит только ссылку на первый блок. Далее при помощи таблицы FAT можно локализовать блоки файла независимо от его размера. В тех строках таблицы, которые соответствуют последним блокам файлов, обычно записывается некоторое граничное значение, например EOF.

Главное достоинство данного подхода состоит в том, что по таблице отображения можно судить о физическом соседстве блоков, располагающихся на диске, и при выделении нового блока можно легко найти свободный блок диска, находящийся поблизости от других блоков данного файла. Минусом данной схемы может быть необходимость хранения в памяти этой довольно большой таблицы.

Номера блоков диска		
1		
2	10	
3	11	Начало файла F_2
4		
5	EOF	
6	2	Начало файла F_1
7	EOF	
8		
9		
10	7	
11	5	

Рис. 11.3. Метод связного списка с использованием таблицы в оперативной памяти

Индексные узлы

Наиболее распространенный метод выделения файлу блоков диска — связать с каждым файлом небольшую таблицу, называемую индексным узлом (*i-node*), которая перечисляет атрибуты и дисковые адреса блоков файла (рис. 11.4). Запись в директории, относящаяся к файлу, содержит адрес индексного блока. По мере заполнения файла указатели на блоки диска в индексном узле принимают осмысленные значения.

Индексирование поддерживает прямой доступ к файлу, без ущерба от внешней фрагментации. Индексированное размещение широко распространено и поддерживает как последовательный, так и прямой доступ к файлу.

Обычно применяется комбинация одноуровневого и многоуровневых индексов. Первые несколько адресов блоков файла хранятся непосредственно в индексном узле, таким образом, для маленьких файлов индексный узел хранит всю необходимую информацию об адресах блоков диска. Для больших файлов один из адресов индексного узла указывает на блок косвенной адресации. Данный блок содержит адреса дополнительных блоков диска. Если этого недостаточно, используется блок двойной косвенной адресации, который

содержит адреса блоков косвенной адресации. Если и этого не хватает, используется блок тройной косвенной адресации.

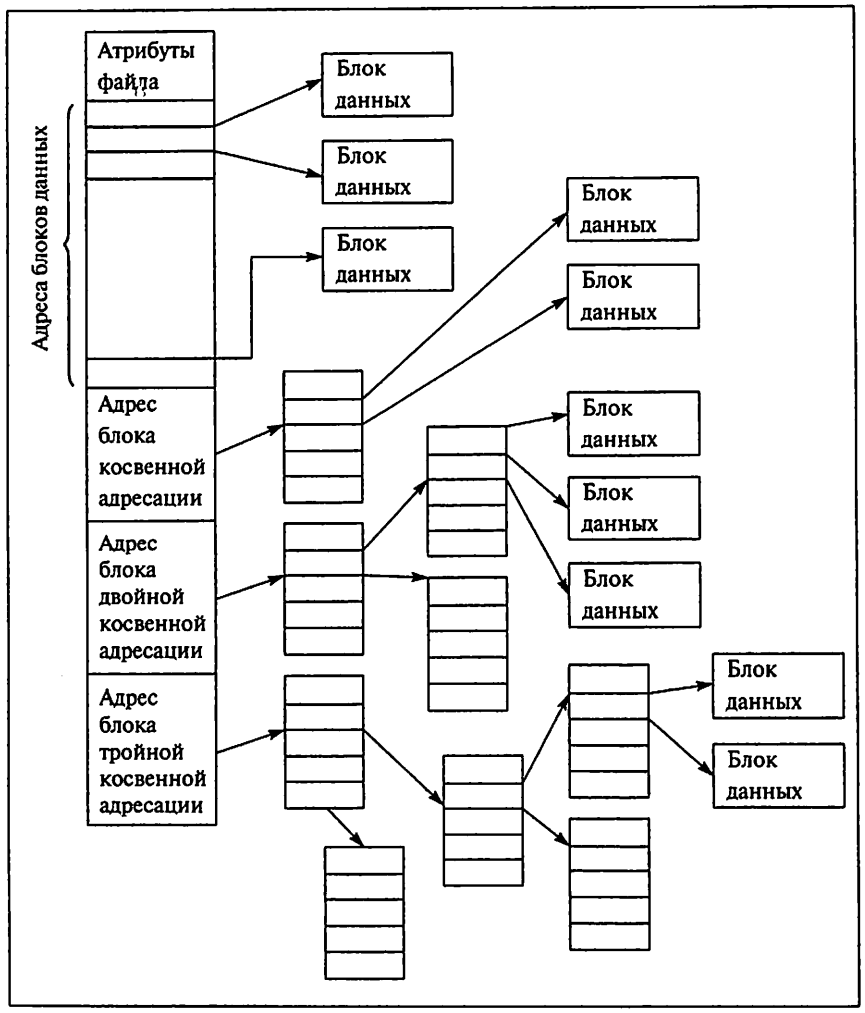

Рис. 11.4. Структура индексного узла

Данную схему используют файловые системы Unix (а также файловые системы HPFS, NTFS и др.). Такой подход позволяет при фиксированном, относительно небольшом размере индексного узла поддерживать работу с файлами, размер которых может меняться от нескольких байтов до нескольких гигабайтов. Существенно, что для маленьких файлов используется только прямая адресация, обеспечивающая максимальную производительность.

Управление свободным и занятым дисковым пространством

Дисковое пространство, не выделенное ни одному файлу, также должно быть управляемым. В современных операционных системах используется несколько способов учета используемого места на диске. Рассмотрим наиболее распространенные.

Учет при помощи организации битового вектора

Часто список свободных блоков диска реализован в виде битового вектора (*bit map* или *bit vector*). Каждый блок представлен одним битом, принимающим значение 0 или 1, в зависимости от того, занят он или свободен. Например, 00111100111100011000001 ... .

Главное преимущество этого подхода состоит в том, что он относительно прост и эффективен при нахождении первого свободного блока или *n* последовательных блоков на диске. Многие компьютеры имеют инструкции манипулирования битами, которые могут использоваться для этой цели. Например, компьютеры семейств Intel и Motorola имеют инструкции, при помощи которых можно легко локализовать первый единичный бит в слове.

Описываемый метод учета свободных блоков используется в Apple Macintosh.

Несмотря на то что размер описанного битового вектора наименьший из всех возможных структур, даже такой вектор может оказаться большого размера. Поэтому данный метод эффективен, только если битовый вектор помещается в памяти целиком, что возможно лишь для относительно небольших дисков. Например, диск размером 4 Гбайт с блоками по 4 Кбайт нуждается в таблице размером 128 Кбайт для управления свободными блоками. Иногда, если битовый вектор становится слишком большим, для ускорения поиска в нем его разбивают на регионы и организуют резюмирующие структуры данных, содержащие сведения о количестве свободных блоков для каждого региона.

Учет при помощи организации связного списка

Другой подход — связать в список все свободные блоки, размещая указатель на первый свободный блок в специально отведенном месте диска, попутно кэшируя в памяти эту информацию.

Подобная схема не всегда эффективна. Для трассирования списка нужно выполнить много обращений к диску. Однако, к счастью, нам необходим, как правило, только первый свободный блок.

Иногда прибегают к модификации подхода связного списка, организуя хранение адресов *n* свободных блоков в первом свободном

блоке. Первые $n - 1$ этих блоков действительно используются. Последний блок содержит адреса других n блоков и т.д.

Существуют и другие методы, например свободное пространство можно рассматривать как файл и вести для него соответствующий индексный узел.

Размер блока

Размер логического блока играет важную роль. В некоторых системах (Unix) он может быть задан при форматировании диска. Небольшой размер блока будет приводить к тому, что каждый файл будет содержать много блоков. Чтение блока осуществляется с задержками на поиск и вращение, таким образом, файл из многих блоков будет читаться медленно. Большие блоки обеспечивают более высокую скорость обмена с диском, но из-за внутренней фрагментации (каждый файл занимает целое число блоков, и в среднем половина последнего блока пропадает) снижается процент полезного дискового пространства.

Для систем со страничной организацией памяти характерна сходная проблема с размером страницы.

Проведенные исследования показали, что большинство файлов имеют небольшой размер. Например, в Unix приблизительно 85% файлов имеют размер менее 8 Кбайт и 48% — менее 1 Кбайта.

Можно также учесть, что в системах с виртуальной памятью желательно, чтобы единицей пересылки диск-память была страница (наиболее распространенный размер страниц памяти — 4 Кбайта). Отсюда обычный компромиссный выбор блока размером 512 байт, 1 Кбайт, 2 Кбайт, 4 Кбайт.

Структура файловой системы на диске

Рассмотрение методов работы с дисковым пространством дает общее представление о совокупности служебных данных, необходимых для описания файловой системы. Структура служебных данных типовой файловой системы, например Unix, на одном из разделов диска, таким образом, может состоять из четырех основных частей (рис. 11.5).

В начале раздела находится суперблок, содержащий общее описание файловой системы, например:

- тип файловой системы;
- размер файловой системы в блоках;
- размер массива индексных узлов;
- размер логического блока.

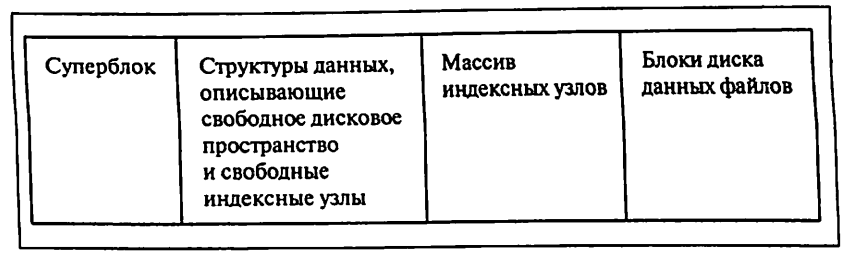

Рис. 11.5. Примерная структура файловой системы на диске

Описанные структуры данных создаются на диске в результате его *форматирования* (например, утилитами *format, makefs* и др.). Их наличие позволяет обращаться к данным на диске как к файловой системе, а не как к обычной последовательности блоков.

В файловых системах современных операционных систем для повышения устойчивости поддерживается несколько копий суперблока. В некоторых версиях Unix суперблок включал также и структуры данных, управляющие распределением дискового пространства, в результате чего суперблок непрерывно подвергался модификации, что снижало надежность файловой системы в целом. Выделение структур данных, описывающих дисковое пространство, в отдельную часть является более правильным решением.

Массив индексных узлов (*ilist*) содержит список индексов, соответствующих файлам данной файловой системы. Размер массива индексных узлов определяется администратором при установке системы. Максимальное число файлов, которые могут быть созданы в файловой системе, определяется числом доступных индексных узлов.

В блоках данных хранятся реальные данные файлов. Размер логического блока данных может задаваться при форматировании файловой системы. Заполнение диска содержательной информацией предполагает использование блоков хранения данных для файлов директорий и обычных файлов и имеет следствием модификацию массива индексных узлов и данных, описывающих пространство диска. Отдельно взятый блок данных может принадлежать одному и только одному файлу в файловой системе.

11.3. Реализация директорий

Как уже говорилось, директория или каталог — это файл, имеющий вид таблицы и хранящий список входящих в него файлов или каталогов. Основная задача файлов-директорий — поддержка иерар-

хической древовидной структуры файловой системы. Запись в директории имеет определенный для данной операционной системы формат, зачастую неизвестный пользователю, поэтому блоки данных файла-директории заполняются не через операции записи, а при помощи специальных системных вызовов (например, создание файла).

Для доступа к файлу операционная система использует путь (pathname), сообщенный пользователем. Запись в директории связывает имя файла или имя поддиректории с блоками данных на диске (рис. 11.6). В зависимости от способа выделения файлу блоков диска эта ссылка может быть номером первого блока или номером индексного узла. В любом случае обеспечивается связь символьного имени файла с данными на диске.

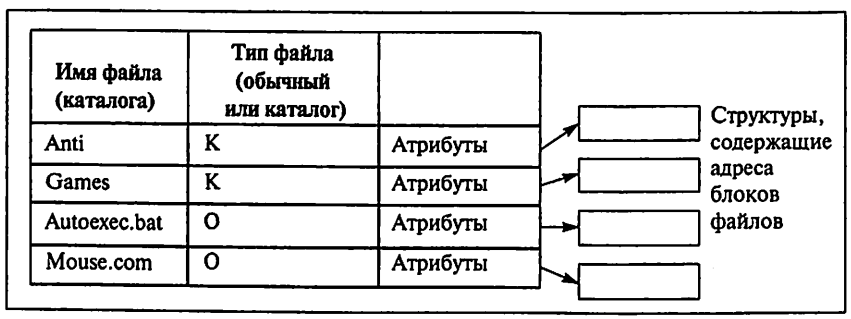

Рис. 11.6. Реализация директорий

Когда система открывает файл, она ищет его имя в директории. Затем из записи в директории или из структуры, на которую запись в директории указывает, извлекаются атрибуты и адреса блоков файла на диске. Эта информация помещается в системную таблицу в главной памяти. Все последующие ссылки на данный файл используют эту информацию. Атрибуты файла можно хранить непосредственно в записи в директории, как показано на рис. 11.6. Однако для организации совместного доступа к файлам удобнее хранить атрибуты в индексном узле, как это делается в Unix.

Рассмотрим несколько конкретных примеров.

Примеры реализации директорий в некоторых операционных системах

Директории в операционной системе MS-DOS

В операционной системе MS-DOS типовая запись в директории имеет вид, показанный на рис. 11.7.

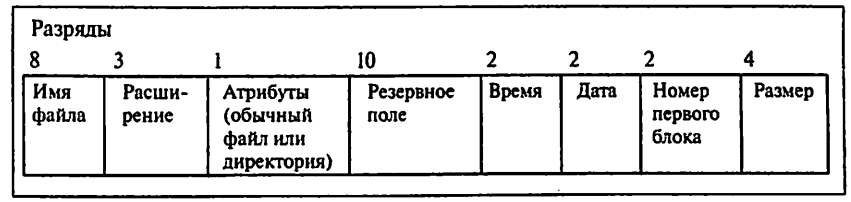

Рис. 11.7. Вариант записи в директории MS-DOS

В операционной системе MS-DOS, как и в большинстве современных операционных систем, директории могут содержать поддиректории (специфицируемые битом атрибута), что позволяет конструировать произвольное дерево директорий файловой системы.

Номер первого блока используется в качестве индекса в таблице FAT. Далее по цепочке в этой таблице могут быть найдены остальные блоки.

Директории в операционной системе Unix

Структура директории проста. Каждая запись содержит имя файла и номер его индексного узла (рис. 11.8). Вся остальная информация о файле (тип, размер, время модификации, владелец и т.д. и номера дисковых блоков) находится в индексном узле.

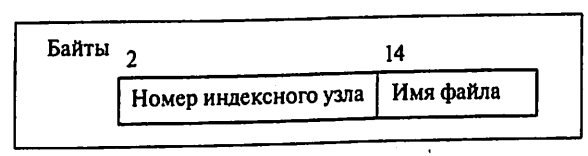

Рис. 11.8. Вариант записи в директории Unix

В более поздних версиях Unix форма записи претерпела ряд изменений, например имя файла описывается структурой. Однако суть осталась прежней.

Поиск в директории

Список файлов в директории обычно не является упорядоченным по именам файлов. Поэтому правильный выбор алгоритма поиска имени файла в директории имеет большое влияние на эффективность и надежность файловых систем.

Линейный поиск

Существует несколько стратегий просмотра списка символьных имен. Простейшей из них является линейный поиск. Директория просматривается с самого начала, пока не встретится нужное имя

файла. Хотя это наименее эффективный способ поиска, оказывается, что в большинстве случаев он работает с приемлемой производительностью. Например, авторы Unix утверждали, что линейного поиска вполне достаточно. По-видимому, это связано с тем, что на фоне относительно медленного доступа к диску некоторые задержки, возникающие в процессе сканирования списка, несущественны.

Метод прост, но требует временных затрат. Для создания нового файла вначале нужно проверить директорию на наличие такого же имени. Затем имя нового файла вставляется в конец директории (если, разумеется, файл с таким же именем в директории не существует, в противном случае нужно информировать пользователя). Для удаления файла нужно также выполнить поиск его имени в списке и пометить запись как неиспользуемую.

Реальный недостаток данного метода — последовательный поиск файла. Информация о структуре директории используется часто, и неэффективный способ поиска будет заметен пользователями. Можно свести поиск к бинарному, если отсортировать список файлов. Однако это усложнит создание и удаление файлов, так как требуется перемещение большого объема информации.

Хеш-таблица

Хеширование [Aho, 2001] — другой способ, который может использоваться для размещения и последующего поиска имени файла в директории. В данном методе имена файлов также хранятся в каталоге в виде линейного списка, но дополнительно используется хеш-таблица. Хеш-таблица, точнее построенная на ее основе хеш-функция, позволяет по имени файла получить указатель на имя файла в списке. Таким образом, можно существенно уменьшить время поиска.

В результате хеширования могут возникать коллизии, т.е. ситуации, когда функция хеширования, примененная к разным именам файлов, дает один и тот же результат. Обычно имена таких файлов объединяют в связные списки, предполагая в дальнейшем осуществление в них последовательного поиска нужного имени файла. Выбор подходящего алгоритма хеширования позволяет свести к минимуму число коллизий. Однако всегда есть вероятность неблагоприятного исхода, когда непропорционально большому числу имен файлов функция хеширования ставит в соответствие один и тот же результат. В таком случае преимущество использования этой схемы по сравнению с последовательным поиском практически утрачивается.

Другие методы поиска

Помимо описанных методов поиска имени файла, в директории существуют и другие. В качестве примера можно привести организацию поиска в каталогах файловой системы NTFS при помощи так называемого B-дерева, которое стало стандартным способом организации индексов в системах баз данных [Aho, 2001].

11.4. Монтирование файловых систем

Так же как файл должен быть открыт перед использованием, и файловая система, хранящаяся на разделе диска, должна быть смонтирована, чтобы стать доступной процессам системы.

Функция *mount* (монтировать) связывает файловую систему из указанного раздела на диске с существующей иерархией файловых систем, а функция *umount* (демонтировать) выключает файловую систему из иерархии. Функция *mount*, таким образом, дает пользователям возможность обращаться к данным в дисковом разделе как к файловой системе, а не как к последовательности дисковых блоков.

Процедура монтирования состоит в следующем. Пользователь (в Unix это суперпользователь) сообщает операционной системе имя устройства и место в файловой структуре (имя пустого каталога), куда нужно присоединить файловую систему (точка монтирования) (рис. 11.9 и 11.10). Например, в операционной системе Unix библиотечный вызов *mount* имеет вид:

mount (*special pathname, directory pathname, options*);

где *special pathname* — имя специального файла устройства (в общем случае имя раздела), соответствующего дисковому разделу с монтируемой файловой системой, *directory pathname* — каталог в существующей иерархии, где будет монтироваться файловая система (другими словами, точка или место монтирования), а *options* указывает, следует ли монтировать файловую систему «только для чтения» (при этом не будут выполняться такие функции, как *write* и *create*, которые производят запись в файловую систему). Затем операционная система должна убедиться, что устройство содержит действительную файловую систему ожидаемого формата с суперблоком, списком индексов и корневым индексом. Некоторые операционные системы осуществляют монтирование автоматически, как только встретят диск в первый раз (жесткие диски на этапе загрузки, гибкие — когда они вставлены в дисковод), операционная система ищет файловую систему на устройстве. Если файловая сис-

тема на устройстве имеется, она монтируется на корневом уровне, при этом к цепочке имен абсолютного имени файла (*pathname*) добавляется буква раздела.

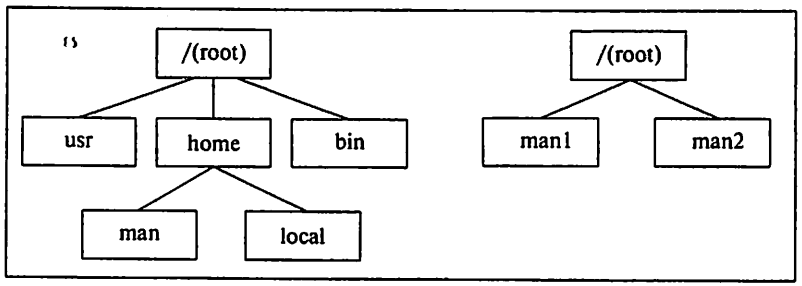

Рис. 11.9. Две файловые системы до монтирования

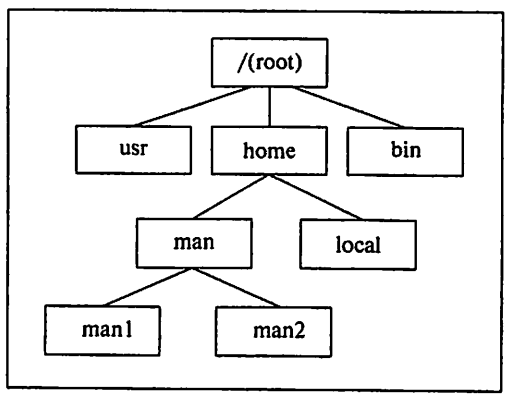

Рис. 11.10. Общая файловая система после монтирования

Ядро поддерживает таблицу монтирования с записями о каждой смонтированной файловой системе. В каждой записи содержится информация о вновь смонтированном устройстве, о его суперблоке и корневом каталоге, а также сведения о точке монтирования. Для устранения потенциально опасных побочных эффектов число линков к каталогу — точке монтирования — должно быть равно 1. Занесение информации в таблицу монтирования производится немедленно, поскольку может возникнуть конфликт между двумя процессами. Например, если монтирующий процесс приостановлен для открытия устройства или считывания суперблока файловой системы, а тем временем другой процесс может попытаться смонтировать файловую систему.

Наличие в логической структуре файлового архива точек монтирования требует аккуратной реализации алгоритмов, осуществляющих навигацию по каталогам. Точку монтирования можно пересечь двумя способами: из файловой системы, где производится монтирование, в файловую систему, которая монтируется (в направлении от глобального корня к листу), и в обратном направлении. Алгоритмы поиска файлов должны предусматривать ситуации, в которых очередной компонент пути к файлу является точкой монтирования, когда вместо анализа индексного узла очередной директории приходится осуществлять обработку суперблока монтированной системы.

11.5. Связывание файлов

Иерархическая организация, положенная в основу древовидной структуры файловой системы современных операционных систем, не предусматривает выражения отношений, в которых потомки связываются более чем с одним предком. Такая негибкость частично устраняется возможностью реализации связывания файлов или организации линков (link).

