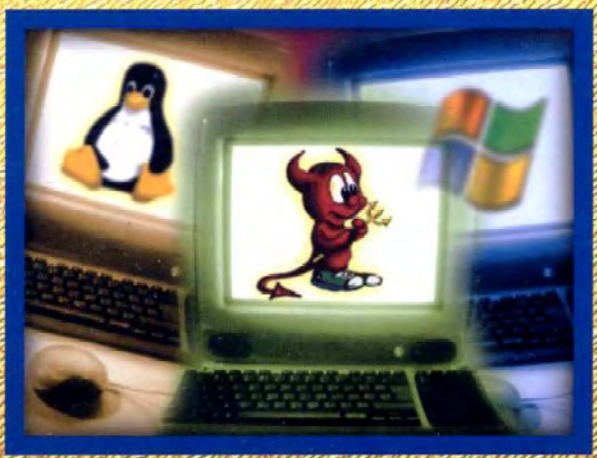
УЧЕБНИК ДЛЯ ВУЗОВ

А. В. Гордеев

ОПЕРАЦИОННЫЕ СИСТЕМЫ

2-е издание



ПИТЕР®

УЧЕБНИК / ДЛЯ ВУЗОВ

А. В. Гордеев

ОПЕРАЦИОННЫЕ СИСТЕМЫ

2-е издание

Допущено Министерством образования и науки Российской Федерации в качестве учебника для студентов высших учебных заведений, обучающихся по направлению подготовки бакалавров и магистров «Информатика и вычислительная техника» и направлению подготовки дипломированных специалистов «Информатика и вычислительная техника»

Издательская программа

300 лучших учебников для высшей школы

осуществляется при поддержке Министерства образования и науки РФ

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2007

ББК 32.973-018.2я7
УДК 681.3.066(075)
Г68

Рецензенты:

Немолочнов О. Ф. — доктор технических наук, профессор, заведующий кафедрой информатики и прикладной математики Санкт-Петербургского университета информационных технологий, механики и оптики

Трофимов В. В. — доктор технических наук, профессор, академик МАИ, заведующий кафедрой информатики Санкт-Петербургского государственного университета экономики и финансов

Гордеев А. В.

Г68 Операционные системы: Учебник для вузов. 2-е изд. — СПб.: Питер, 2007. — 416 с.: ил.

ISBN 978-5-94723-632-3

В учебнике излагаются основные понятия операционных систем, принципы их построения и функционирования. Помимо рассмотрения таких обязательных тем, как управление задачами и ресурсами в операционных системах, организация параллельных взаимодействующих вычислений и связанных с этим проблем, приводятся сведения об особенностях архитектур современных операционных систем, используемых на персональных компьютерах.

Допущено Министерством образования Российской Федерации в качестве учебника для студентов высших учебных заведений, обучающихся по направлению подготовки бакалавров и магистров «Информатика и вычислительная техника» и направлению подготовки дипломированных специалистов «Информатика и вычислительная техника».

ББК 32.973-018.2я7

УДК 681.3.066(075)

ISBN 978-5-94723-632-3

Краткое содержание

Введение 8
Глава 1. Основные понятия 11
Глава 2. Управление задачами 50
Глава 3. Управление памятью в операционных системах 72
Глава 4. Особенности архитектуры микропроцессоров i80x86 для организации мультипрограммных операционных систем 101
Глава 5. Управление вводом-выводом в операционных системах 130
Глава 6. Файловые системы 163
Глава 7. Организация параллельных взаимодействующих вычислений 209
Глава 8. Проблема тупиков и методы борьбы с ними 247
Глава 9. Архитектура операционных систем 278
Глава 10. Краткий обзор современных операционных систем 311
Глава 11. Операционные системы Windows 361
Список терминов 396
Список литературы 406
Алфавитный указатель 409

Содержание

Введение 8

От издательства 10

Глава 1. Основные понятия 11

Назначение и функции операционных систем 11
Понятие операционной среды 16
Прерывания 18
Понятия вычислительного процесса и ресурса 25
Мультипрограммирование, многопользовательский режим работы и режим разделения времени 27
Диаграмма состояний процесса 30
Реализация понятия последовательного процесса в операционных системах 34
Процессы и задачи 36
Основные виды ресурсов и возможности их разделения 42
Классификация операционных систем 46
Контрольные вопросы и задачи 49

Глава 2. Управление задачами 50

Планирование и диспетчеризация процессов и задач 52
Планирование вычислительных процессов и стратегии планирования 53
Дисциплины диспетчеризации 55
Качество диспетчеризации и гарантии обслуживания 63
Диспетчеризация задач с использованием динамических приоритетов 65
Контрольные вопросы и задачи 71

Глава 3. Управление памятью в операционных системах 72

Память и отображения, виртуальное адресное пространство 73
Простое непрерывное распределение и распределение с перекрытием 76
Общие принципы управления памятью в однопрограммных операционных системах 76
Распределение оперативной памяти в MS DOS 78
Распределение памяти статическими и динамическими разделами 82
Разделы с фиксированными границами 82
Разделы с подвижными границами 85

Сегментная, страничная и сегментно-страничная организация памяти ... 86
Сегментный способ организации виртуальной памяти ... 87
Страничный способ организации виртуальной памяти ... 93
Сегментно-страничный способ организации виртуальной памяти ... 97
Контрольные вопросы и задачи ... 99