Ядро позволяет пользователю связывать каталоги, упрощая написание программ, требующих пересечения дерева файловой системы (рис. 11.11). Часто имеет смысл хранить под разными именами одну и ту же команду (выполняемый файл). Например, выполняемый файл традиционного текстового редактора операционной системы Unix *vi* обычно может вызываться под именами *ex, edit, vi, view* и *vedit* файловой системы. Соединение между директорией и разделяемым файлом называется «связью» или «ссылкой» (link). Дерево файловой системы превращается в циклический граф.

Это удобно, но создает ряд дополнительных проблем.

Простейший способ реализовать связывание файла — просто дублировать информацию о нем в обеих директориях. При этом, однако, может возникнуть проблема совместимости в случае, если владельцы этих директорий попытаются независимо друг от друга изменить содержимое файла. Например, в операционной системе CP/M запись в директории о файле непосредственно содержит адреса дисковых блоков. Поэтому копии тех же дисковых адресов должны быть сделаны и в другой директории, куда файл линкуется. Если один из пользователей что-то добавляет к файлу, новые блоки будут перечислены только у него в директории и не будут «видны» другому пользователю.

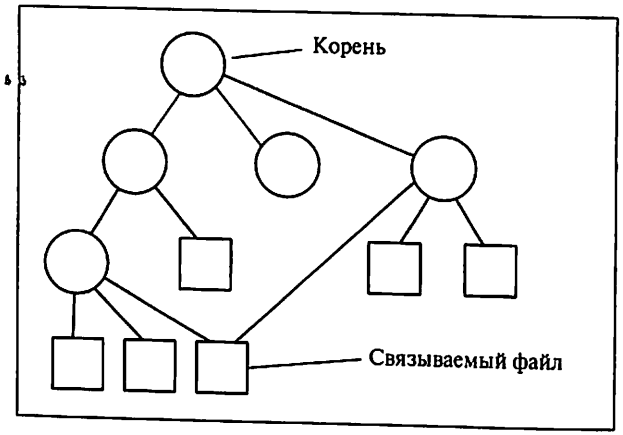

Рис. 11.11. Структура файловой системы с возможностью связывания файла с новым именем

Проблема такого рода может быть решена двумя способами. Первый из них — так называемая *жесткая* ссылка (hard link). Если блоки данных файла перечислены не в директории, а в небольшой структуре данных (например, в индексном узле), связанной собственно с файлом, то второй пользователь может связаться непосредственно с этой, уже существующей структурой.

Альтернативное решение — создание нового файла, который содержит путь к связываемому файлу. Такой подход называется *символической* ссылкой (soft или symbolic link). При этом в соответствующем каталоге создается элемент, в котором имени ссылки сопоставляется некоторое имя файла (этот файл даже не обязан существовать к моменту создания символической ссылки). Для символической ссылки может создаваться отдельный индексный узел и даже заводиться отдельный блок данных для хранения потенциально длинного имени файла.

Каждый из этих методов имеет свои минусы. В случае жесткой ссылки возникает необходимость поддержки счетчика ссылок на файл для корректной реализации операции удаления файла. Например, в Unix такой счетчик является одним из атрибутов, хранящихся в индексном узле. Удаление файла одним из пользователей уменьшает количество ссылок на файл на 1. Реальное удаление файла происходит, когда число ссылок на файл становится равным 0.

В случае символической ссылки такая проблема не возникает, так как только реальный владелец имеет ссылку на индексный узел файла. Если собственник удаляет файл, то он разрушается, и по-

пытки других пользователей работать с ним закончатся провалом. Удаление символической ссылки на файл никак не влияет. Проблема организации символической ссылки — потенциальное снижение скорости доступа к файлу. Файл символической ссылки хранит путь к файлу, содержащий список вложенных директорий, для прохождения по которому необходимо осуществить несколько обращений к диску.

Символическая ссылка имеет то преимущество, что она может использоваться для организации удобного доступа к файлам удаленных компьютеров, если, например, добавить к пути сетевой адрес удаленной машины.

Циклический граф — структура более гибкая, нежели простое дерево, но работа с ней требует большой аккуратности. Поскольку теперь к файлу существует несколько путей, программа поиска файла может найти его на диске несколько раз. Простейшее практическое решение данной проблемы — ограничить число директорий при поиске. Полное устранение циклов при поиске — довольно трудоемкая процедура, выполняемая специальными утилитами и связанная с многократной трассировкой директорий файловой системы.

11.6. Кооперация процессов при работе с файлами

Когда различные пользователи работают вместе над проектом, они часто нуждаются в разделении файлов.

Разделяемый файл — разделяемый ресурс. Как и в случае любого совместно используемого ресурса, процессы должны синхронизировать доступ к совместно используемым файлам, каталогам, чтобы избежать тупиковых ситуаций, дискриминации отдельных процессов и снижения производительности системы.

Например, если несколько пользователей одновременно редактируют какой-либо файл и не принято специальных мер, то результат будет непредсказуем и зависит от того, в каком порядке осуществлялись записи в файл. Между двумя операциями *read* одного процесса другой процесс может модифицировать данные, что для многих приложений неприемлемо. Простейшее решение данной проблемы — предоставить возможность одному из процессов захватить файл, т.е. блокировать доступ к разделяемому файлу других процессов на все время, пока файл остается открытым для данного процесса. Однако это было бы недостаточно гибко и не соответствовало бы характеру поставленной задачи.

Рассмотрим вначале *грубый подход*, т.е. временный захват пользовательским процессом файла или записи (части файла между указанными позициями).

Системный вызов, позволяющий установить и проверить блокировки на файл, является неотъемлемым атрибутом современных многопользовательских операционных систем. В принципе было бы логично связать синхронизацию доступа к файлу как к единому целому с системным вызовом *open* (т.е., например, открытие файла в режиме записи или обновления могло бы означать его монопольную блокировку соответствующим процессом, а открытие в режиме чтения — совместную блокировку). Так поступают во многих операционных системах (начиная с операционной системы Multics). В операционной системе Unix это не так, что имеет исторические причины.

В первой версии системы Unix, разработанной Томпсоном и Ричи, механизм захвата файла отсутствовал. Применялся очень простой подход к обеспечению параллельного (от нескольких процессов) доступа к файлам: система позволяла любому числу процессов одновременно открывать один и тот же файл в любом режиме (чтения, записи или обновления) и не предпринимала никаких синхронизационных действий. Вся ответственность за корректность совместной обработки файла ложилась на использующие его процессы, и система даже не предоставляла каких-либо особых средств для синхронизации доступа процессов к файлу. Однако впоследствии для того, чтобы повысить привлекательность системы для коммерческих пользователей, работающих с базами данных, в версию *V* системы были включены механизмы захвата файла и записи, базирующиеся на системном вызове *fcntl*.

Допускается два варианта синхронизации: с ожиданием, когда требование блокировки может привести к откладыванию процесса до того момента, когда это требование может быть удовлетворено, и без ожидания, когда процесс немедленно оповещается об удовлетворении требования блокировки или о невозможности ее удовлетворения в данный момент.

Установленные блокировки относятся только к тому процессу, который их установил, и не наследуются процессами — потомками этого процесса. Более того, даже если некоторый процесс пользуется синхронизационными возможностями системного вызова *fcntl*, другие процессы по-прежнему могут работать с тем файлом без всякой синхронизации. Другими словами, это дело группы процессов, совместно использующих файл, — договориться о способе синхронизации параллельного доступа.

Более тонкий подход заключается в прозрачной для пользователя блокировке отдельных структур ядра, отвечающих за работу с файлами части пользовательских данных. Например, в операционной системе Unix во время системного вызова, осуществляющего ту или иную операцию с файлом, как правило, происходит блокирование индексного узла, содержащего адреса блоков данных файла. Может показаться, что организация блокировок или запрета более чем одному процессу работать с файлом во время выполнения системного вызова является излишней, так как в подавляющем большинстве случаев выполнение системных вызовов и так не прерывается, т.е. ядро работает в условиях невытесняющей многозадачности. Однако в данном случае это не совсем так. Операции чтения и записи занимают продолжительное время и лишь инициируются центральным процессором, а осуществляются по независимым каналам, поэтому установка блокировок на время системного вызова является необходимой гарантией атомарности операций чтения и записи. На практике оказывается достаточным заблокировать один из буферов кэша диска, в заголовке которого ведется список процессов, ожидающих освобождения данного буфера. Таким образом, в соответствии с семантикой Unix изменения, сделанные одним пользователем, немедленно становятся «видны» другому пользователю, который держит данный файл открытым одновременно с первым.

Примеры разрешения коллизий и тупиковых ситуаций

Логика работы системы в сложных ситуациях может проиллюстрировать особенности организации мультидоступа.

Рассмотрим в качестве примера *образование потенциального тупика* при создании связи (link), когда разрешен совместный доступ к файлу [Bach, 1986].

Два процесса, выполняющие одновременно следующие функции:

- процесс А: *link(«a/b/c/d»,»e/f/g»)*;
- процесс В: *link(«e/f»,»a/b/c/d/ee»)*;

могут зайти в тупик. Предположим, что процесс А обнаружил индекс файла *«a/b/c/d»* в тот самый момент, когда процесс В обнаружил индекс файла *«e/f»*. Фраза «в тот же самый момент» означает, что системой достигнуто состояние, при котором каждый процесс получил искомый индекс. Когда же теперь процесс А попытается получить индекс файла *«e/f»*, он приостановит свое выполнение до тех пор, пока индекс файла *«f»* не освободится. В то же время процесс В пытается получить индекс каталога *«a/b/c/d»* и приостанавливается в ожидании освобождения индекса файла *«d»*. Процесс А будет удер-

живать заблокированным индекс, нужный процессу В, а процесс В, в свою очередь, будет удерживать заблокированным индекс, необходимый процессу А.

Для предотвращения этого классического примера взаимной блокировки в файловой системе принято, чтобы ядро освобождало индекс исходного файла после увеличения значения счетчика связей. Тогда, поскольку первый из ресурсов (индекс) свободен при обращении к следующему ресурсу, взаимной блокировки не происходит.

Поводов для нежелательной конкуренции между процессами много, особенно при удалении имен каталогов. Предположим, что один процесс пытается найти данные файла по его полному символическому имени, последовательно проходя компонент за компонентом, а другой процесс удаляет каталог, имя которого входит в путь поиска. Допустим, процесс А делает разбор имени *«a/b/c/d»* и приостанавливается во время получения индексного узла для файла *«c»*. Он может приостановиться при попытке заблокировать индексный узел или при попытке обратиться к дисковому блоку, где этот индексный узел хранится. Если процессу В нужно удалить связь для каталога с именем *«c»*, он может приостановиться по той же самой причине, что и процесс А. Пусть ядро впоследствии решит возобновить процесс В раньше процесса А. Прежде чем процесс А продолжит свое выполнение, процесс В завершится, удалив связь каталога *«c»* и его содержимое по этой связи. Позднее процесс А попытается обратиться к несуществующему индексному узлу, который уже был удален. Алгоритм поиска файла, проверяющий в первую очередь неравенство значения счетчика связей нулю, должен сообщить об ошибке.

Можно привести и другие примеры, которые демонстрируют необходимость тщательного проектирования файловой системы для ее последующей надежной работы.

11.7. Надежность файловой системы

Жизнь полна неприятных неожиданностей, а разрушение файловой системы зачастую более опасно, чем разрушение компьютера. Поэтому файловые системы должны разрабатываться с учетом подобной возможности. Помимо очевидных решений, например своевременное дублирование информации (backup), файловые системы современных операционных систем содержат специальные средства для поддержки собственной совместимости.

Целостность файловой системы

Важный аспект надежной работы файловой системы — контроль ее целостности. В результате файловых операций блоки диска могут считываться в память, модифицироваться и затем записываться на диск. Причем многие файловые операции затрагивают сразу несколько объектов файловой системы. Например, копирование файла предполагает выделение ему блоков диска, формирование индексного узла, изменение содержимого каталога и т.д. В течение короткого периода времени между этими шагами информация в файловой системе оказывается несогласованной.

И если вследствие непредсказуемой остановки системы на диске будут сохранены изменения только для части этих объектов (нарушена атомарность файловой операции), файловая система на диске может быть оставлена в несовместимом состоянии. В результате могут возникнуть нарушения логики работы с данными, например, появиться «потерянные» блоки диска, которые не принадлежат ни одному файлу и в то же время помечены как занятые, или, наоборот, блоки, помеченные как свободные, но в то же время занятые (на них есть ссылка в индексном узле) или другие нарушения.

В современных операционных системах предусмотрены меры, которые позволяют свести к минимуму ущерб от порчи файловой системы и затем полностью или частично восстановить ее целостность.

Порядок выполнения операций

Очевидно, что для правильного функционирования файловой системы значимость отдельных данных неравноценна. Искажение содержимого пользовательских файлов не приводит к серьезным (с точки зрения целостности файловой системы) последствиям, тогда как несоответствия в файлах, содержащих управляющую информацию (директории, индексные узлы, суперблок и т.п.), могут быть катастрофическими. Поэтому должен быть тщательно продуман порядок выполнения операций со структурами данных файловой системы.

Рассмотрим пример создания жесткой связи для файла [Робачевский, 1999]. Для этого файловой системе необходимо выполнить следующие операции:

- создать новую запись в каталоге, указывающую на индексный узел файла;
- увеличить счетчик связей в индексном узле.

Если аварийный останов произошел между 1-й и 2-й операциями, то в каталогах файловой системы будут существовать два имени файла, адресующих индексный узел со значением счетчика связей, равным 1. Если теперь будет удалено одно из имен, это приведет к удалению файла как такового. Если же порядок операций изменен и, как прежде, останов произошел между первой и второй операциями, файл будет иметь несуществующую жесткую связь, но существующая запись в каталоге будет правильной. Хотя это тоже является ошибкой, но ее последствия менее серьезны, чем в предыдущем случае.

Журнализация

Другим средством поддержки целостности является заимствованный из систем управления базами данных прием, называемый журнализацией (иногда употребляется термин «журналирование»). Последовательность действий с объектами во время файловой операции протоколируется, и если произошел останов системы, то, имея в наличии протокол, можно осуществить откат системы назад в исходное целостное состояние, в котором она пребывала до начала операции. Подобная избыточность может стоить дорого, но она оправданна, так как в случае отказа позволяет реконструировать потерянные данные.

Для отката необходимо, чтобы для каждой протоколируемой в журнале операции существовала обратная. Например, для каталогов и реляционных СУБД это именно так. По этой причине, в отличие от СУБД, в файловых системах протоколируются не все изменения, а лишь изменения метаданных (индексных узлов, записей в каталогах и др.). Изменения в данных пользователя в протокол не заносятся. Кроме того, если протоколировать изменения пользовательских данных, то этим будет нанесен серьезный ущерб производительности системы, поскольку кэширование потеряет смысл.

Журнализация реализована в NTFS, Ext3FS, ReiserFS и других системах. Чтобы подчеркнуть сложность задачи, нужно отметить, что существуют не вполне очевидные проблемы, связанные с процедурой отката. Например, отмена одних изменений может затрагивать данные, уже использованные другими файловыми операциями. Это означает, что такие операции также должны быть отменены. Данная проблема получила название каскадного отката транзакций [Брукшир, 2001].

Проверка целостности файловой системы при помощи утилит

Если же нарушение все же произошло, то для устранения проблемы несовместимости можно прибегнуть к утилитам (*fsck, chkdsk,*

scandisk и др.), которые проверяют целостность файловой системы. Они могут запускаться после загрузки или после сбоя и осуществляют многократное сканирование разнообразных структур данных файловой системы в поисках противоречий.

Возможны также эвристические проверки. Например, нахождение индексного узла, номер которого превышает их число на диске, или поиск в пользовательских директориях файлов, принадлежащих суперпользователю.

К сожалению, приходится констатировать, что *не существует никаких средств, гарантирующих абсолютную сохранность* информации в файлах, и в тех ситуациях, когда целостность информации нужно гарантировать с высокой степенью надежности, прибегают к дорогостоящим процедурам дублирования.

Управление «плохими» блоками

Наличие дефектных блоков на диске — обычное дело. Внутри блока наряду с данными хранится контрольная сумма данных. Под «плохими» блоками обычно понимают блоки диска, для которых вычисленная контрольная сумма считываемых данных не совпадает с хранимой контрольной суммой. Дефектные блоки обычно появляются в процессе эксплуатации. Иногда они уже имеются при поставке вместе со списком, так как очень затруднительно для поставщиков сделать диск полностью свободным от дефектов. Рассмотрим два решения проблемы дефектных блоков — одно на уровне аппаратуры, другое на уровне ядра операционной системы.

Первый способ — хранить список плохих блоков в контроллере диска. Когда контроллер инициализируется, он читает плохие блоки и замещает дефектный блок резервным, помечая отображение в списке плохих блоков. Все реальные запросы будут идти к резервному блоку. Следует иметь в виду, что при этом механизм подъемника (наиболее распространенный механизм обработки запросов к блокам диска) будет работать неэффективно. Дело в том, что существует стратегия очередности обработки запросов к диску. Стратегия диктует направление движения считывающей головки диска к нужному цилиндру. Обычно резервные блоки размещаются на внешних цилиндрах. Если плохой блок расположен на внутреннем цилиндре и контроллер осуществляет подстановку прозрачным образом, то кажущееся движение головки будет осуществляться к внутреннему цилиндру, а фактическое — к внешнему. Это является нарушением стратегии и, следовательно, минусом данной схемы.

Решение на уровне операционной системы может быть следующим. Прежде всего необходимо тщательно сконструировать файл, содержащий дефектные блоки. Тогда они изымаются из списка свободных блоков. Затем нужно каким-то образом скрыть этот файл от прикладных программ.

11.8. Производительность файловой системы

Поскольку обращение к диску — операция относительно медленная, минимизация количества таких обращений — ключевая задача всех алгоритмов, работающих с внешней памятью. Наиболее типичная техника повышения скорости работы с диском — кэширование.

Кэширование

Кэш диска представляет собой буфер в оперативной памяти, содержащий ряд блоков диска (рис. 11.12). Если имеется запрос на чтение/запись блока диска, то сначала производится проверка на предмет наличия этого блока в кэше. Если блок в кэше имеется, то запрос удовлетворяется из кэша, в противном случае запрошенный блок считывается в кэш с диска. Сокращение количества дисковых операций оказывается возможным вследствие присущего операционной системе свойства локальности.

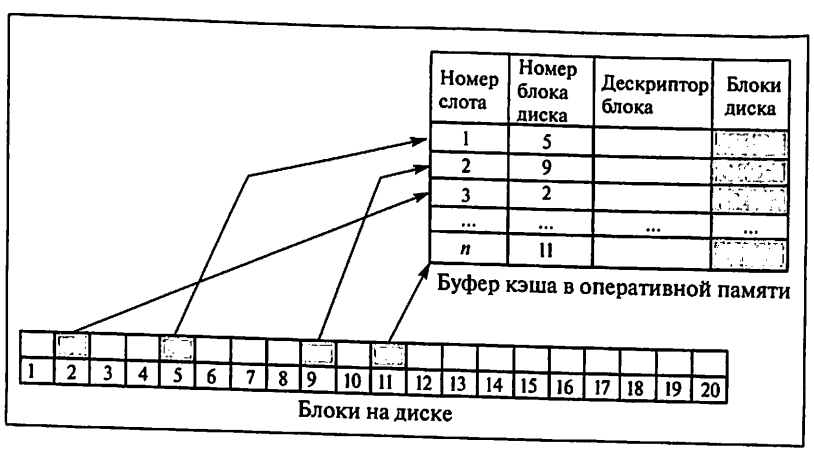

Рис. 11.12. Структура блочного кэша

Аккуратная реализация кэширования требует решения нескольких проблем.

Во-первых, емкость буфера кэша ограничена. Когда блок должен быть загружен в заполненный буфер кэша, возникает *проблема замещения блоков*, т.е. отдельные блоки должны быть удалены из него. Здесь работают те же стратегии и те же FIFO, Second Chance и LRU-алгоритмы замещения, что и при выталкивании страниц памяти.

Замещение блоков должно осуществляться с учетом их важности для файловой системы. Блоки должны быть разделены на категории, например: блоки индексных узлов, блоки косвенной адресации, блоки директорий, заполненные блоки данных и т.д., и в зависимости от принадлежности блока к той или иной категории можно применять к ним разную стратегию замещения.

Во-вторых, поскольку кэширование использует механизм отложенной записи, при котором модификация буфера не вызывает немедленной записи на диск, серьезной проблемой является «старение» информации в дисковых блоках, образы которых находятся в буферном кэше. *Несвоевременная синхронизация буфера кэша и диска* может привести к очень нежелательным последствиям в случае отказов оборудования или программного обеспечения. Поэтому стратегия и порядок отображения информации из кэша на диск должны быть тщательно продуманы.

Так, блоки, существенные для совместимости файловой системы (блоки индексных узлов, блоки косвенной адресации, блоки директорий), должны быть переписаны на диск немедленно, независимо от того, в какой части LRU-цепочки они находятся. Необходимо тщательно выбирать порядок такого переписывания.

В Unix имеется для этого вызов SYNC, который заставляет все модифицированные блоки записываться на диск немедленно. Для синхронизации содержимого кэша и диска периодически запускается фоновый процесс-демон. Кроме того, можно организовать синхронный режим работы с отдельными файлами, задаваемый при открытии файла, когда все изменения в файле немедленно сохраняются на диске.

Наконец, *проблема конкуренции процессов на доступ к блокам кэша* решается ведением списков блоков, пребывающих в различных состояниях, и отметкой о состоянии блока в его дескрипторе. Например, блок может быть заблокирован, участвовать в операции ввода-вывода, а также иметь список процессов, ожидающих освобождения данного блока.

Оптимальное размещение информации на диске

Кэширование — не единственный способ увеличения производительности системы. Другая важная техника — *сокращение количества движений считывающей головки диска* за счет разумной страте-

гии размещения информации. Например, массив индексных узлов в Unix стараются разместить на средних дорожках. Также имеет смысл размещать индексные узлы поблизости от блоков данных, на которые они ссылаются, и т.д.

Кроме того, рекомендуется периодически осуществлять дефрагментацию диска (сборку мусора), поскольку в популярных методиках выделения дисковых блоков (за исключением, может быть, FAT) принцип локальности не работает и последовательная обработка файла требует обращения к различным участкам диска.

11.9. Реализация некоторых операций над файлами

Системные вызовы, работающие с символическим именем файла. Системные вызовы, связывающие pathname с дескриптором файла

Это функции создания и открытия файла. Например, в операционной системе Unix

- fd = creat(pathname,modes);
- fd = open(pathname,flags,modes);

Другие операции над файлами, такие как чтение, запись, позиционирование головок чтения-записи, воспроизведение дескриптора файла, установка параметров ввода-вывода, определение статуса файла и закрытие файла, используют значение полученного дескриптора файла.

Рассмотрим работу системного вызова open.

Логическая файловая подсистема просматривает файловую систему в поисках файла по его имени. Она проверяет права на открытие файла и выделяет открываемому файлу запись в таблице файлов. Запись таблицы файлов содержит указатель на индексный узел открытого файла. Ядро выделяет запись в личной (закрытой) таблице в адресном пространстве процесса, выделенном процессу (таблица эта называется таблицей пользовательских дескрипторов открытых файлов) и запоминает указатель на данную запись. В роли указателя выступает дескриптор файла, возвращаемый пользователю. Запись в таблице пользовательских файлов указывает на запись в глобальной таблице файлов (рис. 11.13).

Первые три пользовательских дескриптора (0, 1 и 2) именуются дескрипторами файлов стандартного ввода, стандартного вывода и стандартного файла ошибок. Процессы в системе Unix по договоренности используют дескриптор файла стандартного ввода при чтении вводимой информации, дескриптор файла стандартного вывода

при записи выводимой информации и дескриптор стандартного файла ошибок для записи сообщений об ошибках.

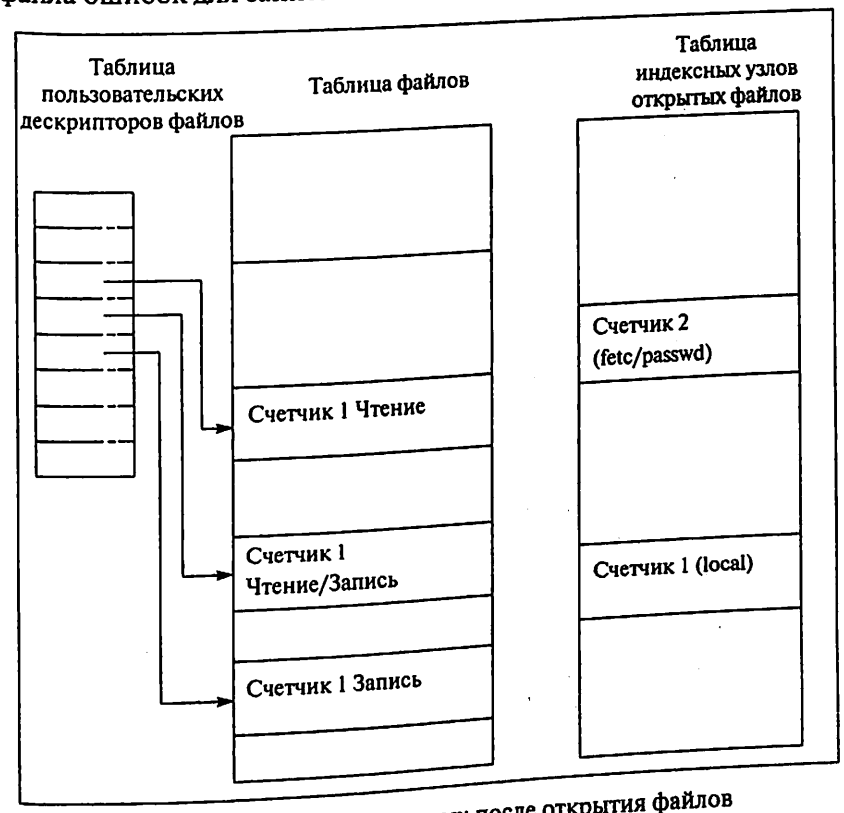

Рис. 11.13. Структуры данных после открытия файлов

Связывание файла

Системная функция link связывает файл с новым именем в структуре каталогов файловой системы, создавая для существующего индекса новую запись в каталоге. Синтаксис вызова функции link:

link(source file name, target file name);

где source file name — существующее имя файла, а target file name — новое (дополнительное) имя, присваиваемое файлу после выполнения функции link.

Сначала операционная система определяет местонахождение индекса исходного файла и увеличивает значение счетчика связей в индексном узле. Затем ядро ищет файл с новым именем; если он существует, функция link завершается неудачно, и ядро восстанавливает

прежнее значение счетчика связей, измененное ранее. В противном случае ядро находит в родительском каталоге свободную запись для файла с новым именем, записывает в нее новое имя и номер индекса исходного файла.

Удаление файла

В Unix системная функция unlink удаляет из каталога точку входа для файла. Синтаксис вызова функции unlink:

unlink(pathname).

Если удаляемое имя является последней связью файла с каким-либо каталогом, ядро в итоге освобождает все информационные блоки файла. Однако если у файла было несколько связей, он остается все еще доступным под другими именами.

Для того чтобы забрать дисковые блоки, ядро в цикле просматривает таблицу содержимого индексного узла, освобождая все блоки прямой адресации немедленно. Что касается блоков косвенной адресации, то ядро освобождает все блоки, появляющиеся на различных уровнях косвенности, рекурсивно, причем в первую очередь освобождаются блоки с меньшим уровнем.

Системные вызовы, работающие с файловым дескриптором

Открытый файл может использоваться для чтения и записи последовательностей байтов. Для этого поддерживаются два системных вызова read и write, работающие с файловым дескриптором (или handle в терминологии Microsoft), полученным при ранее выполненных системных вызовах open или creat.

Функции ввода-вывода из файла

Системный вызов read выполняет чтение обычного файла

number = read(fd,buffer,count),

где fd — дескриптор файла, возвращаемый функцией open, buffer — адрес структуры данных в пользовательском процессе, где будут размещаться считанные данные в случае успешного завершения выполнения функции read, count — количество байтов, которые пользователю нужно прочитать, number — количество фактически прочитанных байтов.

Синтаксис вызова системной функции write (писать):

number = write(fd,buffer,count),

где переменные fd, buffer, count и number имеют тот же смысл, что и для вызова системной функции read.

Алгоритм записи в обычный файл похож на алгоритм чтения из обычного файла. Однако если в файле отсутствует блок, соответствующий смещению в байтах до места, куда должна производиться запись, ядро выделяет блок и присваивает ему номер в соответствии с точным указанием места в таблице содержимого индексного узла.

Обычное использование системных функций read и write обеспечивает последовательный доступ к файлу, однако процессы могут использовать вызов системной функции lseek для указания места в файле, где будет производиться ввод-вывод, и осуществления произвольного доступа к файлу. Синтаксис вызова системной функции:

position = lseek(fd,offset,reference);

где fd — дескриптор файла, идентифицирующий файл, offset — смещение в байтах, a reference указывает, является ли значение offset смещением от начала файла, смещением от текущей позиции вывода или смещением от конца файла. Возвращаемое значение, position, является смещением в байтах до места, где будет начинаться следующая операция чтения или записи.

11.10. Современные архитектуры файловых систем

Современные операционные системы предоставляют пользователю возможность работать сразу с несколькими файловыми системами (Linux работает с Ext2fs, FAT и др.). Файловая система в традиционном понимании становится частью более общей многоуровневой структуры (рис. 11.14).

На верхнем уровне располагается так называемый диспетчер файловых систем (например, в Windows 95 этот компонент называется installable filesystem manager). Он связывает запросы прикладной программы с конкретной файловой системой.

Каждая файловая система (иногда говорят — драйвер файловой системы) на этапе инициализации регистрируется у диспетчера, сообщая ему точки входа, для последующих обращений к данной файловой системе.

Та же идея поддержки нескольких файловых систем в рамках одной операционной системы может быть реализована по-другому, например исходя из концепции виртуальной файловой системы. Виртуальная файловая система (vfs) представляет собой независимый от реализации уровень и опирается на реальные файловые сис-

темы (s5fs, ufs, FAT, NFS, FFS. Ext2fs s). При этом возникают структуры данных виртуальной файловой системы типа виртуальных индексных узлов vnode, которые обобщают индексные узлы конкретных систем.

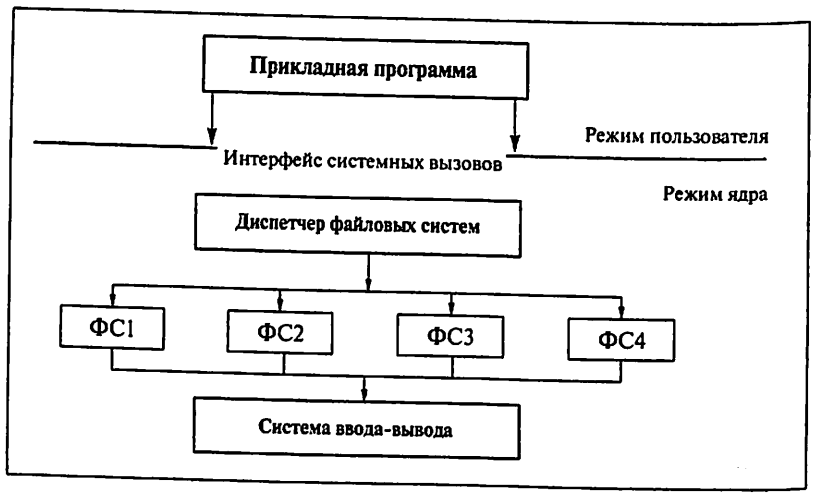

Рис. 11.14. Архитектура современной файловой системы

Глава 12
УСТРОЙСТВА ВВОДА-ВЫВОДА

Функционирование любой вычислительной системы обычно сводится к выполнению двух видов работы: обработке информации и операций по осуществлению ее ввода-вывода. Поскольку в рамках модели, принятой нами, все, что выполняется в вычислительной системе, организовано как набор процессов, эти два вида работы выполняются процессами. Процессы занимаются обработкой информации и выполнением операций ввода-вывода.

Содержание понятий «обработка информации» и «операции ввода-вывода» зависит от того, с какой точки зрения мы смотрим на них. С точки зрения программиста, под «обработкой информации» понимается выполнение команд процессора над данными, лежащими в памяти независимо от уровня иерархии — в регистрах, кэше, оперативной или вторичной памяти. Под «операциями ввода-вывода» программист понимает обмен данными между памятью и устройствами, внешними по отношению к памяти и процессору, такими как магнитные ленты, диски, монитор, клавиатура, таймер. С точки зрения операционной системы «обработкой информации» являются только операции, совершаемые процессором над данными, находящимися в памяти на уровне иерархии не ниже, чем оперативная память. Все остальное относится к «операциям ввода-вывода». Чтобы выполнять операции над данными, временно расположенными во вторичной памяти, операционная система сначала производит их подкачку в оперативную память, и лишь затем процессор совершает необходимые действия.

Объяснение того, что именно делает процессор при обработке информации, как он решает задачу и какой алгоритм выполняет, не входит в задачи нашего курса. Это скорее относится к курсу «Алгоритмы и структуры данных», с которого обычно начинается изучение информатики. Как операционная система управляет обработкой информации, мы разобрали в главе 2, в деталях описав два состояния процессов — «исполнение» и «готовность» (очереди планирования и т.д.), а также правила, по которым осуществляется перевод процессов из одного состояния в другое (алгоритмы планирования процессов).

Прежде чем говорить о работе операционной системы при осуществлении операций ввода-вывода, нам придется вспомнить неко-

торые сведения из курса «Архитектура современных ЭВМ и язык Ассемблера», чтобы понять, как осуществляется передача информации между оперативной памятью и внешним устройством и почему для подключения к вычислительной системе новых устройств ее не требуется перепроектировать.

12.1. Физические принципы организации ввода-вывода

Существует много разнообразных устройств, которые могут взаимодействовать с процессором и памятью: таймер, жесткие диски, клавиатура, дисплеи, мышь, модемы и т.д., вплоть до устройств отображения и ввода информации в авиационно-космических тренажерах. Часть этих устройств может быть встроена внутрь корпуса компьютера, часть — вынесена за его пределы и общаться с компьютером через различные линии связи: кабельные, оптоволоконные, радиорелейные, спутниковые и т.д. Конкретный набор устройств и способы их подключения определяются целями функционирования вычислительной системы, желаниями и финансовыми возможностями пользователя. Несмотря на все многообразие устройств, управление их работой и обмен информацией с ними строятся на относительно небольшом наборе принципов, которые мы постараемся разобрать в этом разделе.

Общие сведения об архитектуре компьютера

В простейшем случае процессор, память и многочисленные внешние устройства связаны большим количеством электрических соединений — *линий*, которые в совокупности принято называть локальной магистралью компьютера. Внутри локальной магистрали линии, служащие для передачи сходных сигналов и предназначенные для выполнения сходных функций, принято группировать в *шины*. При этом понятие шины включает в себя не только набор проводников, но и набор жестко заданных протоколов, определяющий перечень сообщений, который может быть передан с помощью электрических сигналов по этим проводникам. В современных компьютерах выделяют как минимум три шины:

- шину данных, состоящую из линий данных и служащую для передачи информации между процессором и памятью, процессором и устройствами ввода-вывода, памятью и внешними устройствами;
- адресную шину, состоящую из линий адреса и служащую для задания адреса ячейки памяти или указания устройства ввода-вывода, участвующих в обмене информацией;

- шину управления, состоящую из линий управления локальной магистралью и линий ее состояния, определяющих поведение локальной магистрали. В некоторых архитектурных решениях линии состояния выносятся из этой шины в отдельную шину состояния.

Количество линий, входящих в состав шины, принято называть *разрядностью* (*шириной*) этой шины.

Ширина адресной шины, например, определяет максимальный размер оперативной памяти, которая может быть установлена в вычислительной системе. Ширина шины данных определяет максимальный объем информации, которая за один раз может быть получена или передана по этой шине.

Операции обмена информацией осуществляются при одновременном участии всех шин. Рассмотрим, к примеру, действия, которые должны быть выполнены для передачи информации из процессора в память.

В простейшем случае необходимо выполнить три действия.

1. На адресной шине процессор должен выставить сигналы, соответствующие адресу ячейки памяти, в которую будет осуществляться передача информации.

2. На шину данных процессор должен выставить сигналы, соответствующие информации, которая должна быть записана в память.

3. После выполнения действий 1 и 2 на шину управления выставляются сигналы, соответствующие операции записи и работе с памятью, что приведет к занесению необходимой информации по нужному адресу.

Естественно, что приведенные выше действия являются необходимыми, но недостаточными при рассмотрении работы конкретных процессоров и микросхем памяти. Конкретные архитектурные решения могут требовать дополнительных действий: например, выставления на шину управления сигналов частичного использования шины данных (для передачи меньшего количества информации, чем позволяет ширина этой шины); выставления сигнала готовности магистрали после завершения записи в память, разрешающего приступить к новой операции, и т.д. Однако общие принципы выполнения операции записи в память остаются неизменными.

В то время как память легко можно представить себе в виде последовательности пронумерованных адресами ячеек, локализованных внутри одной микросхемы или набора микросхем, к устройствам ввода-вывода подобный подход неприменим. Внешние устройства разнесены пространственно и могут подключаться к локальной ма-

гистрали в одной точке или множестве точек, получивших название портов ввода-вывода. Тем не менее, точно так же, как ячейки памяти взаимно однозначно отображались в адресное пространство памяти, порты ввода-вывода можно взаимно однозначно отобразить в другое адресное пространство — адресное пространство ввода-вывода. При этом каждый порт ввода-вывода получает свой номер или адрес в этом пространстве. В некоторых случаях, когда адресное пространство памяти (размер которого определяется шириной адресной шины) задействовано не полностью (остались адреса, которым не соответствуют физические ячейки памяти) и протоколы работы с внешним устройством совместимы с протоколами работы с памятью, часть портов ввода-вывода может быть отображена непосредственно в адресное пространство памяти (так, например, поступают с видеопамятью дисплеев), правда, тогда эти порты уже не принято называть портами. Надо отметить, что при отображении портов в адресное пространство памяти для организации доступа к ним в полной мере могут быть задействованы существующие механизмы защиты памяти без организации специальных защитных устройств.

В ситуации прямого отображения портов ввода-вывода в адресное пространство памяти действия, необходимые для записи информации и управляющих команд в эти порты или для чтения данных из них и их состояний, ничем не отличаются от действий, производимых для передачи информации между оперативной памятью и процессором, и для их выполнения применяются те же самые команды. Если же порт отображен в адресное пространство ввода-вывода, то процесс обмена информацией инициируется специальными командами ввода-вывода и включает в себя несколько другие действия. Например, для передачи данных в порт необходимо выполнить следующее.

- На адресной шине процессор должен выставить сигналы, соответствующие адресу порта, в который будет осуществляться передача информации, в адресном пространстве ввода-вывода.
- На шину данных процессор должен выставить сигналы, соответствующие информации, которая должна быть передана в порт.
- После выполнения действий 1 и 2 на шину управления выставляются сигналы, соответствующие операции записи и работе с устройствами ввода-вывода (переключение адресных пространств!), что приведет к передаче необходимой информации в нужный порт.

Существенное отличие памяти от устройств ввода-вывода заключается в том, что занесение информации в память является окончанием операции записи, в то время как занесение информации в порт зачастую представляет собой инициализацию реального совершения операции ввода-вывода. Что именно должны делать устройства, приняв информацию через свой порт, и каким именно образом они должны поставлять информацию для чтения из порта, определяется электронными схемами устройств, получившими название контроллеров. Контроллер может непосредственно управлять отдельным устройством (например, контроллер диска), а может управлять несколькими устройствами, связываясь с их контроллерами посредством специальных шин ввода-вывода (шина IDE, шина SCSI и т.д.).

Современные вычислительные системы могут иметь разнообразную архитектуру, множество шин и магистралей, мосты для перехода информации от одной шины к другой и т.п. Для нас сейчас важными являются только следующие моменты.

- Устройства ввода-вывода подключаются к системе через порты.
- Могут существовать два адресных пространства: пространство памяти и пространство ввода-вывода.
- Порты, как правило, отображаются в адресное пространство ввода-вывода и иногда — непосредственно в адресное пространство памяти.
- Использование того или иного адресного пространства определяется типом команды, выполняемой процессором, или типом ее операндов.
- Физическим управлением устройством ввода-вывода, передачей информации через порт и выставлением некоторых сигналов на магистрали занимается контроллер устройства.

Именно единообразие подключения внешних устройств к вычислительной системе является одной из составляющих идеологии, позволяющих добавлять новые устройства без перепроектирования всей системы.

Структура контроллера устройства

Контроллеры устройств ввода-вывода весьма различны как по своему внутреннему строению, так и по исполнению (от одной микросхемы до специализированной вычислительной системы со своим процессором, памятью и т.д.), поскольку им приходится управлять совершенно разными приборами. Не вдаваясь в детали этих различий, мы выделим некоторые общие черты контроллеров, необходимые им для взаимодействия с вычислительной системой. Обычно каждый контроллер имеет по крайней мере четыре внутренних регистра, называемых регистрами *состояния*, *управления*, *входных дан-*

ных и выходных данных. Для доступа к содержимому этих регистров вычислительная система может использовать один или несколько портов, что для нас не существенно. Для простоты изложения будем считать, что каждому регистру соответствует свой порт.

Регистр состояния содержит биты, значение которых определяется состоянием устройства ввода-вывода и которые доступны только для чтения вычислительной системой. Эти биты индицируют завершение выполнения текущей команды на устройстве (*бит занятости*), наличие очередного данного в регистре выходных данных (*бит готовности данных*), возникновение ошибки при выполнении команды (*бит ошибки*) и т.д.

Регистр управления получает данные, которые записываются вычислительной системой для инициализации устройства ввода-вывода или выполнения очередной команды, а также изменения режима работы устройства. Часть битов в этом регистре может быть отведена под код выполняемой команды, часть битов будет кодировать режим работы устройства, бит *готовности команды* свидетельствует о том, что можно приступить к ее выполнению.

Регистр выходных данных служит для помещения в него данных для чтения вычислительной системой, а регистр входных данных предназначен для помещения в него информации, которая должна быть выведена на устройство. Обычно емкость этих регистров не превышает ширину линии данных (а чаще всего меньше ее), хотя некоторые контроллеры могут использовать в качестве регистров очередь FIFO для буферизации поступающей информации.

Разумеется, набор регистров и составляющих их битов приблизителен, он призван послужить нам моделью для описания процесса передачи информации от вычислительной системы к внешнему устройству и обратно, но в том или ином виде он обычно присутствует во всех контроллерах устройств.

Опрос устройств и прерывания. Исключительные ситуации и системные вызовы

Построив модель контроллера и представляя себе, что скрывается за словами «прочитать информацию из порта» и «записать информацию в порт», мы готовы к рассмотрению процесса взаимодействия устройства и процессора. Как и в предыдущих случаях, примером нам послужит команда записи, теперь уже записи или вывода данных на внешнее устройство. В нашей модели для вывода информации, помещающейся в регистр входных данных, без проверки успешности вывода процессор и контроллер должны связываться следующим образом.

1. Процессор в цикле читает информацию из порта регистра состояний и проверяет значение *бита занятости*. Если *бит занятости* установлен, то это означает, что устройство еще не завершило предыдущую операцию, и процессор уходит на новую итерацию цикла. Если *бит занятости* сброшен, то устройство готово к выполнению новой операции, и процессор переходит на следующий шаг.
2. Процессор записывает код команды вывода в порт регистра управления.
3. Процессор записывает данные в порт регистра входных данных.
4. Процессор устанавливает *бит готовности команды*. В следующих шагах процессор не задействован.
5. Когда контроллер замечает, что *бит готовности команды* установлен, он устанавливает *бит занятости*.
6. Контроллер анализирует код команды в регистре управления и обнаруживает, что это команда вывода. Он берет данные из регистра входных данных и инициирует выполнение команды.
7. После завершения операции контроллер обнуляет *бит готовности команды*.
8. При успешном завершении операции контроллер обнуляет *бит ошибки* в регистре состояния, при неудачном завершении команды — устанавливает его.
9. Контроллер сбрасывает *бит занятости*.

При необходимости вывода новой порции информации все эти шаги повторяются. Если процессор интересует, корректно или некорректно была выведена информация, то после шага 4 он должен в цикле считывать информацию из порта регистра состояний до тех пор, пока не будет сброшен *бит занятости* устройства, после чего проанализировать состояние *бита ошибки*.

Как видим, на первом шаге (и, возможно, после шага 4) процессор ожидает освобождения устройства, непрерывно опрашивая значение *бита занятости*. Такой способ взаимодействия процессора и контроллера получил название polling или, в русском переводе — *способа опроса устройств*. Если скорости работы процессора и устройства ввода-вывода примерно равны, то это не приводит к существенному уменьшению полезной работы, совершаемой процессором. Если же скорость работы устройства существенно меньше скорости процессора, то указанная техника резко снижает производительность системы и необходимо применять другой архитектурный подход.

Для того чтобы процессор не дожидался состояния готовности устройства ввода-вывода в цикле, а мог выполнять в это время другую работу, необходимо, чтобы устройство само умело сигнализировать процессору о своей готовности. Технический механизм, который позволяет внешним устройствам оповещать процессор о завершении команды вывода или команды ввода, получил название механизма прерываний.

В простейшем случае для реализации механизма прерываний необходимо к имеющимся у нас шинам локальной магистрали добавить еще одну линию, соединяющую процессор и устройства ввода-вывода, — линию прерываний. По завершении выполнения операции внешнее устройство выставляет на эту линию специальный сигнал, по которому процессор после выполнения очередной команды (или после завершения очередной итерации при выполнении цепочечных команд, т.е. команд, повторяющихся циклически со сдвигом по памяти) изменяет свое поведение. Вместо выполнения очередной команды из потока команд он частично сохраняет содержимое своих регистров и переходит на выполнение программы обработки прерывания, расположенной по заранее оговоренному адресу. При наличии только одной линии прерываний процессор при выполнении этой программы должен опросить состояние всех устройств ввода-вывода, чтобы определить, от какого именно устройства пришло прерывание (polling прерываний!), выполнить необходимые действия (например, вывести в это устройство очередную порцию информации или перевести соответствующий процесс из состояния «ожидание» в состояние «готовность») и сообщить устройству, что прерывание обработано (снять прерывание).

В большинстве современных компьютеров процессор стараются полностью освободить от необходимости опроса внешних устройств, в том числе и от определения с помощью опроса устройства, сгенерировавшего сигнал прерывания. Устройства сообщают о своей готовности процессору не напрямую, а через специальный контроллер прерываний, при этом для общения с процессором он может использовать не одну линию, а целую шину прерываний. Каждому устройству присваивается свой номер прерывания, который при возникновении прерывания контроллер прерывания заносит в свой регистр состояния и, возможно, после распознавания процессором сигнала прерывания и получения от него специального запроса выставляет на шину прерываний или шину данных для чтения процессором. Номер прерывания обычно служит индексом в специальной таблице прерываний, хранящейся по адресу, задаваемому при инициализации

вычислительной системы, и содержащей адреса программ обработки прерываний — векторы прерываний. Для распределения устройств по номерам прерываний необходимо, чтобы от каждого устройства к контроллеру прерываний шла специальная линия, соответствующая одному номеру прерывания. При наличии множества устройств такое подключение становится невозможным, и на один проводник (один номер прерывания) подключается несколько устройств. В этом случае процессор при обработке прерывания все равно вынужден заниматься опросом устройств для определения устройства, выдавшего прерывание, но в существенно меньшем объеме. Обычно при установке в систему нового устройства ввода-вывода требуется аппаратно или программно определить, каким будет номер прерывания, вырабатываемый этим устройством.

Рассматривая кооперацию процессов и взаимоисключения, мы говорили о существовании критических секций внутри ядра операционной системы, при выполнении которых необходимо исключить всякие прерывания от внешних устройств. Для запрещения прерываний, а точнее, для невосприимчивости процессора к внешним прерываниям обычно существуют специальные команды, которые могут маскировать (запрещать) все или некоторые из прерываний устройств ввода-вывода. В то же время определенные кризисные ситуации в вычислительной системе (например, неустранимый сбой в работе оперативной памяти) должны требовать ее немедленной реакции. Такие ситуации вызывают прерывания, которые невозможно замаскировать или запретить и которые поступают в процессор по специальной линии шины прерываний, называемой линией немаскируемых прерываний (NMI — Non-Maskable Interrupt).

Не все внешние устройства являются одинаково важными с точки зрения вычислительной системы («все животные равны, но некоторые равнее других»). Соответственно, некоторые прерывания являются более существенными, чем другие. Контроллер прерываний обычно позволяет устанавливать приоритеты для прерываний от внешних устройств. При почти одновременном возникновении прерываний от нескольких устройств (во время выполнения одной и той же команды процессора) процессору сообщается номер наиболее приоритетного прерывания для его обслуживания в первую очередь. Менее приоритетное прерывание при этом не пропадает, о нем процессору будет доложено после обработки более приоритетного прерывания. Более того, при обработке возникшего прерывания процессор может получить сообщение о возникновении прерывания с более высоким приоритетом и переключиться на его обработку.

Механизм обработки прерываний, по которому процессор прекращает выполнение команд в обычном режиме и, частично сохранив свое состояние, отвлекается на выполнение других действий, оказался настолько удобен, что зачастую разработчики процессоров используют его и для других целей. Хотя эти случаи и не относятся к операциям ввода-вывода, мы вынуждены упомянуть их здесь, для того чтобы их не путали с прерываниями. Похожим образом процессор обрабатывает исключительные ситуации и программные прерывания.

Для внешних прерываний характерны следующие особенности.

- Внешнее прерывание обнаруживается процессором между выполнением команд (или между итерациями в случае выполнения цепочечных команд).
- Процессор при переходе на обработку прерывания сохраняет часть своего состояния перед выполнением *следующей* команды.

Прерывания происходят *асинхронно* с работой процессора и *непредсказуемо*, программист никоим образом не может предугадать, в каком именно месте работы программы произойдет прерывание.

Исключительные ситуации возникают во время выполнения процессором команды. К их числу относятся ситуации переполнения, деления на ноль, обращения к отсутствующей странице памяти и т.д. Для исключительных ситуаций характерно следующее.

- Исключительные ситуации обнаруживаются процессором во время выполнения команд.
- Процессор при переходе на выполнение обработки исключительной ситуации сохраняет часть своего состояния перед выполнением текущей команды.
- Исключительные ситуации возникают *синхронно* с работой процессора, но *непредсказуемо* для программиста, если только тот специально не заставил процессор делить некоторое число на ноль.

Программные прерывания возникают после выполнения специальных команд, как правило, для выполнения привилегированных действий внутри системных вызовов. Программные прерывания имеют следующие свойства.

- Программное прерывание происходит в результате выполнения специальной команды.
- Процессор при выполнении программного прерывания сохраняет свое состояние перед выполнением *следующей* команды.
- Программные прерывания, естественно, возникают синхронно с работой процессора и *абсолютно предсказуемы* программистом.

Надо сказать, что реализация похожих механизмов обработки внешних прерываний, исключительных ситуаций и программных прерываний лежит целиком на совести разработчиков процессоров. Существуют вычислительные системы, где все три ситуации обрабатываются по-разному.

Прямой доступ к памяти (Direct Memory Access — DMA)

Использование механизма прерываний позволяет разумно загружать процессор в то время, когда устройство ввода-вывода занимается своей работой. Однако запись или чтение большого количества информации из адресного пространства ввода-вывода (например, с диска) приводят к большому количеству операций ввода-вывода, которые должен выполнять процессор. Для освобождения процессора от операций последовательного вывода данных из оперативной памяти или последовательного ввода в нее был предложен механизм прямого доступа внешних устройств к памяти — ПДП или Direct Memory Access — DMA.

Давайте кратко рассмотрим, как работает этот механизм.

Для того чтобы какое-либо устройство, кроме процессора, могло записать информацию в память или прочитать ее из памяти, необходимо, чтобы это устройство могло забрать у процессора управление локальной магистралью для выставления соответствующих сигналов на шины адреса, данных и управления. Для централизации эти обязанности обычно возлагаются не на каждое устройство в отдельности, а на специальный контроллер — контроллер прямого доступа к памяти. Контроллер прямого доступа к памяти имеет несколько спаренных линий — каналов DMA, которые могут подключаться к различным устройствам. Перед началом использования прямого доступа к памяти этот контроллер необходимо запрограммировать, записав в его порты информацию о том, какой канал или каналы предполагается задействовать, какие операции они будут совершать, какой адрес памяти является начальным для передачи информации и какое количество информации должно быть передано. Получив по одной из линий — каналов DMA, сигнал запроса на передачу данных от внешнего устройства, контроллер по шине управления сообщает процессору о желании взять на себя управление локальной магистралью. Процессор, возможно, через некоторое время, необходимое для завершения его действий с магистралью, передает управление ею контроллеру DMA, известив его специальным сигналом. Контроллер DMA выставляет на адресную шину адрес памяти для передачи очередной порции информации и по второй линии канала прямого до-

ступа к памяти сообщает устройству о готовности магистрали к передаче данных. После этого, используя шину данных и шину управления, контроллер DMA, устройство ввода-вывода и память осуществляют процесс обмена информацией. Затем контроллер прямого доступа к памяти извещает процессор о своем отказе от управления магистралью, и тот берет руководящие функции на себя. При передаче большого количества данных весь процесс повторяется циклически.

При прямом доступе к памяти процессор и контроллер DMA по очереди управляют локальной магистралью. Это, конечно, несколько снижает производительность процессора, так как при выполнении некоторых команд или при чтении очередной порции команд во внутренний кэш он должен поджидать освобождения магистрали, но в целом производительность вычислительной системы существенно возрастает.

При подключении к системе нового устройства, которое умеет использовать прямой доступ к памяти, обычно необходимо программно или аппаратно задать номер канала DMA, к которому будет приписано устройство. В отличие от прерываний, где один номер прерывания мог соответствовать нескольким устройствам, каналы DMA всегда находятся в монопольном владении устройств.

Логические принципы организации ввода-вывода

Рассмотренные в предыдущем разделе физические механизмы взаимодействия устройств ввода-вывода с вычислительной системой позволяют понять, почему разнообразные внешние устройства легко могут быть добавлены в существующие компьютеры. Все, что необходимо сделать пользователю при подключении нового устройства, — это отобразить порты устройства в соответствующее адресное пространство, определить, какой номер будет соответствовать прерыванию, генерируемому устройством, и, если нужно, закрепить за устройством некоторый канал DMA. Для нормального функционирования hardware этого будет достаточно. Однако мы до сих пор ничего не сказали о том, как должна быть построена подсистема управления вводом-выводом в операционной системе для легкого и безболезненного добавления новых устройств и какие функции вообще обычно на нее возлагаются.

Структура системы ввода-вывода

Если поручить неподготовленному пользователю сконструировать систему ввода-вывода, способную работать со всем множеством

внешних устройств, то, скорее всего, он окажется в ситуации, в которой находились биологи и зоологи до появления трудов Линнея. Все устройства разные, отличаются по выполняемым функциям и своим характеристикам, и кажется, что принципиально невозможно создать систему, которая без больших постоянных переделок позволяла бы охватывать все многообразие видов. Вот перечень лишь нескольких направлений (далеко не полный), по которым различаются устройства:

- Скорость обмена информацией может варьироваться в диапазоне от нескольких байтов в секунду (клавиатура) до нескольких гигабайтов в секунду (сетевые карты).
- Одни устройства могут использоваться несколькими процессами параллельно (являются разделяемыми), в то время как другие требуют монопольного захвата процессом.
- Устройства могут запоминать выведенную информацию для ее последующего ввода или не обладать этой функцией. Устройства, запоминающие информацию, в свою очередь, могут дифференцироваться по формам доступа к сохраненной информации: обеспечивать к ней последовательный доступ в жестко заданном порядке или уметь находить и передавать только необходимую порцию данных.
- Часть устройств умеет передавать данные только по одному байту последовательно (символьные устройства), а часть устройств умеет передавать блок байтов как единое целое (блочные устройства).
- Существуют устройства, предназначенные только для ввода информации, устройства, предназначенные только для вывода информации, и устройства, которые могут выполнять и ввод, и вывод.

В области технического обеспечения удалось выделить несколько основных принципов взаимодействия внешних устройств с вычислительной системой, т.е. создать единый интерфейс для их подключения, возложив все специфические действия на контроллеры самих устройств. Тем самым конструкторы вычислительных систем переложили все хлопоты, связанные с подключением внешней аппаратуры, на разработчиков самой аппаратуры, заставляя их придерживаться определенного стандарта.

Похожий подход оказался продуктивным и в области программного подключения устройств ввода-вывода. Подобно тому как Линнею удалось заложить основы систематизации знаний о растительном и животном мире, разделив все живое в природе на относи-

тельно небольшое число классов и отрядов, мы можем разделить устройства на относительно небольшое число типов, отличающихся по набору операций, которые могут быть ими выполнены, считая все остальные различия несущественными. Мы можем затем специфицировать интерфейсы между ядром операционной системы, осуществляющим некоторую общую политику ввода-вывода, и программными частями, непосредственно управляющими устройствами, для каждого из таких типов. Более того, разработчики операционных систем получают возможность освободиться от написания и тестирования этих специфических программных частей, получивших название *драйверов*, передав эту деятельность производителям самих внешних устройств. Фактически мы приходим к использованию принципа уровневого или слоеного построения системы управления вводом-выводом для операционной системы (рис. 12.1).

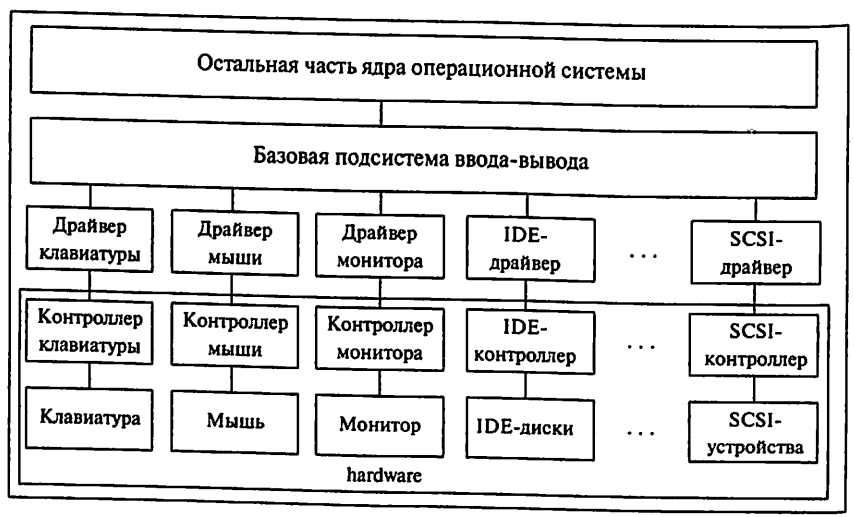

Рис. 12.1. Структура системы ввода-вывода

Два нижних уровня этой слоеной системы составляет hardware: сами устройства, непосредственно выполняющие операции, и их контроллеры, служащие для организации совместной работы устройств и остальной вычислительной системы. Следующий уровень составляют драйверы устройств ввода-вывода, скрывающие от разработчиков операционных систем особенности функционирования конкретных приборов и обеспечивающие четко определенный интерфейс между hardware и вышележащим уровнем — уровнем базовой подсистемы ввода-вывода, которая, в свою очередь, предоставляет механизм взаимодействия между драйверами и программной частью вычислительной системы в целом.

Систематизация внешних устройств и интерфейс между базовой подсистемой ввода-вывода и драйверами

Как и система видов Линнея, система типов устройств является далеко не полной и не строго выдержанной. Устройства обычно принято разделять по преобладающему типу интерфейса на следующие виды:

- символьные (клавиатура, модем, терминал и т.п.);
- блочные (магнитные и оптические диски и ленты и т.д.);
- сетевые (сетевые карты);
- все остальные (таймеры, графические дисплеи, телевизионные устройства, видеокамеры и т.п.).

Такое деление является весьма условным. В одних операционных системах сетевые устройства могут не выделяться в отдельную группу, в некоторых других — отдельные группы составляют звуковые устройства и видеоустройства и т.д. Некоторые группы в свою очередь могут разбиваться на подгруппы: подгруппа жестких дисков, подгруппа мышек, подгруппа принтеров. Нас такие детали не интересуют. Мы не ставим перед собой цель осуществить систематизацию всех возможных устройств, которые могут быть подключены к вычислительной системе. Единственное, для чего нам понадобится эта классификация, так это для иллюстрации того положения, что устройства могут быть разделены на группы по выполняемым ими функциям, и для понимания функций драйверов, и интерфейса между ними и базовой подсистемой ввода-вывода.

Для этого мы рассмотрим только две группы устройств: символьные и блочные. Как уже упоминалось в предыдущем разделе, символьные устройства — это устройства, которые умеют передавать данные только последовательно, байт за байтом, а блочные устройства — это устройства, которые могут передавать блок байтов как единое целое.

К символьным устройствам обычно относятся устройства ввода информации, которые спонтанно генерируют входные данные: клавиатура, мышь, модем, джойстик. К ним же относятся и устройства вывода информации, для которых характерно представление данных в виде линейного потока: принтеры, звуковые карты и т.д. По своей природе символьные устройства обычно умеют совершать две общие операции: ввести символ (байт) и вывести символ (байт) — get и put.

Для блочных устройств, таких как магнитные и оптические диски, ленты и т.п., естественными являются операции чтения и записи блока информации — read и write, а также, для устройств прямого доступа, операция поиска требуемого блока информации — seek.

Драйверы символьных и блочных устройств должны предоставлять базовой подсистеме ввода-вывода функции для осуществления описанных общих операций. Помимо общих операций некоторые устройства могут выполнять операции специфические, свойственные только им, например звуковые карты умеют увеличивать или уменьшать среднюю громкость звучания, дисплеи умеют изменять свою разрешающую способность. Для выполнения таких специфических действий в интерфейс между драйвером и базовой подсистемой ввода-вывода обычно входит еще одна функция, позволяющая непосредственно передавать драйверу устройства произвольную команду с произвольными параметрами, что позволяет задействовать любую возможность драйвера без изменения интерфейса. В операционной системе Unix такая функция получила название ioctl (от input-output control).

Помимо функций read, write, seek (для блочных устройств), get, put (для символьных устройств) и ioctl, в состав интерфейса обычно включают еще следующие функции:

- функцию инициализации или повторной инициализации работы драйвера и устройства — open;
- функцию временного завершения работы с устройством (может, например, вызывать отключение устройства) — close;
- функцию опроса состояния устройства (если по каким-либо причинам работа с устройством производится методом опроса его состояния, например, в операционных системах Windows NT и Windows 9x так построена работа с принтерами через параллельный порт) — poll;
- функцию останова драйвера, которая вызывается при останове операционной системы или выгрузке драйвера из памяти, halt.

Существует еще ряд действий, выполнение которых может быть возложено на драйвер, но поскольку, как правило, это действия базовой подсистемы ввода-вывода.

Приведенные выше названия функций, конечно, являются условными и могут меняться от одной операционной системы к другой, но действия, выполняемые драйверами, характерны для большинства операционных систем, и соответствующие функции присутствуют в интерфейсах к ним.

Функции базовой подсистемы ввода-вывода

Базовая подсистема ввода-вывода служит посредником между процессами вычислительной системы и набором драйверов. Системные вызовы для выполнения операций ввода-вывода трансформируются ею в вызовы функций необходимого драйвера устройства. Однако обязанности базовой подсистемы не сводятся к выполнению только действий трансляции общего системного вызова в обращение к частной функции драйвера. Базовая подсистема предоставляет вычислительной системе такие услуги, как поддержка блокирующихся, неблокирующихся и асинхронных системных вызовов, буферизация и кэширование входных и выходных данных, осуществление spooling'а и монопольного захвата внешних устройств, обработка ошибок и прерываний, возникающих при операциях ввода-вывода, планирование последовательности запросов на выполнение этих операций. Давайте остановимся на этих услугах подробнее.

Блокирующиеся, неблокирующиеся и асинхронные системные вызовы

Все системные вызовы, связанные с осуществлением операций ввода-вывода, можно разбить на три группы по способам реализации взаимодействия процесса и устройства ввода-вывода.

- К первой, наиболее привычной для большинства программистов группе относятся блокирующиеся системные вызовы. Как следует из самого названия, применение такого вызова приводит к блокировке инициировавшего его процесса, т.е. процесс переводится операционной системой из состояния «исполнение» в состояние «ожидание». Завершив выполнение всех операций ввода-вывода, предписанных системным вызовом, операционная система переводит процесс из состояния «ожидание» в состояние «готовность». После того как процесс будет снова выбран для «исполнения», в нем произойдет окончательный возврат из системного вызова. Типичным для применения такого системного вызова является случай, когда процессу необходимо получить от устройства строго определенное количество данных, без которых он не может выполнять работу далее.
- Ко второй группе относятся неблокирующиеся системные вызовы. Их название не совсем точно отражает суть дела. В простейшем случае процесс, применивший неблокирующийся вызов, не переводится в состояние «ожидание» вообще. Системный вызов возвращается немедленно, выполнив предписанные ему операции ввода-вывода полностью, частично или не выполнив совсем,

в зависимости от текущей ситуации (состояния устройства, наличия данных и т.д.). В более сложных ситуациях процесс может блокироваться, но условием его разблокирования является завершение всех необходимых операций или окончание некоторого промежутка времени. Типичным случаем применения неблокирующегося системного вызова может являться периодическая проверка на поступление информации с клавиатуры при выполнении трудоемких расчетов.

- К третьей группе относятся асинхронные системные вызовы. Процесс, использовавший асинхронный системный вызов, никогда в нем не блокируется. Системный вызов инициирует выполнение необходимых операций ввода-вывода и немедленно возвращается, после чего процесс продолжает свою регулярную деятельность. Об окончании завершения операции ввода-вывода операционная система впоследствии информирует процесс изменением значений некоторых переменных, передачей ему сигнала или сообщения или каким-либо иным способом. Необходимо четко понимать разницу между неблокирующимися и асинхронными вызовами. Неблокирующийся системный вызов для выполнения операции read вернется немедленно, но может прочитать запрошенное количество байтов, меньшее количество или вообще ничего. Асинхронный системный вызов для этой операции также вернется немедленно, но требуемое количество байтов рано или поздно будет прочитано в полном объеме.

Буферизация и кэширование

Под *буфером* обычно понимается некоторая область памяти для запоминания информации при обмене данных между двумя устройствами, двумя процессами или процессом и устройством. Обмен информацией между двумя процессами относится к области кооперации процессов, который мы подробно рассмотрели. Здесь нас будет интересовать использование буферов в том случае, когда одним из участников обмена является внешнее устройство. Существует три причины, приводящие к использованию буферов в базовой подсистеме ввода-вывода.

Первая причина буферизации — это разные скорости приема и передачи информации, которыми обладают участники обмена. Рассмотрим, например, случай передачи потока данных от клавиатуры к модему. Скорость, с которой поставляет информацию клавиатура, определяется скоростью набора текста человеком и обычно существенно меньше скорости передачи данных модемом. Для того

чтобы не занимать модем на все время набора текста, делая его недоступным для других процессов и устройств, целесообразно накапливать введенную информацию в буфере или нескольких буферах достаточного размера и отсылать ее через модем после заполнения буферов.

Вторая причина буферизации — это разные объемы данных, которые могут быть приняты или получены участниками обмена единовременно. Возьмем другой пример. Пусть информация поставляется модемом и записывается на жесткий диск. Помимо обладания разными скоростями совершения операций, модем и жесткий диск представляют собой устройства разного типа. Модем является символьным устройством и выдает данные байт за байтом, в то время как диск является блочным устройством и для проведения операции записи для него требуется накопить необходимый блок данных в буфере. Здесь также можно применять более одного буфера. После заполнения первого буфера модем начинает заполнять второй, одновременно с записью первого на жесткий диск. Поскольку скорость работы жесткого диска в тысячи раз больше, чем скорость работы модема, к моменту заполнения второго буфера операция записи первого будет завершена и модем снова сможет заполнять первый буфер одновременно с записью второго на диск.

Третья причина буферизации связана с необходимостью копирования информации из приложений, осуществляющих ввод-вывод, в буфер ядра операционной системы и обратно. Допустим, что некоторый пользовательский процесс пожелал вывести информацию из своего адресного пространства на внешнее устройство. Для этого он должен выполнить системный вызов с обобщенным названием write, передав в качестве параметров адрес области памяти, где расположены данные, и их объем. Если внешнее устройство временно занято, то возможна ситуация, когда к моменту его освобождения содержимое нужной области окажется испорченным (например, при использовании асинхронной формы системного вызова). Чтобы избежать возникновения подобных ситуаций, проще всего в начале работы системного вызова скопировать необходимые данные в буфер ядра операционной системы, постоянно находящийся в оперативной памяти, и выводить их на устройство из этого буфера.

Под словом *кэш* (cash — «наличные»), этимологию которого мы не будем здесь рассматривать, обычно понимают область быстрой памяти, содержащую копию данных, расположенных где-либо в более медленной памяти, предназначенную для ускорения работы вычислительной системы. Мы с вами сталкивались с этим понятием

при рассмотрении иерархии памяти. В базовой подсистеме ввода-вывода не следует смешивать два понятия — буферизацию и кэширование, хотя зачастую для выполнения этих функций отводится одна и та же область памяти. Буфер часто содержит единственный набор данных, существующий в системе, в то время как кэш, по определению, содержит копию данных, существующих где-нибудь еще. Например, буфер, используемый базовой подсистемой для копирования данных из пользовательского пространства процесса при выводе на диск, может в свою очередь применяться как кэш для этих данных, если операции модификации и повторного чтения данного блока выполняются достаточно часто.

Функции буферизации и кэширования не обязательно должны быть локализованы в базовой подсистеме ввода-вывода. Они могут быть частично реализованы в драйверах и даже в контроллерах устройств, скрытно по отношению к базовой подсистеме.

Spooling и захват устройств

О понятии *spooling* мы говорили в первой главе учебника как о механизме, впервые позволившем совместить реальные операции ввода-вывода одного задания с выполнением другого задания. Теперь мы можем определить это понятие более точно. Под словом «spool» мы подразумеваем буфер, содержащий входные или выходные данные для устройства, на котором следует избегать чередования его использования различными процессами. Правда, в современных вычислительных системах spool для ввода данных практически не используется, а в основном предназначен для накопления выходной информации.

Рассмотрим в качестве внешнего устройства принтер. Хотя принтер не может печатать информацию, поступающую одновременно от нескольких процессов, может оказаться желательным разрешить процессам совершать вывод на принтер параллельно. Для этого операционная система вместо передачи информации напрямую на принтер накапливает выводимые данные в буферах на диске, организованных в виде отдельного spool-файла для каждого процесса. После завершения некоторого процесса соответствующий ему spool-файл ставится в очередь для реальной печати. Механизм, обеспечивающий подобные действия, и получил название «spooling».

В некоторых операционных системах вместо использования spooling для устранения race condition применяется механизм монопольного захвата устройств процессами. Если устройство свободно, то один из процессов может получить его в монопольное распоряжение. При этом все другие процессы при попытке осуществления операций над этим устройством будут либо блокированы (переведены в состояние «ожидание»), либо получат информацию о невозможности выполнения операции до тех пор, пока процесс, захвативший устройство, не завершится или явно не сообщит операционной системе о своем отказе от его использования.

Обеспечение spooling и механизма захвата устройств является прерогативой базовой подсистемы ввода-вывода.

Обработка прерываний и ошибок

Если при работе с внешним устройством вычислительная система не пользуется методом опроса его состояния, а задействует механизм прерываний, то при возникновении прерывания, как мы уже говорили раньше, процессор, частично сохранив свое состояние, передает управление специальной программе обработки прерывания. Мы уже рассматривали действия операционной системы над процессами, происходящими при возникновении прерывания, где после возникновения прерывания осуществлялись следующие действия: сохранение контекста, обработка прерывания, планирование использования процессора, восстановление контекста. Тогда мы обращали больше внимания на действия, связанные с сохранением и восстановлением контекста и планированием использования процессора. Теперь давайте подробнее остановимся на том, что скрывается за словами «обработка прерывания».

Одна и та же процедура обработки прерывания может применяться для нескольких устройств ввода-вывода (например, если эти устройства используют одну линию прерываний, идущую от них к контроллеру прерываний), поэтому первое действие собственно программы обработки состоит в определении того, какое именно устройство выдало прерывание. Зная устройство, мы можем выявить процесс, который инициировал выполнение соответствующей операции. Поскольку прерывание возникает как при удачном, так и при неудачном ее выполнении, следующее, что мы должны сделать, — это определить успешность завершения операции, проверив значение *бита ошибки* в регистре состояния устройства. В некоторых случаях операционная система может предпринять определенные действия, направленные на компенсацию возникшей ошибки. Например, в случае возникновения ошибки чтения с гибкого диска можно попробовать несколько раз повторить выполнение команды. Если компенсация ошибки невозможна, то операционная система впоследствии известит об этом процесс, запросивший выполнение

операции (например, специальным кодом возврата из системного вызова). Если этот процесс был заблокирован до выполнения завершившейся операции, то операционная система переводит его в состояние «готовность». При наличии других неудовлетворенных запросов к освободившемуся устройству операционная система может инициировать выполнение следующего запроса, одновременно известив устройство, что прерывание обработано. На этом, собственно, обработка прерывания заканчивается, и система может приступать к планированию использования процессора.

Действия по обработке прерывания и компенсации возникающих ошибок могут быть частично переложены на плечи соответствующего драйвера. Для этого в состав интерфейса между драйвером и базовой подсистемой ввода-вывода добавляют еще одну функцию — функцию обработки прерывания intr.

Планирование запросов

При использовании неблокирующегося системного вызова может оказаться, что нужное устройство уже занято выполнением некоторых операций. В этом случае неблокирующийся вызов может немедленно вернуться, не выполнив запрошенных команд. При организации запроса на совершение операций ввода-вывода с помощью блокирующегося или асинхронного вызова занятость устройства приводит к необходимости постановки запроса в очередь к данному устройству. В результате с каждым устройством оказывается связан список неудовлетворенных запросов процессов, находящихся в состоянии «ожидания», и запросов, выполняющихся в асинхронном режиме. Состояние «ожидание» расщепляется на набор очередей процессов, дожидающихся различных устройств ввода-вывода или ожидающих изменения состояний различных объектов — семафоров, очередей сообщений, условных переменных в мониторах и т.д.

После завершения выполнения текущего запроса операционная система (по ходу обработки возникшего прерывания) должна решить, какой из запросов в списке должен быть удовлетворен следующим, и инициировать его исполнение. Точно так же, как для выбора очередного процесса на исполнение из списка готовых нам приходилось осуществлять краткосрочное планирование процессов, здесь нам необходимо осуществлять планирование применения устройств, пользуясь каким-либо алгоритмом этого планирования. Критерии и цели такого планирования мало отличаются от критериев и целей планирования процессов.

Задача планирования использования устройства обычно возлагается на базовую подсистему ввода-вывода, однако для некоторых устройств лучшие алгоритмы планирования могут быть тесно связаны с деталями их внутреннего функционирования. В таких случаях операция планирования переносится внутрь драйвера соответствующего устройства, так как эти детали скрыты от базовой подсистемы. Для этого в интерфейс драйвера добавляется еще одна специальная функция, которая осуществляет выбор очередного запроса, — функция strategy.

Алгоритмы планирования запросов к жесткому диску

Прежде чем приступить к непосредственному изложению самих алгоритмов, давайте вспомним внутреннее устройство жесткого диска и определим, какие параметры запросов мы можем использовать для планирования.

Строение жесткого диска и параметры планирования

Современный жесткий магнитный диск представляет собой набор круглых пластин, находящихся на одной оси и покрытых с одной или двух сторон специальным магнитным слоем (рис. 12.2). Около каждой рабочей поверхности каждой пластины расположены магнитные головки для чтения и записи информации. Эти головки присоединены к специальному рычагу, который может перемещать весь блок головок над поверхностями пластин как единое целое. Поверхности пластин разделены на концентрические кольца, внутри которых, собственно, и может храниться информация. Набор концентрических колец на всех пластинах для одного положения головок (т.е. все кольца, равноудаленные от оси) образует цилиндр. Каждое кольцо внутри цилиндра получило название дорожки (по одной или две дорожки на каждую пластину). Все дорожки делятся на равное число секторов. Количество дорожек, цилиндров и секторов может варьироваться от одного жесткого диска к другому в достаточно широких пределах. Как правило, сектор является минимальным объемом информации, которая может быть прочитана с диска за один раз.

При работе диска набор пластин вращается вокруг своей оси с высокой скоростью, подставляя по очереди под головки соответствующих дорожек все их сектора. Номер сектора, номер дорожки и номер цилиндра однозначно определяют положение данных на жестком диске и, наряду с типом совершаемой операции — чтение или запись, полностью характеризуют часть запроса, связанную с устройством, при обмене информацией в объеме одного сектора.

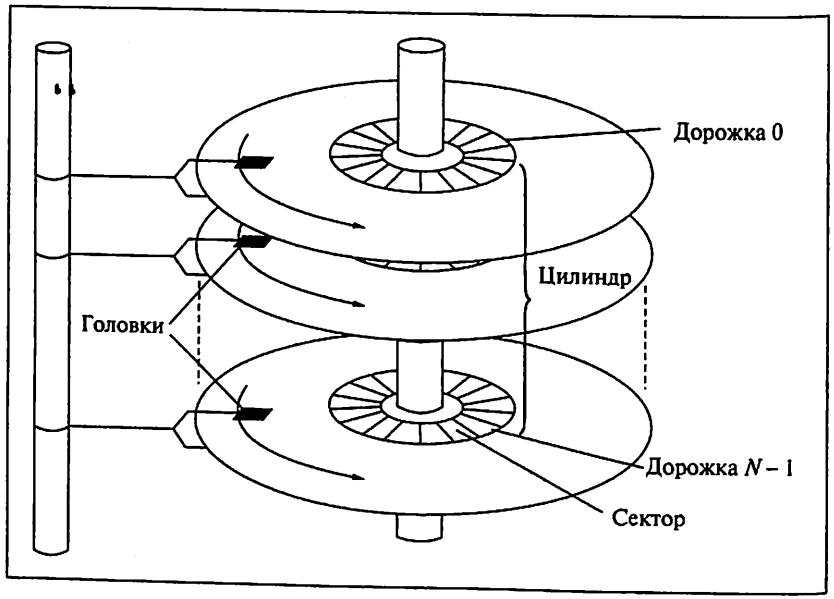

Рис. 12.2. Схема жесткого диска

При планировании использования жесткого диска естественным параметром планирования является время, которое потребуется для выполнения очередного запроса. Время, необходимое для чтения или записи определенного сектора на определенной дорожке определенного цилиндра, можно разделить на две составляющие: время обмена информацией между магнитной головкой и компьютером, которое обычно не зависит от положения данных и определяется скоростью их передачи (transfer speed), и время, необходимое для позиционирования головки над заданным сектором, — время позиционирования (positioning time). Время позиционирования, в свою очередь, состоит из времени, необходимого для перемещения головок на нужный цилиндр, — времени поиска (seek time) и времени, которое требуется для того, чтобы нужный сектор довернулся под головку, — задержки на вращение (rotational latency). Времена поиска пропорциональны разнице между номерами цилиндров предыдущего и планируемого запросов, и их легко сравнивать. Задержка на вращение определяется довольно сложными соотношениями между номерами цилиндров и секторов предыдущего и планируемого запросов и скоростями вращения диска и перемещения головок. Без знания соотношения этих скоростей сравнение становится невозможным. Поэтому естественно, что набор параметров планирования сокращается до времени поиска различных запросов, определяемого

текущим положением головки и номерами требуемых цилиндров, а разницей в задержках на вращение пренебрегают.

Алгоритм First Come First Served (FCFS)

Простейшим алгоритмом, к которому мы уже должны были привыкнуть, является алгоритм First Come First Served (FCFS) — первым пришел, первым обслужен. Все запросы организуются в очередь FIFO и обслуживаются в порядке поступления. Алгоритм прост в реализации, но может приводить к достаточно длительному общему времени обслуживания запросов. Рассмотрим пример. Пусть у нас на диске из 100 цилиндров (от 0 до 99) есть следующая очередь запросов: 23, 67, 55, 14, 31, 7, 84, 10 и головки в начальный момент находятся на 63-м цилиндре. Тогда положение головок будет меняться следующим образом:

63 23 67 55 14 31 7 84 10

и всего головки переместятся на 329 цилиндров. Неэффективность алгоритма хорошо иллюстрируется двумя последними перемещениями с 7 цилиндра через весь диск на 84 цилиндр и затем опять через весь диск на цилиндр 10. Простая замена порядка двух последних перемещений (7 10 84) позволила бы существенно сократить общее время обслуживания запросов. Поэтому давайте перейдем к рассмотрению другого алгоритма.

Алгоритм Short Seek Time First (SSTF)

Как мы убедились, достаточно разумным является первоочередное обслуживание запросов, данные для которых лежат рядом с текущей позицией головок, а уж затем далеко отстоящих. Алгоритм Short Seek Time First (SSTF) — короткое время поиска первым — как раз и исходит из этой позиции. Для очередного обслуживания будем выбирать запрос, данные для которого лежат наиболее близко к текущему положению магнитных головок. Естественно, что при наличии равноудаленных запросов решение о выборе между ними может приниматься исходя из различных соображений, например по алгоритму FCFS. Для предыдущего примера алгоритм даст такую последовательность положений головок:

63 67 55 31 23 14 10 7 84

и всего головки переместятся на 141 цилиндр. Заметим, что наш алгоритм похож на алгоритм SJF планирования процессов, если за аналог оценки времени очередного CPU burst процесса выбирать

расстояние между текущим положением головки и положением, необходимым для удовлетворения запроса. И точно так же, как алгоритм SJF, он может приводить к длительному откладыванию выполнения какого-либо запроса. Необходимо вспомнить, что запросы в очереди могут появляться в любой момент времени. Если у нас все запросы, кроме одного, постоянно группируются в области с большими номерами цилиндров, то этот один запрос может находиться в очереди неопределенно долго.

Точный алгоритм SJF являлся оптимальным для заданного набора процессов с заданными временами CPU burst. Очевидно, что алгоритм SSTF не является оптимальным. Если мы перенесем обслуживание запроса 67-го цилиндра в промежуток между запросами 7-го и 84-го цилиндров, мы уменьшим общее время обслуживания. Это наблюдение приводит нас к идее целого семейства других алгоритмов — алгоритмов сканирования.

Алгоритмы сканирования (SCAN, C-SCAN, LOOK, C-LOOK)

В простейшем из алгоритмов сканирования — SCAN — головки постоянно перемещаются от одного края диска до другого, по ходу дела обслуживая все встречающиеся запросы. По достижении другого края направление движения меняется, и все повторяется снова. Пусть в предыдущем примере в начальный момент времени головки двигаются в направлении уменьшения номеров цилиндров. Тогда мы и получим порядок обслуживания запросов, рассмотренный в конце предыдущего раздела. Последовательность перемещения головок выглядит следующим образом:

63 55 31 23 14 10 7 0 67 84

и всего головки переместятся на 147 цилиндров.

Если мы знаем, что обслужили последний попутный запрос в направлении движения головок, то мы можем не доходить до края диска, а сразу изменить направление движения на обратное:

63 55 31 23 14 10 7 67 84

и всего головки переместятся на 133 цилиндра. Полученная модификация алгоритма SCAN получила название LOOK.

Допустим, что к моменту изменения направления движения головки в алгоритме SCAN, т.е. когда головка достигла одного из краев диска, у этого края накопилось большое количество новых запросов, на обслуживание которых будет потрачено достаточно много времени (не забываем, что надо не только перемещать головку, но еще и передавать прочитанные данные!). Тогда запросы, относящиеся к другому краю диска и поступившие раньше, будут ждать обслуживания несправедливо долго. Для сокращения времени ожидания запросов применяется другая модификация алгоритма SCAN — циклическое сканирование. Когда головка достигает одного из краев диска, она без чтения попутных запросов (иногда существенно быстрее, чем при выполнении обычного поиска цилиндра) перемещается на другой край, откуда вновь начинает движение в прежнем направлении. Для этого алгоритма, получившего название C-SCAN, последовательность перемещений будет выглядеть так:

63 55 31 23 14 10 7 0 99 84 67

По аналогии с алгоритмом LOOK для алгоритма SCAN можно предложить и алгоритм C-LOOK для алгоритма C-SCAN:

63 55 31 23 14 10 7 84 67

Глава 13
БЕЗОПАСНОСТЬ В ОПЕРАЦИОННЫХ СИСТЕМАХ

В октябре 1988 г. в США произошло событие, названное специалистами крупнейшим нарушением безопасности американских компьютерных систем из когда-либо случавшихся. 23-летний студент выпускного курса Корнельского университета Роберт Т. Моррис запустил в компьютерной сети ARPANET программу, представлявшую собой редко встречающуюся разновидность компьютерных вирусов — сетевых «червей». В результате атаки был полностью или частично заблокирован ряд общенациональных компьютерных сетей, в частности Internet, CSnet, NSFnet, BITnet, ARPANET и несекретная военная сеть Milnet. В итоге вирус поразил более 6200 компьютерных систем по всей Америке, включая системы многих крупнейших университетов, институтов, правительственных лабораторий, частных фирм, военных баз, клиник, агентства NASA. Общий ущерб от этой атаки оценивается специалистами минимум в 100 млн долл. Р. Моррис был исключен из университета с правом повторного поступления через год и приговорен судом к штрафу в 270 тыс. долл. и трем месяцам тюремного заключения.

Важность решения проблемы информационной безопасности в настоящее время общепризнана, подтверждением чему служат громкие процессы о нарушении целостности систем. Убытки ведущих компаний в связи с нарушениями безопасности информации составляют триллионы долларов, причем только треть опрошенных компаний смогли определить количественно размер потерь. Проблема обеспечения безопасности носит комплексный характер, для ее решения необходимо сочетание законодательных, организационных и программно-технических мер.

Таким образом, обеспечение информационной безопасности требует системного подхода и нужно использовать разные средства и приемы — морально-этические, законодательные, административные и технические. Нас будут интересовать последние. Технические средства реализуются программным и аппаратным обеспечением и решают разные задачи по защите, они могут быть встроены в операционные системы либо могут быть реализованы в виде отдельных продуктов. Во многих случаях центр тяжести смещается в сторону защищенности операционных систем.

Есть несколько причин для реализации дополнительных средств защиты. Наиболее очевидная — помешать внешним попыткам нарушить доступ к конфиденциальной информации. Не менее важно, однако, гарантировать, что каждый программный компонент в системе использует системные ресурсы только способом, совместимым с установленной политикой применения этих ресурсов. Такие требования абсолютно необходимы для надежной системы. Кроме того, наличие защитных механизмов может увеличить надежность системы в целом за счет обнаружения скрытых ошибок интерфейса между компонентами системы. Раннее обнаружение ошибок может предотвратить «заражение» неисправной подсистемой остальных.

Политика в отношении ресурсов может меняться в зависимости от приложения и с течением времени.

Операционная система должна обеспечивать прикладные программы инструментами для создания и поддержки защищенных ресурсов. Здесь реализуется важный для гибкости системы принцип — отделение политики от механизмов. Механизмы определяют, как может быть сделано что-либо, тогда как политика решает, что должно быть сделано. Политика может меняться в зависимости от места и времени. Желательно, чтобы были реализованы по возможности общие механизмы, тогда как изменение политики требует лишь модификации системных параметров или таблиц.

К сожалению, построение защищенной системы предполагает необходимость склонить пользователя к отказу от некоторых интересных возможностей. Например, письмо, содержащее в качестве приложения документ в формате Word, может включать макросы. Открытие такого письма влечет за собой запуск чужой программы, что потенциально опасно. То же самое можно сказать про Web-страницы, содержащие апплеты. Вместо критического отношения к использованию такой функциональности пользователи современных компьютеров предпочитают периодически запускать антивирусные программы и читать успокаивающие статьи о безопасности Java.

Угрозы безопасности

Знание возможных угроз, а также уязвимых мест защиты, которые эти угрозы обычно эксплуатируют, необходимо для того, чтобы выбирать наиболее экономичные средства обеспечения безопасности.

Считается, что безопасная система должна обладать свойствами конфиденциальности, доступности и целостности. Любое потенциальное действие, которое направлено на нарушение конфиденци-

альности, целостности и доступности информации, называется угрозой. Реализованная угроза называется атакой.

Конфиденциальная (*confidentiality*) система обеспечивает уверенность в том, что секретные данные будут доступны только тем пользователям, которым этот доступ разрешен (такие пользователи называются авторизованными). Под доступностью (*availability*) понимают гарантию того, что авторизованным пользователям всегда будет доступна информация, которая им необходима. И наконец, целостность (*integrity*) системы подразумевает, что неавторизованные пользователи не могут каким-либо образом модифицировать данные.

Защита информации ориентирована на борьбу с так называемыми умышленными угрозами, т.е. с теми, которые, в отличие от случайных угроз (ошибок пользователя, сбоев оборудования и др.), преследуют цель нанести ущерб пользователям операционной системы.

Умышленные угрозы подразделяются на активные и пассивные. *Пассивная* угроза — несанкционированный доступ к информации без изменения состояния системы, *активная* — несанкционированное изменение системы. Пассивные атаки труднее выявить, так как они не влекут за собой никаких изменений данных. Защита против пассивных атак базируется на средствах их предотвращения.

Можно выделить несколько типов угроз. Наиболее распространенная угроза — *попытка проникновения в систему под видом легального пользователя*, например попытки угадывания и подбора паролей. Более сложный вариант — внедрение в систему программы, которая выводит на экран слово login. Многие легальные пользователи при этом начинают пытаться входить в систему, и их попытки могут протоколироваться. Такие безобидные с виду программы, выполняющие нежелательные функции, называются «троянскими конями». Иногда удается торпедировать работу программы проверки пароля путем многократного нажатия клавиш del, break, cancel и т.д. Для защиты от подобных атак ОС запускает процесс, называемый *аутентификацией* пользователя.

Угрозы другого рода связаны с *нежелательными действиями легальных пользователей*, которые могут, например, предпринимать попытки чтения страниц памяти, дисков и лент, которые сохранили информацию, связанную с предыдущим использованием. Защита в таких случаях базируется на надежной системе *авторизации*. В эту категорию также попадают атаки типа отказ в обслуживании, когда сервер затоплен мощным потоком запросов и становится фактически недоступным для отдельных авторизованных пользователей.

Наконец, функционирование системы может быть нарушено с помощью *программ-вирусов* или *программ-«червей»*, которые специально предназначены для того, чтобы причинить вред или недолжным образом использовать ресурсы компьютера. Общее название угроз такого рода — вредоносные программы (malicious software). Обычно они распространяются сами по себе, переходя на другие компьютеры через зараженные файлы, дискеты или по электронной почте. Наиболее эффективный способ борьбы с подобными программами — соблюдение правил «компьютерной гигиены». Многопользовательские компьютеры меньше страдают от вирусов по сравнению с персональными, поскольку там имеются системные средства защиты.

Таковы основные угрозы, на долю которых приходится львиная доля ущерба, наносимого информационным системам.

В ряде монографий [Столлингс, 2001; Таненбаум, 2002] также рассматривается модель злоумышленника, поскольку очевидно, что необходимые для организации защиты усилия зависят от предполагаемого противника.

Формализация подхода к обеспечению информационной безопасности

Проблема информационной безопасности оказалась настолько важной, что в ряде стран были выпущены основополагающие документы, в которых регламентированы основные подходы к проблеме информационной безопасности. В результате оказалось возможным ранжировать информационные системы по степени надежности.

Наиболее известна оранжевая (по цвету обложки) книга Министерства обороны США [DoD, 1993]. В этом документе определяется четыре уровня безопасности — D, C, B и A. По мере перехода от уровня D до A к надежности систем предъявляются все более жесткие требования. Уровни C и B подразделяются на классы (C1, C2, B1, B2, B3). Чтобы система в результате процедуры сертификации могла быть отнесена к некоторому классу, ее защита должна удовлетворять оговоренным требованиям.

В качестве примера рассмотрим требования класса C2, которому удовлетворяют ОС Windows NT, отдельные реализации Unix и ряд других.

- Каждый пользователь должен быть идентифицирован уникальным входным именем и паролем для входа в систему. Доступ к компьютеру предоставляется лишь после аутентификации.

- Система должна быть в состоянии использовать эти уникальные идентификаторы, чтобы следить за действиями пользователя (управление избирательным доступом). Владелец ресурса (например, файла) должен иметь возможность контролировать доступ к этому ресурсу.
- Операционная система должна защищать объекты от повторного использования. Перед выделением новому пользователю все объекты, включая память и файлы, должны инициализироваться.
- Системный администратор должен иметь возможность вести учет всех событий, относящихся к безопасности.
- Система должна защищать себя от внешнего влияния или навязывания, такого как модификация загруженной системы или системных файлов, хранящихся на диске.

Сегодня на смену оранжевой книге пришел стандарт Common Criteria, а набор критериев Controlled Access Protection Profile сменил критерии класса C2.

Основополагающие документы содержат определения многих ключевых понятий, связанных с информационной безопасностью. Некоторые из них (аутентификация, авторизация, домен безопасности и др.) будут рассмотрены в главе 15. В дальнейшем мы также будем оперировать понятиями «субъект» и «объект» безопасности. Субъект безопасности — активная системная составляющая, к которой применяется политика безопасности, а объект — пассивная. Примерами субъектов могут служить пользователи и группы пользователей, а объектов — файлы, системные таблицы, принтер и т.п.

По существу, проектирование системы безопасности подразумевает ответы на следующие вопросы: какую информацию защищать, какого рода атаки на безопасность системы могут быть предприняты, какие средства использовать для защиты каждого вида информации? Поиск ответов на данные вопросы называется формированием политики безопасности, которая помимо чисто технических аспектов включает также и решение организационных проблем. На практике реализация политики безопасности состоит в присвоении субъектам и объектам идентификаторов и фиксации набора правил, позволяющих определить, имеет ли данный субъект авторизацию, достаточную для предоставления к данному объекту указанного типа доступа.

Формируя политику безопасности, необходимо учитывать несколько базовых принципов. Так, Зальтцер (Saltzer) и Шредер (Schroeder) (1975) на основе своего опыта работы с MULTICS сформулировали следующие рекомендации для проектирования системы безопасности операционной системы.

- Проектирование системы должно быть открытым. Нарушитель и так все знает (криптографические алгоритмы открыты).
- Не должно быть доступа по умолчанию. Ошибки с отклонением легитимного доступа будут обнаружены скорее, чем ошибки там, где разрешен неавторизованный доступ.
- Нужно тщательно проверять текущее авторство. Так, многие системы проверяют привилегии доступа при открытии файла и не делают этого после. В результате пользователь может открыть файл и держать его открытым в течение недели и иметь к нему доступ, хотя владелец уже сменил защиту.
- Давать каждому процессу минимум возможных привилегий.
- Защитные механизмы должны быть просты, постоянны и встроены в нижний слой системы, это не аддитивные добавки (известно много неудачных попыток «улучшения» защиты слабо приспособленной для этого операционной системы MS-DOS).
- Важна физиологическая приемлемость. Если пользователь видит, что защита требует слишком больших усилий, он от нее откажется. Ущерб от атаки и затраты на ее предотвращение должны быть сбалансированы.

Приведенные соображения показывают необходимость продумывания и встраивания защитных механизмов на самых ранних стадиях проектирования системы.

Глава 14
ИСПОЛЬЗОВАНИЕ КРИПТОАЛГОРИТМОВ В ОПЕРАЦИОННЫХ СИСТЕМАХ

Многие службы информационной безопасности, такие как контроль входа в систему, разграничение доступа к ресурсам, обеспечение безопасного хранения данных и ряд других, опираются на использование криптографических алгоритмов. Имеется обширная литература по этому актуальному для безопасности информационных систем вопросу.

Шифрование — процесс преобразования сообщения из открытого текста (plaintext) в шифротекст (ciphertext) таким образом, чтобы:

- его могли прочитать только те стороны, для которых оно предназначено;
- проверить подлинность отправителя (аутентификация);
- гарантировать, что отправитель действительно послал данное сообщение.

В алгоритмах шифрования предусматривается наличие ключа. Ключ — это некий параметр, не зависящий от открытого текста. Результат применения алгоритма шифрования зависит от используемого ключа. В криптографии принято правило Кирхгофа: «Стойкость шифра должна определяться только секретностью ключа». Правило Кирхгофа подразумевает, что алгоритмы шифрования должны быть открыты.

В методе шифрования с *секретным* или симметричным ключом имеется один ключ, который используется как для шифрования, так и для расшифровки сообщения. Такой ключ нужно хранить в секрете. Это затрудняет использование системы шифрования, поскольку ключи должны регулярно меняться, для чего требуется их секретное распространение. Наиболее популярные алгоритмы шифрования с секретным ключом: DES, TripleDES, ГОСТ и ряд других.

Часто используется шифрование с помощью односторонней функции, называемой также хеш- или дайджест-функцией. Применение этой функции к шифруемым данным позволяет сформировать небольшой дайджест из нескольких байтов, по которому невозможно восстановить исходный текст. Получатель сообщения может проверить целостность данных, сравнивая полученный вместе с сообщением дайджест с вычисленным вновь при помощи той же односторонней функции. Эта техника активно используется для контроля входа в систему. Например, пароли пользователей хранятся на диске в зашифрованном односторонней функцией виде. Наиболее популярные хеш-функции: MD4, MD5 и др.

В системах шифрования с *открытым* или *асимметричным ключом* (public/ assymmetric key) используется два ключа (рис. 14.1). Один из ключей, называемый открытым, несекретным, используется для шифрования сообщений, которые могут быть расшифрованы только с помощью секретного ключа, имеющегося у получателя, для которого предназначено сообщение. Иногда поступают по-другому. Для шифрования сообщения используется секретный ключ, и, если сообщение можно расшифровать с помощью открытого ключа, подлинность отправителя будет гарантирована (система электронной подписи).

Этот принцип изобретен Уитфилдом Диффи (Whitfield Diffie) и Мартином Хеллманом (Martin Hellman) в 1976 г.

Рис. 14.1. Шифрование открытым ключом

Использование открытых ключей снимает проблему обмена и хранения ключей, свойственную системам с симметричными ключами. Открытые ключи могут храниться публично, и каждый может послать зашифрованное открытым ключом сообщение владельцу ключа. Однако расшифровать это сообщение может только владелец открытого ключа при помощи своего секретного ключа и никто другой. Несмотря на очевидные удобства, связанные с хранением и распространением ключей, асимметричные алгоритмы гораздо менее эффективны, чем симметричные, поэтому во многих криптографических системах используются оба метода.

Среди несимметричных алгоритмов наиболее известен RSA, предложенный Роном Ривестом (Ron Rivest), Ади Шамиром (Adi Shamir) и Леонардом Эдлманом (Leonard Adleman). Рассмотрим его более подробно.

Шифрование с использованием алгоритма RSA

Идея, положенная в основу метода, состоит в том, чтобы найти такую функцию $y = \Phi(x)$, для которой получение обратной функции $x = f^{-1}(y)$ было бы в общем случае очень сложной задачей (NP-полной задачей).

Например, получить произведение двух чисел $n = p \cdot q$ просто, а разложить n на множители, если p и q достаточно большие простые числа, — NP-полная задача с вычислительной сложностью $\sim n^{10}$. Однако если знать некую секретную информацию, то найти обратную функцию $x = f^{-1}(y)$ существенно проще. Такие функции также называют односторонними функциями с лазейкой или потайным ходом.

Применяемые в RSA прямая и обратная функции просты. Они базируются на применении теоремы Эйлера из теории чисел.

Прежде чем сформулировать теорему Эйлера, необходимо определить важную функцию $\Phi(n)$ из теории чисел, называемую функцией Эйлера. Это число взаимно простых (взаимно простыми называются целые числа, не имеющие общих делителей) с n целых чисел, меньших n. Например, $\Phi(7) = 6$. Очевидно, что, если p и q — простые числа и $p \neq q$, то $\Phi(p) = p - 1$ и $\Phi(p \cdot q) = (p - 1) \cdot (q - 1)$.

Теорема Эйлера

Теорема Эйлера утверждает, что для любых взаимно простых чисел x и n ($x < n$)

$$x^{\Phi(n)} mod\ n = 1$$

или в более общем виде

$$x^{k\Phi(n)+1} mod\ n = 1.$$

Сформулируем еще один важный результат. Для любого $m > 0$ и $0 < e < m$, где e и m взаимно просты, найдется единственное $0 < d < m$, такое, что

$$de\ mod\ m = 1.$$

Здесь d легко можно найти по обобщенному алгоритму Евклида (см., например: *Кнут Д.* Искусство программирования на ЭВМ. Т. 2, раздел 4.5.2). Известно, что вычислительная сложность алгоритма Евклида ~ $ln\ n$. Подставляя $\Phi(n)$ вместо m, получим $de\ mod \Phi(n) = 1$ или

$$de = k\Phi(n) + 1.$$

Тогда прямой функцией будет

$$\Phi(x) = x^e\ mod\ n,$$

где x — положительное целое, $x < n = pq$, p и q — целые простые числа и, следовательно,

$$\Phi(n) = (p - 1)(q - 1),$$

где e — положительное целое и $e < \Phi(n)$. Здесь e и n открыты. Однако p и q неизвестны (чтобы их найти, нужно выполнить разбиение n на множители), следовательно, неизвестна и $\Phi(n)$, а именно они и составляют потайной ход.

Вычислим обратную функцию

$$\Phi^{-1}(y) = y^d\ mod\ n = x^{ed}\ mod\ n = x^{k\Phi(n)+1}\ mod\ n = x.$$

Последнее преобразование справедливо, поскольку $x < n$, а x и n взаимно просты.

При практическом использовании алгоритма RSA вначале необходимо выполнить генерацию ключей.

Для этого нужно:

1. Выбрать два очень больших простых числа p и q;
2. Вычислить произведение $n = p \cdot q$;
3. Выбрать большое случайное число d, не имеющее общих сомножителей с числом $(p - 1) \cdot (q - 1)$;
4. Определить число e, чтобы выполнялось

$$(e \cdot d) mod((p - 1) \cdot (q - 1)) = 1.$$

Тогда открытым ключом будут числа e и n, а секретным ключом — числа d и n.

Теперь, чтобы зашифровать данные по известному ключу $\{e,n\}$, необходимо сделать следующее:

- Разбить шифруемый текст на блоки, где i-й блок представить в виде числа M, величина которого меньше, чем n. Это можно сделать различными способами, например используя вместо букв их номера в алфавите.
- Зашифровать текст, рассматриваемый как последовательность чисел $m(i)$, по формуле

$$c(i) = (m(i)^e) mod\ n.$$

Чтобы расшифровать эти данные, используя секретный ключ $\{d,n\}$, необходимо вычислить: $m(i) = (c(i)^d)\ mod\ n$. В результате будет получено множество чисел $m(i)$, которые представляют собой часть исходного текста.

Например, зашифруем и расшифруем сообщение «АБВ», которое представим как число 123.

Выбираем $p = 5$ и $q = 11$ (числа на самом деле должны быть большими).

Находим $n = 5 \cdot 11 = 55$.

Определяем $(p - 1) \cdot (q - 1) = 40$. Тогда d будет равно, например, 7.

Выберем e, исходя из $(e \cdot 7)\ mod\ 40 = 1$. Например, $e = 3$.

Теперь зашифруем сообщение, используя открытый ключ {3,55}

$C1 = (1^3)\ mod\ 55 = 1$;

$C2 = (2^3)\ mod\ 55 = 8$;

$C3 = (3^3)\ mod\ 55 = 27$.

Теперь расшифруем эти данные, используя закрытый ключ {7,55}.

$M1 = (1^7)\ mod\ 55 = 1$;

$M2 = (8^7)\ mod\ 55 = 2097152\ mod\ 55 = 2$;

$M3 = (27^7)\ mod\ 55 = 10460353203\ mod\ 55 = 3$.

Таким образом, все данные расшифрованы.

Глава 15
СРЕДСТВА АУТЕНТИФИКАЦИИ

Решение вопросов безопасности операционных систем обусловлено их архитектурными особенностями и связано с правильной организацией идентификации и аутентификации, авторизации и аудита.

Перейдем к описанию системы защиты операционных систем. Ее основными задачами являются идентификация, аутентификация, разграничение доступа пользователей к ресурсам, протоколирование и аудит самой системы.

15.1. Идентификация и аутентификация

Для начала рассмотрим проблему контроля доступа в систему. Наиболее распространенным способом контроля доступа является процедура регистрации. Обычно каждый пользователь в системе имеет уникальный идентификатор. Идентификаторы пользователей применяются с той же целью, что и идентификаторы любых других объектов, файлов, процессов. Идентификация заключается в сообщении пользователем своего идентификатора. Для того чтобы установить, что пользователь именно тот, за кого себя выдает, т.е. что именно ему принадлежит введенный идентификатор, в информационных системах предусмотрена процедура аутентификации (authentication, опознавание, в переводе с латинского означает «установление подлинности»), задача которой — предотвращение доступа к системе нежелательных лиц.

Обычно аутентификация базируется на одном или более из трех пунктов:

- то, чем пользователь владеет (ключ или магнитная карта);
- то, что пользователь знает (пароль);
- атрибуты пользователя (отпечатки пальцев, подпись, голос).

Пароли, уязвимость паролей

Наиболее простой подход к аутентификации — применение пользовательского пароля.

Когда пользователь идентифицирует себя при помощи уникального идентификатора или имени, у него запрашивается пароль. Если пароль, сообщенный пользователем, совпадает с паролем, храня-

щимся в системе, система предполагает, что пользователь легитимен. Пароли часто используются для защиты объектов в компьютерной системе в отсутствие более сложных схем защиты.

Недостатки паролей связаны с тем, что трудно сохранить баланс между удобством пароля для пользователя и его надежностью. Пароли могут быть угаданы, случайно показаны или нелегально переданы авторизованным пользователем неавторизованному.

Есть два общих способа угадать пароль. Один связан со сбором информации о пользователе. Люди обычно используют в качестве паролей очевидную информацию (скажем, имена животных или номерные знаки автомобилей). Для иллюстрации важности разумной политики назначения идентификаторов и паролей можно привести данные исследований, проведенных в AT&T, показывающие, что из 500 попыток несанкционированного доступа около 300 составляют попытки угадывания паролей или беспарольного входа по пользовательским именам guest, demo и т.д.

Другой способ — попытаться перебрать все наиболее вероятные комбинации букв, чисел и знаков пунктуации (атака по словарю). Например, четыре десятичные цифры дают только 10 000 вариантов, более длинные пароли, введенные с учетом регистра символов и пунктуации, не столь уязвимы, но, тем не менее, таким способом удается разгадать до 25% паролей. Чтобы заставить пользователя выбрать трудноугадываемый пароль, во многих системах внедрена реактивная проверка паролей, которая при помощи собственной программы-взломщика паролей может оценить качество пароля, введенного пользователем.

Несмотря на все это, пароли распространены, поскольку они удобны и легко реализуемы.

Шифрование пароля

Для хранения секретного списка паролей на диске во многих операционных системах используется криптография. Система задействует одностороннюю функцию, которую просто вычислить, но для которой чрезвычайно трудно (разработчики надеются, что невозможно) подобрать обратную функцию.

Например, в ряде версий Unix в качестве односторонней функции используется модифицированный вариант алгоритма DES. Введенный пароль длиной до 8 знаков преобразуется в 56-битовое значение, которое служит входным параметром для процедуры *crypt*(), основанной на этом алгоритме. Результат шифрования зависит не только от введенного пароля, но и от случайной последовательности битов,

называемой привязкой (переменная *salt*). Это сделано для того, чтобы решить проблему совпадающих паролей. Очевидно, что саму привязку после шифрования необходимо сохранять, иначе процесс не удастся повторить. Модифицированный алгоритм DES выполняется, имея входное значение в виде 64-битового блока нулей, с использованием пароля в качестве ключа, а на каждой следующей итерации входным параметром служит результат предыдущей итерации. Всего процедура повторяется 25 раз. Полученное 64-битовое значение преобразуется в 11 символов и хранится рядом с открытой переменной *salt*.

В операционной системе Windows NT преобразование исходного пароля также осуществляется многократным применением алгоритма DES и алгоритма MD4.

Хранятся только кодированные пароли. В процессе аутентификации представленный пользователем пароль кодируется и сравнивается с хранящимися на диске. Таким образом, файл паролей нет необходимости держать в секрете.

При удаленном доступе к операционной системе нежелательна передача пароля по сети в открытом виде. Одним из типовых решений является использование криптографических протоколов. В качестве примера можно рассмотреть протокол опознавания с подтверждением установления связи путем вызова — CHAP (Challenge Handshake Authentication Protocol).

Опознавание достигается за счет проверки того, что у пользователя, осуществляющего доступ к серверу, имеется секретный пароль, который уже известен серверу.

Пользователь инициирует диалог, передавая серверу свой идентификатор. В ответ сервер посылает пользователю запрос (вызов), состоящий из идентифицирующего кода, случайного числа и имени узла сервера или имени пользователя. При этом пользовательское оборудование в результате запроса пароля пользователя отвечает следующим ответом, зашифрованным с помощью алгоритма одностороннего хеширования, наиболее распространенным видом которого является MD5. После получения ответа сервер при помощи той же функции с теми же аргументами шифрует собственную версию пароля пользователя. В случае совпадения результатов вход в систему разрешается. Существенно, что незашифрованный пароль при этом по каналу связи не посылается.

В микротелефонных трубках используется аналогичный метод.

В системах, работающих с большим количеством пользователей, когда хранение всех паролей затруднительно, применяются для опознавания сертификаты, выданные доверенной стороной.

15.2. Авторизация. Разграничение доступа к объектам операционной системы

После успешной регистрации система должна осуществлять авторизацию (authorization) — предоставление субъекту прав на доступ к объекту. Средства авторизации контролируют доступ легальных пользователей к ресурсам системы, предоставляя каждому из них именно те права, которые были определены администратором, а также осуществляют контроль возможности выполнения пользователем различных системных функций. Система контроля базируется на общей модели, называемой матрицей доступа. Рассмотрим ее более подробно.

Как уже говорилось в предыдущей лекции, компьютерная система может быть смоделирована как набор субъектов (процессы, пользователи) и объектов. Под объектами мы понимаем как ресурсы оборудования (процессор, сегменты памяти, принтер, диски и ленты), так и программные ресурсы (файлы, программы, семафоры), т.е. все то, доступ к чему контролируется. Каждый объект имеет уникальное имя, отличающее его от других объектов в системе, и каждый из них может быть доступен через хорошо определенные и значимые операции.

Операции зависят от объектов. Например, процессор может только выполнять команды, сегменты памяти могут быть записаны и прочитаны, считыватель магнитных карт может только читать, а файлы данных могут быть записаны, прочитаны, переименованы и т.д.

Желательно добиться того, чтобы процесс осуществлял авторизованный доступ только к тем ресурсам, которые ему нужны для выполнения его задачи. Это требование минимума привилегий полезно с точки зрения ограничения количества повреждений, которые процесс может нанести системе. Например, когда процесс P вызывает процедуру A, ей должен быть разрешен доступ только к переменным и формальным параметрам, переданным ей, она не должна иметь возможность влиять на другие переменные процесса. Аналогично компилятор не должен оказывать влияния на произвольные файлы, а только на их хорошо определенное подмножество (исходные файлы, листинги и др.), имеющее отношение к компиляции. С другой стороны, компилятор может иметь личные файлы, используемые для оптимизационных целей, к которым процесс P не имеет доступа.

Различают дискреционный (избирательный) способ управления доступом и полномочный (мандатный).

При дискреционном доступе, подробно рассмотренном ниже, определенные операции над конкретным ресурсом запрещаются или разрешаются субъектам или группам субъектов. С концептуальной точки зрения текущее состояние прав доступа при дискреционном управлении описывается матрицей, в строках которой перечислены субъекты, в столбцах — объекты, а в ячейках — операции, которые субъект может выполнить над объектом.

Полномочный подход заключается в том, что все объекты могут иметь уровни секретности, а все субъекты делятся на группы, образующие иерархию в соответствии с уровнем допуска к информации. Иногда это называют моделью многоуровневой безопасности, которая должна обеспечивать выполнение следующих правил.

- Простое свойство секретности. Субъект может читать информацию только из объекта, уровень секретности которого не выше уровня секретности субъекта. Генерал читает документы лейтенанта, но не наоборот.
- *-свойство. Субъект может записывать информацию в объекты только своего уровня или более высоких уровней секретности. Генерал не может случайно разгласить нижним чинам секретную информацию.

Некоторые авторы утверждают [Таненбаум, 2002], что последнее требование называют *-свойством, потому что в оригинальном докладе не смогли придумать для него подходящего названия. В итоге во все последующие документы и монографии оно вошло как *-свойство.

Отметим, что данная модель разработана для хранения секретов, но не гарантирует целостности данных.

Например, здесь лейтенант имеет право писать в файлы генерала. Более подробно о реализации подобных формальных моделей рассказано в [Столлингс, 2002], [Таненбаум, 2002].

Большинство операционных систем реализуют именно дискреционное управление доступом. Главное его достоинство — гибкость, основные недостатки — рассредоточенность управления и сложность централизованного контроля.

Домены безопасности

Чтобы рассмотреть схему дискреционного доступа более детально, введем концепцию домена безопасности (protection domain). Каждый домен определяет набор объектов и типов операций, кото-

рые могут производиться над каждым объектом. Возможность выполнять операции над объектом есть права доступа, каждое из которых есть упорядоченная пара <object-name, rights-set>. Домен, таким образом, есть набор прав доступа. Например, если домен D имеет права доступа <file F, {read, write}>, это означает, что процесс, выполняемый в домене D, может читать или писать в файл F, но не может выполнять других операций над этим объектом. Пример доменов можно увидеть на рис. 15.1.

Домен \ Объект	F1	F2	F3	Printer
D1	read			
D2				print
D3		read	execute	
D4	read write		read write	

Рис. 15.1. Специфицирование прав доступа к ресурсам

Связь конкретных субъектов, функционирующих в операционных системах, может быть организована следующим образом.

- Каждый пользователь может быть доменом. В этом случае набор объектов, к которым может быть организован доступ, зависит от идентификации пользователя.
- Каждый процесс может быть доменом. В этом случае набор доступных объектов определяется идентификацией процесса.
- Каждая процедура может быть доменом. В этом случае набор доступных объектов соответствует локальным переменным, определенным внутри процедуры. Заметим, что когда процедура выполнена, происходит смена домена.

Рассмотрим стандартную двухрежимную модель выполнения операционной системы. Когда процесс выполняется в режиме системы (kernel mode), он может выполнять привилегированные инструкции и иметь полный контроль над компьютерной системой. С другой стороны, если процесс выполняется в пользовательском режиме, он может вызывать только непривилегированные инструкции. Следовательно, он может выполняться только внутри предопределенного пространства памяти. Наличие этих двух режимов позволяет защитить операционную систему (kernel domain) от пользовательских

процессов (выполняющихся в user domain). В мультипрограммных системах двух доменов недостаточно, так как появляется необходимость защиты пользователей друг от друга. Поэтому требуется более тщательно разработанная схема.

В операционной системе Unix домен связан с пользователем. Каждый пользователь обычно работает со своим набором объектов.

Матрица доступа

Модель безопасности, специфицированная в предыдущем разделе (рис. 15.1), имеет вид матрицы, которая называется матрицей доступа. Какова может быть эффективная реализация матрицы доступа? В общем случае она будет разреженной, т.е. большинство ее клеток будут пустыми. Хотя существуют структуры данных для представления разреженной матрицы, они не слишком полезны для приложений, использующих возможности защиты. Поэтому на практике матрица доступа применяется редко. Эту матрицу можно разложить по столбцам, в результате чего получаются *списки прав доступа* (access control list — ACL). В результате разложения по строкам получаются мандаты возможностей (capability list или capability tickets).

Список прав доступа. Access control list

Каждая колонка в матрице может быть реализована как список доступа для одного объекта. Очевидно, что пустые клетки могут не учитываться. В результате для каждого объекта имеем список упорядоченных пар <domain, rights-set>, который определяет все домены с непустыми наборами прав для данного объекта.

Элементами списка могут быть процессы, пользователи или группы пользователей. При реализации широко применяется предоставление доступа по умолчанию для пользователей, права которых не указаны.

Например, в Unix все субъекты-пользователи разделены на три группы (владелец, группа и остальные), и для членов каждой группы контролируются операции чтения, записи и исполнения (rwx). В итоге имеем ACL — 9-битный код, который является атрибутом разнообразных объектов Unix.

Мандаты возможностей. Capability list

Как отмечалось выше, если матрицу доступа хранить по строкам, т.е. если каждый субъект хранит список объектов и для каждого объекта — список допустимых операций, то такой способ хранения называется «мандаты» или «перечни возможностей» (capability list).

Каждый пользователь обладает несколькими мандатами и может иметь право передавать их другим. Мандаты могут быть рассеяны по системе и вследствие этого представлять большую угрозу для безопасности, чем списки контроля доступа. Их хранение должно быть тщательно продумано.

Примерами систем, использующих перечни возможностей, являются Hydra, Cambridge CAP System [Denning, 1996].

Другие способы контроля доступа

Иногда применяется *комбинированный способ*. Например, в том же Unix на этапе открытия файла происходит анализ ACL (операция open). В случае благоприятного исхода файл заносится в список открытых процессом файлов, и при последующих операциях чтения и записи проверки прав доступа не происходит. Список открытых файлов можно рассматривать как перечень возможностей.

Существует также схема *lock-key*, которая является компромиссом между списками прав доступа и перечнями возможностей. В этой схеме каждый объект имеет список уникальных битовых шаблонов (patterns), называемых locks. Аналогично каждый домен имеет список уникальных битовых шаблонов, называемых ключами (keys). Процесс, выполняющийся в домене, может получить доступ к объекту, только если домен имеет ключ, который соответствует одному из шаблонов объекта.

Как и в случае мандатов, список ключей для домена должен управляться операционной системой. Пользователям не разрешается проверять или модифицировать списки ключей (или шаблонов) непосредственно. Более подробно данная схема изложена в [Silberschatz, 2002].

Смена домена

В большинстве операционных систем для определения домена применяются идентификаторы пользователей. Обычно переключение между доменами происходит, когда меняется пользователь. Но почти все системы нуждаются в дополнительных механизмах смены домена, которые используются, когда некая привилегированная возможность необходима большому количеству пользователей. Например, может понадобиться разрешить пользователям иметь доступ к сети, не заставляя их писать собственные сетевые программы. В таких случаях для процессов операционной системы Unix предусмотрена установка бита *set-uid*. В результате установки этого бита в сетевой программе она получает привилегии ее создателя (а не пользователя), заставляя домен меняться на время ее выполнения. Таким образом, рядовой пользователь может получить нужные привилегии для доступа к сети.

Недопустимость повторного использования объектов

Контроль повторного использования объекта предназначен для предотвращения попыток незаконного получения конфиденциальной информации, остатки которой могли сохраниться в некоторых объектах, ранее использовавшихся и освобожденных другим пользователем. Безопасность повторного применения должна гарантироваться для областей оперативной памяти (в частности, для буферов с образами экрана, расшифрованными паролями и т.п.), для дисковых блоков и магнитных носителей в целом. Очистка должна производиться путем записи маскирующей информации в объект при его освобождении (перераспределении). Например, для дисков на практике применяется способ двойной перезаписи освободившихся после удаления файлов блоков случайной битовой последовательностью.

15.3. Выявление вторжений. Аудит системы защиты

Даже самая лучшая система защиты рано или поздно будет взломана. Обнаружение попыток вторжения является важнейшей задачей системы защиты, поскольку ее решение позволяет минимизировать ущерб от взлома и собирать информацию о методах вторжения. Как правило, поведение взломщика отличается от поведения легального пользователя. Иногда эти различия можно выразить количественно, например подсчитывая число некорректных вводов пароля во время регистрации.

Основным инструментом выявления вторжений является запись данных аудита. Отдельные действия пользователей протоколируются, а полученный протокол используется для выявления вторжений.

Аудит, таким образом, заключается в регистрации специальных данных о различных типах событий, происходящих в системе и так или иначе влияющих на состояние безопасности компьютерной системы.

К числу таких событий обычно причисляют следующие:

- вход или выход из системы;
- операции с файлами (открыть, закрыть, переименовать, удалить);
- обращение к удаленной системе;

- смена привилегий или иных атрибутов безопасности (режима доступа, уровня благонадежности пользователя и т.п.).

Если фиксировать все события, объем регистрационной информации, скорее всего, будет расти слишком быстро, а ее эффективный анализ станет невозможным. Следует предусматривать наличие средств выборочного протоколирования как в отношении пользователей, когда слежение осуществляется только за подозрительными личностями, так и в отношении событий. Слежка важна в первую очередь как профилактическое средство. Можно надеяться, что многие воздержатся от нарушений безопасности, зная, что их действия фиксируются.

Помимо протоколирования, можно периодически **сканировать** систему на наличие слабых мест в системе безопасности. Такое сканирование может проверить разнообразные аспекты системы:

- короткие или легкие пароли;
- неавторизованные set-uid программы, если система поддерживает этот механизм;
- неавторизованные программы в системных директориях;
- долго выполняющиеся программы;
- нелогичная защита как пользовательских, так и системных директорий и файлов. Примером нелогичной защиты может быть файл, который запрещено читать его автору, но в который разрешено записывать информацию постороннему пользователю;
- потенциально опасные списки поиска файлов, которые могут привести к запуску «троянского коня»;
- изменения в системных программах, обнаруженные при помощи контрольных сумм.

Любая проблема, обнаруженная сканером безопасности, может быть как ликвидирована автоматически, так и передана для решения менеджеру системы.

Глава 16
МЕЖПРОЦЕССНОЕ ВЗАИМОДЕЙСТВИЕ: СИГНАЛЫ, TCP/IP, D-BUS, СОКЕТЫ BERKLEY

Межпроцессное взаимодействие (*inter-process communication*, IPC) — обмен данными между потоками одного или разных процессов. Реализуется посредством механизмов, предоставляемых ядром операционной системы или процессом, использующим механизмы операционной системы и реализующим новые возможности IPC. Может осуществляться как на одном компьютере, так и между несколькими компьютерами сети.

Из механизмов, предоставляемых операционной системой и используемых для IPC, можно выделить:

- механизмы обмена сообщениями;
- механизмы синхронизации;
- механизмы разделения памяти;
- механизмы удаленных вызовов (RPC).

Для оценки производительности различных механизмов IPC используют следующие параметры:

- пропускная способность (количество сообщений в единицу времени, которое ядро операционной системы или процесс способно обработать);
- задержки (время между отправкой сообщения одним потоком и его получением другим потоком).

IPC может называться терминами *межпотоковое взаимодействие* (*inter-thread communication*) и *межпрограммное взаимодействие* (*inter-application communication*).

Межпроцессное взаимодействие, наряду с механизмами адресации памяти, является основой для разграничения адресного пространства между процессами.

Таблица методов межпроцессного взаимодействия

Метод	Реализуется операционной системой или процессом
Файл	Все операционные системы
Сигнал	Большинство операционных систем; в некоторых операционных системах, например в Windows, сигналы доступны только в библиотеках, реализующих стандартную библиотеку языка Си, и не могут использоваться для IPC

Окончание таблицы

Метод	Реализуется операционной системой или процессом
Сокет	Большинство операционных систем
Канал	Все операционные системы, совместимые со стандартом POSIX
Именованный канал	Все операционные системы, совместимые со стандартом POSIX
Неименованный канал	Все операционные системы, совместимые со стандартом POSIX
Семафор	Все операционные системы, совместимые со стандартом POSIX
Разделяемая память	Все операционные системы, совместимые со стандартом POSIX
Обмен сообщениями (без разделения)	Используется в парадигме MPI, Java RMI, CORBA и других
Проецируемый в память файл (mmap)	Все операционные системы, совместимые со стандартом POSIX. При использовании временного файла возможно возникновение гонки. Операционная система Windows также предоставляет этот механизм, но посредством API, отличающегося от API, описанного в стандарте POSIX
Очередь сообщений (Message queue)	Большинство операционных систем
Почтовый ящик	Некоторые операционные системы

Сигналы

Сигнал — это асинхронное уведомление процесса о каком-либо событии. Когда сигнал послан процессу, операционная система прерывает выполнение процесса. Если процесс установил собственный *обработчик сигнала*, операционная система запускает этот обработчик, передав ему информацию о сигнале. Если процесс не установил обработчик, то выполняется обработчик по умолчанию.

Названия сигналов «SIG...» являются числовыми константами (макроопределениями Си) со значениями, определяемыми в заголовочном файле `signal.h`. Числовые значения сигналов могут меняться от системы к системе, хотя основная их часть имеет в разных системах одни и те же значения. Утилита `kill` позволяет задавать сигнал как числом, так и символьным обозначением.

Посылка сигналов

Сигналы посылаются:

- с терминала, нажатием специальных клавиш или комбинаций (например, нажатие Ctrl-C генерирует SIGINT, Ctrl-\ SIGQUIT, а Ctrl-Z SIGTSTP);
- ядром системы:
  - при возникновении аппаратных исключений (недопустимых инструкций, нарушениях при обращении в память, системных сбоях и т.п.);
  - ошибочных системных вызовах;
  - для информирования о событиях ввода-вывода;
- одним процессом другому (или самому себе), с помощью системного вызова `kill()`, в том числе:
  из shell, утилитой `/bin/kill`.

Сигналы не могут быть посланы завершившемуся процессу, находящемуся в состоянии «зомби».

Обработка сигналов

Обработчик по умолчанию для большинства сигналов завершает выполнение процесса. Для альтернативной обработки всех сигналов, за исключением SIGKILL и SIGSTOP, процесс может назначить свой обработчик или игнорировать их возникновение модификацией своей *сигнальной маски*. Единственное исключение — процесс с pid 1 (init), который имеет право игнорировать или обрабатывать любые сигналы, включая KILL и STOP.

Процесс (или пользователь из оболочки) с эффективным UID, не равным 0 (UID суперпользователя), может посылать сигналы только процессам с тем же UID.

Классификация сигналов

POSIX определяет 28 сигналов, которые можно классифицировать следующим образом:

Название	Код	Действие по умолчанию	Описание	Тип
SIGABRT	6	Завершение с дампом памяти	Сигнал, посылаемый функцией `abort()`	Управление
SIGALRM	14	Завершение	Сигнал истечения времени, заданного `alarm()`	Уведомление

Продолжение таблицы

Название	Код	Действие по умолчанию	Описание	Тип
SIGBUS	10	Завершение с дампом памяти	Неправильное обращение в физическую память	Исключение
SIGCHLD	18	Игнорируется	Дочерний процесс завершен или остановлен	Уведомление
SIGCONT	25	Продолжить выполнение	Продолжить выполнение ранее остановленного процесса	Управление
SIGFPE	8	Завершение с дампом памяти	Ошибочная арифметическая операция	Исключение
SIGHUP	1	Завершение	Закрытие терминала	Уведомление
SIGILL	4	Завершение с дампом памяти	Недопустимая инструкция процессора	Исключение
SIGINT	2	Завершение	Сигнал прерывания (Ctrl-C) с терминала	Управление
SIGKILL	9	Завершение	Безусловное завершение	Управление
SIGPIPE	13	Завершение	Запись в разорванное соединение (пайп, сокет)	Уведомление
SIGQUIT	3	Завершение с дампом памяти	Сигнал «Quit» с терминала (Ctrl-\)	Управление
SIGSEGV	11	Завершение с дампом памяти	Нарушение при обращении в память	Исключение
SIGSTOP	23	Остановка процесса	Остановка выполнения процесса	Управление
SIGTERM	15	Завершение	Сигнал завершения (сигнал по умолчанию для утилиты kill)	Управление

Продолжение таблицы

Название	Код	Действие по умолчанию	Описание	Тип
SIGTSTP	20	Остановка процесса	Сигнал остановки с терминала (Ctrl-Z)	Управление
SIGTTIN	26	Остановка процесса	Попытка чтения с терминала фоновым процессом	Управление
SIGTTOU	27	Остановка процесса	Попытка записи на терминал фоновым процессом	Управление
SIGUSR1	16	Завершение	Пользовательский сигнал № 1	Пользовательский
SIGUSR2	17	Завершение	Пользовательский сигнал № 2	Пользовательский
SIGPOLL	22	Завершение	Событие, отслеживаемое `poll()`	Уведомление
SIGPROF	29	Завершение	Истечение таймера профилирования	Отладка
SIGSYS	12	Завершение с дампом памяти	Неправильный системный вызов	Исключение
SIGTRAP	5	Завершение с дампом памяти	Ловушка трассировки или брейкпоинт	Отладка
SIGURG	21	Игнорируется	На сокете получены срочные данные	Уведомление
SIGVTALRM	28	Завершение	Истечение «виртуального таймера»	Уведомление
SIGXCPU	30	Завершение с дампом памяти	Процесс превысил лимит процессорного времени	Исключение

Окончание таблицы

Название	Код	Действие по умолчанию	Описание	Тип
SIGXFSZ	31	Завершение с дампом памяти	Процесс превысил допустимый размер файла	Исключение

При обработке исключений и отладочных сигналов перед завершением процесс может записать в текущий каталог файл с дампом памяти процесса (*core image*), используя который, отладчик может восстановить условия, при которых возникло данное исключение. Иногда (например, для программ, выполняемых от имени суперпользователя) дамп памяти не создается из соображений безопасности.

SA_SIGINFO

Обычно обработчик сигнала получает только один аргумент — номер сигнала (это позволяет использовать одну функцию-обработчик для нескольких сигналов). Если при задании обработчика сигнала (функцией `sigaction()`) указать опцию SA_SIGINFO, то в обработчик будут переданы еще два аргумента:

1) указатель на структуру `siginfo_t`, включающую:
- битовую маску дополнительных «кодов сигнала», определяющих причину его возникновения;
- идентификатор процесса (PID), пославшего сигнал;
- эффективный идентификатор пользователя (UID), от имени которого выполняется процесс (например, утилита `kill`), пославший сигнал;
- адрес инструкции, в которой возникло исключение;
- и т.п.;

2) указатель на «машинный контекст» на момент возникновения сигнала (со «стеком сигнала» — дополнительными данными, которые помещаются в стек при вызове некоторых сигналов-исключений).

Большинство дополнительных кодов специфичны для каждого сигнала. Коды, общие для всех сигналов:

Код	Описание
SI_USER	Сигнал послан функцией `kill()` (или утилитой `kill`)
SI_QUEUE	Сигнал послан функцией `sigqueue()`

Окончание таблицы

Код	Описание
SI_TIMER	Сигнал послан по истечении времени, установленного функцией `timer_settime()`
SI_ASYNCIO	Сигнал послан по завершении запроса на «асинхронный ввод-вывод»
SI_MESGQ	Сигнал послан по появлению сообщения в пустой «очереди сообщений Unix»

TCP/IP

TCP/IP — сетевая модель, разработанная для описания набора протоколов передачи данных, на которых в настоящее время базируется Интернет. Название TCP/IP происходит из двух важнейших протоколов семейства — Transmission Control Protocol (TCP) и Internet Protocol (IP), которые были разработаны и описаны первыми в данном стандарте. Также изредка упоминается как модель DOD в связи с историческим происхождением от сети ARPANET из 1970-х гг. (под управлением DARPA, Министерства обороны США).

Уровни протоколов TCP/IP расположены по принципу стека (*stack*, стопка) — это означает, что протокол, располагающийся на уровне выше, работает «поверх» нижнего, используя механизмы инкапсуляции. Например, протокол TCP работает поверх протокола IP.

Стек протоколов TCP/IP был создан на основе NCP (Network Control Protocol) группой разработчиков под руководством Винтона Серфа в 1972 г.. В июле 1976 г. Винт Серф и Боб Кан впервые продемонстрировали передачу данных с использованием TCP по трем различным сетям. Пакет прошел по следующему маршруту: Сан-Франциско — Лондон — Университет Южной Калифорнии. В конце своего путешествия пакет проделал 150 тыс. км, не потеряв ни одного бита. В 1978 г. Серф, Джон Постел и Дэнни Кохэн решили выделить в TCP две отдельные функции: TCP и IP. TCP был ответственен за разбивку сообщения на датаграммы (*datagram*) и соединение их в конечном пункте отправки. IP отвечал за передачу (с контролем получения) отдельных датаграмм. Вот так родился современный протокол Интернета. А 1 января 1983 г. ARPANET перешла на новый протокол. Этот день принято считать официальной датой рождения Интернета.

Стек протоколов TCP/IP включает в себя четыре уровня:
- прикладной уровень (application layer);
- транспортный уровень (transport layer);

- сетевой уровень (межсетевой) (Internet layer);
- канальный уровень (link layer).

Протоколы этих уровней полностью реализуют функциональные возможности модели OSI. На стеке протоколов TCP/IP построено все взаимодействие пользователей в IP-сетях. Стек является независимым от физической среды передачи данных, благодаря чему, в частности, обеспечивается полностью прозрачное взаимодействие между проводными и беспроводными сетями.

Распределение протоколов по уровням модели TCP/IP

1	**Прикладной** (Application layer)	напр., HTTP, RTSP, FTP, DNS
2	**Транспортный** (Transport layer)	напр., TCP, UDP, SCTP, DCCP (*RIP, протоколы маршрутизации, подобные OSPF, что работают поверх IP, являются частью сетевого уровня*)
3	**Сетевой (межсетевой)** (Internet layer)	Для TCP/IP это IP (*вспомогательные протоколы, вроде ICMP и IGMP, работают поверх IP, но тоже относятся к сетевому уровню; протокол ARP является самостоятельным вспомогательным протоколом, работающим поверх канального уровня*)
4	**Канальный** (Link layer)	Ethernet, IEEE 802.11 WLAN, SLIP, Token Ring, ATM и MPLS, физическая среда и принципы кодирования информации, T1, E1

Прикладной уровень

На прикладном уровне (Application layer) работает большинство сетевых приложений.

Эти программы имеют свои собственные протоколы обмена информацией, например, интернет-браузер для протокола HTTP, ftp-клиент для протокола FTP (передача файлов), почтовая программа для протокола SMTP (электронная почта), SSH (безопасное соединение с удаленной машиной), DNS (преобразование символьных имен в IP-адреса) и многие другие.

В массе своей эти протоколы работают поверх TCP или UDP и привязаны к определенному порту, например:

- HTTP на TCP-порт 80 или 8080;
- FTP на TCP-порт 20 (для передачи данных) и 21 (для управляющих команд);
- SSH на TCP-порт 22;
- запросы DNS на порт UDP (реже TCP) 53;
- обновление маршрутов по протоколу RIP на UDP-порт 520.

Эти порты определены Агентством по выделению имен и уникальных параметров протоколов (IANA).

К этому уровню относятся: DHCP, Echo, Finger, Gopher, HTTP, HTTPS, IMAP, IMAPS, IRC, NNTP, NTP, POP3, POPS, QOTD, RTSP, SNMP, SSH, Telnet, XDMCP.

Транспортный уровень

Протоколы транспортного уровня (Transport layer) могут решать проблему негарантированной доставки сообщений («дошло ли сообщение до адресата?»), а также гарантировать правильную последовательность прихода данных. В стеке TCP/IP транспортные протоколы определяют, для какого именно приложения предназначены эти данные.

Протоколы автоматической маршрутизации, логически представленные на этом уровне (поскольку работают поверх IP), на самом деле являются частью протоколов сетевого уровня; например OSPF (IP идентификатор 89).

TCP (IP идентификатор 6) — «гарантированный» транспортный механизм с предварительным установлением соединения, предоставляющий приложению надежный поток данных, дающий уверенность в безошибочности получаемых данных, перезапрашивающий данные в случае потери и устраняющий дублирование данных. TCP позволяет регулировать нагрузку на сеть, а также уменьшать время ожидания данных при передаче на большие расстояния. Более того, TCP гарантирует, что полученные данные были отправлены точно в такой же последовательности. В этом его главное отличие от UDP.

UDP (IP идентификатор 17) протокол передачи датаграмм без установления соединения. Также его называют протоколом «ненадежной» передачи, в смысле невозможности удостовериться в доставке сообщения адресату, а также возможного перемешивания пакетов. В приложениях, требующих гарантированной передачи данных, используется протокол TCP.

UDP обычно используется в таких приложениях, как потоковое видео и компьютерные игры, где допускается потеря пакетов, а повторный запрос затруднен или не оправдан, либо в приложениях вида запрос-ответ (например, запросы к DNS), где создание соединения занимает больше ресурсов, чем повторная отправка.

И TCP, и UDP используют для определения протокола верхнего уровня число, называемое портом.

Сетевой (межсетевой) уровень

Межсетевой уровень (Internet layer) изначально разработан для передачи данных из одной сети в другую. На этом уровне работают маршрутизаторы, которые перенаправляют пакеты в нужную сеть путем расчета адреса сети по маске сети. Примерами такого протокола является X.25 и IPC в сети ARPANET.

С развитием концепции глобальной сети в уровень были внесены дополнительные возможности по передаче из любой сети в любую сеть, независимо от протоколов нижнего уровня, а также возможность запрашивать данные от удаленной стороны, например в протоколе ICMP (используется для передачи диагностической информации IP-соединения) и IGMP (используется для управления multicast-потоками).

ICMP и IGMP расположены над IP и должны попасть на следующий — транспортный — уровень, но функционально являются протоколами сетевого уровня, и поэтому их невозможно вписать в модель OSI.

Пакеты сетевого протокола IP могут содержать код, указывающий, какой именно протокол следующего уровня нужно использовать, чтобы извлечь данные из пакета. Это число — уникальный *IP-номер протокола*. ICMP и IGMP имеют номера, соответственно, 1 и 2.

К этому уровню относятся: DVMRP, ICMP, IGMP, MARS, PIM, RIP, RIP2, RSVP

Канальный уровень

Канальный уровень (Link layer) описывает способ кодирования данных для передачи пакета данных на физическом уровне (т.е. специальные последовательности бит, определяющих начало и конец пакета данных, а также обеспечивающие помехоустойчивость). Ethernet, например, в полях заголовка пакета содержит указание того, какой машине или машинам в сети предназначен этот пакет.

Примеры протоколов канального уровня — Ethernet, IEEE 802.11 WLAN, SLIP, Token Ring, ATM и MPLS.

PPP не совсем вписывается в такое определение, поэтому обычно описывается в виде пары протоколов HDLC/SDLC.

MPLS занимает промежуточное положение между канальным и сетевым уровнем и, строго говоря, его нельзя отнести ни к одному из них.

Канальный уровень иногда разделяют на 2 подуровня — LLC и MAC.

Кроме того, канальный уровень описывает среду передачи данных (будь то коаксиальный кабель, витая пара, оптическое волокно или радиоканал), физические характеристики такой среды и принцип передачи данных (разделение каналов, модуляцию, амплитуду сигналов, частоту сигналов, способ синхронизации передачи, время ожидания ответа и максимальное расстояние).

При проектировании стека протоколов на канальном уровне рассматривают помехоустойчивое кодирование — позволяющие обнаруживать и исправлять ошибки в данных вследствие воздействия шумов и помех на канал связи.

D-Bus

D-Bus — система межпроцессного взаимодействия, которая позволяет приложениям в операционной системе сообщаться друг с другом.

D-Bus является частью проекта freedesktop.org. Она обладает высокой скоростью работы, не зависит от рабочей среды, работает на POSIX-совместимых операционных системах, также существует версия для Windows (пока на стадии разработки). Состоит из двух частей: демона и низкоуровневого API. Существуют высокоуровневые библиотеки для фреймворков Qt, Java, GLib, C#, Python, Ruby и библиотека для C++.

Приложения одной среды рабочего стола должны тесно взаимодействовать между собой. В графической среде KDE для этого не так давно использовался DCOP, но другие настольные среды (например, GNOME) не имели аналогичных систем.

Существовала возможность сообщения посредством CORBA, SOAP или XML-RPC, но CORBA больше подходит для систем уровня предприятия, чем для графических сред (KDE и GNOME прошли этап его использования за время своего существования), а SOAP и XML-RPC предназначены для веб-служб.

Раньше GNOME использовал Bonobo, основанный на CORBA, но из-за зависимости от GObject, Bonobo не использовался в других рабочих средах, а низкое быстродействие CORBA сказывалось на скорости всей среды.

Требовалось организовать обмен сообщениями между приложениями двух разных сред. Для решения этой задачи и был создан проект D-Bus. Реализация оказалась удачной и впоследствии было решено проект KDE 4 перевести на использование D-Bus.

D-Bus предоставляет системе несколько шин:

Системная шина. Создается при старте демона D-Bus. С ее помощью происходит общение различных демонов, таких как UPower, а также взаимодействие пользовательских приложений с этими демонами.

Сессионная шина. Создается для пользователя, авторизировавшегося в системе. Для каждой такой шины запускается отдельная копия демона, посредством нее будут общаться приложения, с которыми работает пользователь.

Каждое сообщение D-Bus, передаваемое по шине, имеет своего отправителя. В случае, если сообщение не является широковещательным сигналом, то оно имеет и получателя. Адреса отправителей и получателей называются путями объектов, поскольку D-Bus предполагает, что каждое приложение состоит из набора объектов, а сообщения пересылаются не между приложениями, а между объектами этих самых приложений.

Каждый объект может поддерживать один или более интерфейсов, которые представлены здесь в виде именованных групп методов и сигналов — аналогично интерфейсам Glib, Qt или Java.

D-Bus также предусматривает концепцию сервисов. *Сервис* — уникальное местоположение приложения на шине. При запуске приложение регистрирует один или несколько сервисов, которыми оно будет владеть до тех пор, пока самостоятельно не освободит, до этого момента никакое другое приложение, претендующее на тот же сервис, занять его не сможет. Именуются сервисы аналогично интерфейсам. После закрытия приложения ассоциированные сервисы также разрегистрируются, а D-Bus посылает сигнал о том, что сервис закрыт.

Сервисы делают доступной еще одну функцию — запуск необходимых приложений в случае поступления сообщений для них. Для этого должна быть включена автоактивация, а в конфигурации D-Bus за этим сервисом должно быть закреплено одно приложение.

После подключения к шине приложение должно указать, какие сообщения оно желает получать путем добавления масок совпадений (matchers). Маски представляют собой наборы правил для сообщений, которые будут доставляться приложению. Фильтрация может основываться на интерфейсах, путях объектов и методах.

Сообщения в D-Bus бывают четырех видов: вызовы методов, результаты вызовов методов, сигналы (широковещательные сообщения) и ошибки.

В D-Bus у каждого объекта свое, уникальное имя, которое выглядит как путь в файловой системе. Например, объект может быть именован как */org/kde/kspread/sheets/3/cells/4/5*. Предпочтительны имена, которые несут какую-либо смысловую нагрузку, тем не менее разработчики могут выбрать и имя */com/mycompany/c5yo817y0c1y1c5b*, если это имеет смысл для их приложения.

Имена объектов находятся в пространствах имен, чтобы обеспечить разграничение разных программных модулей. Пространствам имен обычно дается префикс, специфичный для разработчика, например */org/kde*.

Сокеты Беркли (Berkley)

Сокеты Беркли — интерфейс программирования приложений (API), представляющий собой библиотеку для разработки приложений на языке С с поддержкой межпроцессного взаимодействия, часто применяемый в компьютерных сетях.

Сокеты Беркли (также известные как API сокетов BSD) впервые появились как API в операционной системе 4.1BSD Unix (выпущенной в 1982 г.). Тем не менее, только в 1989 г. Калифорнийский университет в Беркли смог начать выпускать версии операционной системы и сетевой библиотеки без лицензионных ограничений AT&T, действующих в защищенной авторским правом Unix.

API сокетов Беркли сформировал фактически стандарт абстракции для сетевых сокетов. Большинство прочих языков программирования используют интерфейс, схожий с API языка Си.

API Интерфейса транспортного уровня (TLI), основанный на STREAMS, представляет собой альтернативу сокетному API. Тем не менее, API сокетов Беркли значительно преобладает в популярности и количестве реализаций.

Интерфейс сокета Беркли — API, позволяющий реализовывать взаимодействие между компьютерами или между процессами на одном компьютере. Данная технология может работать со множеством различных устройств ввода-вывода и драйверов, несмотря на то что их поддержка зависит от реализации операционной системы. Подобная реализация интерфейса лежит в основе TCP/IP, благодаря чему считается одной из фундаментальных технологий, на которых основывается Интернет. Технология сокетов впервые была разработана в Калифорнийском университете Беркли для применения на UNIX-системах. Все современные операционные системы имеют ту или иную реализацию интерфейса сокетов Беркли, так как это стало стандартным интерфейсом для подключения к сети Интернет.

Программисты могут получать доступ к интерфейсу сокетов на трех различных уровнях, наиболее мощным и фундаментальным из

которых является уровень сырых сокетов. Довольно небольшое число приложений нуждается в ограничении контроля над исходящими соединениями, реализуемыми ими, поэтому поддержка сырых сокетов задумывалась быть доступной только на компьютерах, применяемых для разработки на основе технологий, связанных с Интернетом. Впоследствии в большинстве операционных систем была реализована их поддержка, включая Windows XP.

Заголовочные файлы

Программная библиотека сокетов Беркли включает в себя множество связанных заголовочных файлов.

<sys/socket.h>

Базовые функции сокетов BSD и структуры данных.

<netinet/in.h>

Семейства адресов/протоколов PF_INET и PF_INET6. Широко используются в сети Интернет, включают в себя IP-адреса, а также номера портов TCP и UDP.

<sys/un.h>

Семейство адресов PF_UNIX/PF_LOCAL. Используется для локального взаимодействия между программами, запущенными на одном компьютере. В компьютерных сетях не применяется.

<arpa/inet.h>

Функции для работы с числовыми IP-адресами.

<netdb.h>

Функции для преобразования протокольных имен и имен хостов в числовые адреса. Используются локальные данные аналогично DNS.

Структуры

sockaddr — обобщенная структура адреса, к которой, в зависимости от используемого семейства протоколов, приводится соответствующая структура, например:

```
struct sockaddr_in stSockAddr;
...
bind(SocketFD,(const struct sockaddr *)&stSockAddr, sizeof(struct sockaddr_in));
```

sockaddr_in
sockaddr_in6
in_addr
in6_addr

Функции

socket()

`socket()` создает конечную точку соединения и возвращает дескриптор. `socket()` принимает три аргумента:

domain, указывающий семейство протоколов создаваемого сокета. Этот параметр задает правила использования именования и формат адреса. Например:

`PF_INET` для сетевого протокола IPv4 или

`PF_INET6` для IPv6.

`PF_UNIX` для локальных сокетов (используя файл);

type (тип) один из:

`SOCK_STREAM` надежная потокоориентированная служба (TCP) (сервис) или потоковый сокет

`SOCK_DGRAM` служба датаграмм (UDP) или датаграммный сокет

`SOCK_SEQPACKET` надежная служба последовательных пакетов

`SOCK_RAW` Сырой сокет — сырой протокол поверх сетевого уровня;

protocol определяет используемый транспортный протокол. Самые распространенные — это `IPPROTO_TCP`, `IPPROTO_SCTP`, `IPPROTO_UDP`, `IPPROTO_DCCP`. Эти протоколы указаны в <netinet/in.h>. Значение «0» может быть использовано для выбора протокола по умолчанию из указанного семейства (`domain`) и типа (`type`).

Функция возвращает –1 в случае ошибки. Иначе она возвращает целое число, представляющее присвоенный дескриптор.

Прототип

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

gethostbyname() и gethostbyaddr()

Функции `gethostbyname()` и `gethostbyaddr()` возвращают указатель на объект типа *struct hostent*, описывающий интернет-узел по имени или по адресу соответственно. Эта структура содержит или информацию, полученную от сервера имен, или произвольные поля из строки в /etc/hosts. Если локальный сервер имен не запущен, то эти подпрограммы просматривают /etc/hosts. Функции принимают следующие аргументы:

name, определяющий имя хоста. Например: www.wikipedia.org;

addr, определяющий указатель на *struct in_addr*, содержащую адрес хоста;

len, определяющий длину в байтах *addr*;

type, определяющий тип области адресов хоста. Например: PF_INET ‘‘

Функции возвращают NULL-указатель в случае ошибки. В этом случае может быть проверена дополнительная целая *h_errno* для выявления ошибки или неправильного или неизвестного хоста. В противном случае возвращается корректная *struct hostent **.

Прототип

```
struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyaddr(const void *addr, int len, int type);
```

connect()

`connect()` Устанавливает соединение с сервером. Возвращает целое число, представляющее код ошибки: 0 означает успешное выполнение, а –1 свидетельствует об ошибке.

Некоторые типы сокетов работают без установления соединения, это в основном касается UDP-сокетов. Для них соединение приобретает особое значение: цель по умолчанию для посылки и получения данных присваивается переданному адресу, позволяя использовать такие функции, как `send()` и `recv()`, на сокетах без установления соединения.

Загруженный сервер может отвергнуть попытку соединения, поэтому в некоторых видах программ необходимо предусмотреть повторные попытки соединения.

Прототип

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
```

bind()

`bind()` связывает сокет с конкретным адресом. Когда сокет создается при помощи `socket()`, он ассоциируется с некоторым семейством адресов, но не с конкретным адресом. До того как сокет сможет принять входящие соединения, он должен быть связан с адресом. `bind()` принимает три аргумента:

`sockfd` — дескриптор, представляющий сокет при привязке;

`serv_addr` — указатель на структуру `sockaddr`, представляющую адрес, к которому привязываем;

`addrlen` — поле `socklen_t`, представляющее длину структуры `sockaddr`.

Возвращает 0 при успехе и –1 при возникновении ошибки.

Прототип

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen);
```

listen()

`listen()` подготавливает привязываемый сокет к принятию входящих соединений. Данная функция применима только к типам сокетов `SOCK_STREAM` и `SOCK_SEQPACKET`. Принимает два аргумента:

`sockfd` — корректный дескриптор сокета;

`backlog` — целое число, означающее число установленных соединений, которые могут быть обработаны в любой момент времени. Операционная система обычно ставит его равным максимальному значению.

После принятия соединения оно выводится из очереди. В случае успеха возвращается 0, в случае возникновения ошибки возвращается –1.

Прототип

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

accept()

`accept()` используется для принятия запроса на установление соединения от удаленного хоста. Принимает следующие аргументы:

`sockfd` — дескриптор слушающего сокета на принятие соединения;

`cliaddr` — указатель на структуру `sockaddr`, для принятия информации об адресе клиента;

`addrlen` — указатель на `socklen_t`, определяющее размер структуры, содержащей клиентский адрес и переданной в `accept()`. Когда `accept()` возвращает некоторое значение, `socklen_t` указывает, сколько байт структуры `cliaddr` использовано в данный момент.

Функция возвращает дескриптор сокета, связанный с принятым соединением, или –1 в случае возникновения ошибки.

Прототип

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen).
```

Дополнительные параметры для сокетов

После создания сокета можно задавать для него дополнительные параметры. Вот некоторые из них:

`TCP_NODELAY` отключает алгоритм Нейгла;

`SO_KEEPALIVE` включает периодические проверки на наличие 'признаков жизни', если это поддерживается ОС.

Блокирующие и неблокирующие сокеты

Сокеты Беркли могут работать в одном из двух режимов: блокирующем или неблокирующем. *Блокирующий* сокет не возвращает контроль, пока не отошлет (или пока не получит) все данные, указанные для операции. Это верно лишь для Linux-систем. В других системах, например во FreeBSD, вполне естественно для блокирующего сокета посылать не все данные (но можно поставить в send() или recv() флаг MSG_WAITALL). Приложение должно проверять возвращаемое значение для отслеживания того, сколько байт было послано/получено и, соответственно, перепосылать необработанную на данный момент информацию. Это может привести к проблемам, если сокет продолжает «слушать»: программа может повиснуть из-за того, что сокет ждет данных, которые могут никогда не прибыть.

Сокет обычно указывается блокирующим или неблокирующим при помощи функций `fcntl()` или `ioctl()`.

Передача данных

Для передачи данных можно пользоваться стандартными функциями чтения/записи файлов `read` и `write`, но есть специальные функции для передачи данных через сокеты: send, recv, sendto, recvfrom, sendmsg, recvmsg.

Нужно обратить внимание, что при использовании протокола TCP (сокеты типа `SOCK_STREAM`) есть вероятность получить меньше данных, чем было передано, так как еще не все данные были приняты, поэтому нужно либо дождаться, когда функция `recv` возвратит 0 байт, либо выставить флаг `MSG_WAITALL` для функции `recv`, что заставит ее дождаться окончания передачи. Для остальных типов сокетов флаг `MSG_WAITALL` ничего не меняет (например, в UDP весь пакет = целое сообщение).

Высвобождение ресурсов

Система не освобождает ресурсы, выделенные при вызове `socket()`, пока не произойдет вызова `close()`. Это особенно важно в случае, если вызов `connect()` прошел неудачно и может быть повторен. Каждый вызов `socket()` должен иметь соответствующий вызов `close()` во всех возможных путях исполнения. Необходимо добавлять заголовочный файл <unistd.h> для поддержки функции закрытия.

Результатом выполнения системного вызова `close()` является только обращение к интерфейсу для закрытия сокета, а не закрытие самого сокета. Это является командой для ядра закрыть сокет. Иногда на серверной стороне сокет может перейти в режим ожидания `TIME_WAIT` длительностью до 4 минут.

Список использованной литературы

1. *A. Silberschatz, P.B. Galvin, G. Gagne.* Operating System Concepts, 6th ed. — New York: Wiley, 2002 — 912 с.
2. Department of Defense Trusted Computer System Evaluation Criteria, TCSEC, DoD 5200.28-STD, December 26, 1985.
3. *Peter J. Denning.* Before memory was virtual. — Draft, June, 6th ed. 1996. — 16 с.
4. *Ахо А.В.* Структуры данных и алгоритмы [Текст] / А.В. Ахо, Д. Хопкрофт, Д.Д. Ульман. — М.: Вильямс, 2001. — 384 с.
5. *Бах М.Д.* Архитектура операционной системы UNIX [Текст] / М.Д. Бах. — Prentice-Hall, 1986. — 584 с.
6. *Брукшир Д.Г.* Введение в компьютерные науки. Общий обзор [Текст] / Д.Г. Брукшир. — М.: Вильямс, 2001. — 688 с.
7. *Дейтел Г.* Введение в операционные системы [Текст] / Г. Дейтел: в 2 т. Т. 1. — М.: Мир, 1987. — 359 с.
8. *Дейтел Г.* Введение в операционные системы [Текст] / Г. Дейтел: в 2 т. Т. 2. — М.: Мир, 1987. — 398 с.
9. *Кузнецов С.Д.* Операционная система UNIX [Электронный ресурс]: учеб. пособие / С.Д. Кузнецов // Операционная система UNIX, 1999. — URL: http://citforum.ru/operating_systems/unix/contents.shtml
10. *Лав Р.* Ядро Linux. Описание процесса разработки [Текст] / Р. Лав. — М.: Вильямс, 2014. — 496 с.
11. *Олифер В.* Компьютерные сети. Принципы, технологии, протоколы [Текст] / В. Олифер, Н. Олифер. — 5-е изд. — СПб.: Питер, 2016. — 992 с.
12. *Олифер В.* Сетевые операционные системы [Текст] / В. Олифер, Н. Олифер. — СПб.: Питер, 2001. — 544 с.
13. *Робачевский А.М.* Операционная система UNIX [Текст] / А.М. Робачевский. — СПб.: БХВ, 1999. — 528 с.
14. *Столлингс В.* Криптография и защита сетей. Принципы и практика [Текст] / В. Столлингс. — М.: Вильямс, 2001. — 672 с.
15. *Таненбаум Э.* Современные операционные системы [Текст] / Э. Таненбаум. — 2-е изд. — СПб.: Питер, 2002. — 1040 с.
16. *Таненбаум Э.* Современные операционные системы [Текст] / Э. Таненбаум, Х. Бос. — 4-е изд. — СПб.: Питер, 2015. — 1120 с.
17. *Таненбаум Э.* Архитектура компьютера [Текст] / Э. Таненбаум, Т. Остин. — 6-е изд. — СПб.: Питер, 2017. — 816 с.
18. *Таненбаум Э.* Компьютерные сети [Текст] / Э. Таненбаум, Д. Уэзеролл. — 5-е изд. — СПб.: Питер, 2016. — 960 с.
19. *Цикритзис Д.* Операционные системы [Текст] / Д. Цикритзис, Ф. Бернстайн. — М.: Мир, 1977. — 336 с.

СОДЕРЖАНИЕ

Учебное издание

Баранчиков Павел Алексеевич,
Баринов Иван Валерьевич,
Коротаев Александр Николаевич

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Учебник

Оригинал-макет подготовлен в Издательстве «КУРС»

Подписано в печать 10.01.2018.
Формат 60×90/16. Бумага офсетная. Гарнитура Newton.
Печать цифровая. Усл. печ. л. 18,0.
Тираж 500 экз. Заказ № 889

ТК 677986-948063-100118

ООО Издательство «КУРС»
127273, Москва, ул. Олонецкая, д. 17А, офис 104.
Тел.: (495) 203-57-83.
E-mail: kursizdat@gmail.com http://www. kursizdat.ru

Для заметок

4757766egll