Глава 4. Особенности архитектуры микропроцессоров i80x86 для организации мультипрограммных операционных систем ... 101

Реальный и защищенный режимы работы процессора ... 101
Новые системные регистры микропроцессоров i80x86 ... 103
Адресация в 32-разрядных микропроцессорах i80x86 при работе в защищенном режиме ... 105
Поддержка сегментного способа организации виртуальной памяти ... 105
Поддержка страничного способа организации виртуальной памяти ... 110
Режим виртуальных машин для исполнения приложений реального режима ... 113
Защита адресного пространства задач ... 115
Уровни привилегий для защиты адресного пространства задач ... 115
Механизм шлюзов для передачи управления на сегменты кода с другими уровнями привилегий ... 118
Система прерываний 32-разрядных микропроцессоров i80x86 ... 122
Работа системы прерываний в реальном режиме ... 122
Работа системы прерываний в защищенном режиме ... 124
Контрольные вопросы и задачи ... 128

Глава 5. Управление вводом-выводом в операционных системах ... 130

Основные концепции организации ввода-вывода в операционных системах ... 131
Режимы управления вводом-выводом ... 134
Закрепление устройств, общие устройства ввода-вывода ... 136
Основные системные таблицы ввода-вывода ... 138
Синхронный и асинхронный ввод-вывод ... 143
Организация внешней памяти на магнитных дисках ... 145
Основные понятия ... 145
Логическая структура магнитного диска ... 146
Системный загрузчик Windows NT/2000/XP ... 155
Кэширование операций ввода-вывода при работе с накопителями на магнитных дисках ... 156
Контрольные вопросы и задачи ... 161
Вопросы для проверки ... 161
Задания ... 162

Глава 6. Файловые системы ... 163

Функции файловой системы и иерархия данных ... 163
Файловая система FAT ... 166
Таблица размещения файлов ... 167
Структура загрузочной записи DOS ... 170
Файловые системы VFAT и FAT32 ... 171
Файловая система HPFS ... 177
Файловая система NTFS ... 188
Основные возможности файловой системы NTFS ... 189
Структура тома с файловой системой NTFS ... 190
Разрешения NTFS ... 194

Контрольные вопросы и задачи ... 207
Вопросы для проверки ... 207
Задания ... 208

Глава 7. Организация параллельных взаимодействующих вычислений ... 209

Независимые и взаимодействующие вычислительные процессы ... 209
Средства синхронизации и связи взаимодействующих вычислительных процессов ... 215
Использование блокировки памяти при синхронизации параллельных процессов ... 215
Семафорные примитивы Дейкстры ... 224
Мьютексы ... 229
Использование семафоров при проектировании взаимодействующих вычислительных процессов ... 230
Мониторы Хоара ... 236
Почтовые ящики ... 240
Конвейеры и очереди сообщений ... 242
Конвейеры ... 242
Очереди сообщений ... 244
Контрольные вопросы и задачи ... 246

Глава 8. Проблема тупиков и методы борьбы с ними ... 247

Понятие тупиковой ситуации при выполнении параллельных вычислительных процессов ... 247
Примеры тупиковых ситуаций и причины их возникновения ... 249
Пример тупика на ресурсах типа CR ... 250
Пример тупика на ресурсах типа CR и SR ... 251
Пример тупика на ресурсах типа SR ... 252
Формальные модели для изучения проблемы тупиковых ситуаций ... 254
Сети Петри ... 254
Модель пространства состояний системы ... 259
Методы борьбы с тупиками ... 263
Предотвращение тупиков ... 263
Обход тупиков ... 264
Обнаружение тупика ... 267
Контрольные вопросы и задачи ... 277

Глава 9. Архитектура операционных систем ... 278

Основные принципы построения операционных систем ... 279
Принцип модульности ... 279
Принцип особого режима работы ... 280
Принцип виртуализации ... 281
Принцип мобильности ... 283
Принцип совместимости ... 285
Принцип генерируемости ... 286
Принцип открытости ... 287
Принцип обеспечения безопасности вычислений ... 287
Микроядерные операционные системы ... 289
Макроядерные операционные системы ... 292
Требования к операционным системам реального времени ... 293
Мультипрограммность и мультизадачность ... 294
Приоритеты задач ... 294
Наследование приоритетов ... 295
Сихронизация процессов и задач ... 295
Предсказуемость ... 296
Интерфейсы операционных систем ... 296

Интерфейс прикладного программирования ... 298
Реализация функций API на уровне модулей операционной системы ... 299
Реализация функций API на уровне системы программирования ... 300
Реализация функций API с помощью внешних библиотек ... 302
Интерфейс POSIX ... 304
Примеры программирования для разных интерфейсов API ... 307
Контрольные вопросы и задачи ... 310

Глава 10. Краткий обзор современных операционных систем ... 311

Семейство операционных систем UNIX ... 312
Общая характеристика и особенности архитектуры ... 312
Основные понятия ... 314
Функционирование ... 320
Файловая система ... 323
Взаимодействие между процессами ... 329
Операционная система Linux ... 336
Операционная система FreeBSD ... 339
Сетевая операционная система реального времени QNX ... 340
Архитектура системы QNX ... 342
Основные механизмы организации распределенных вычислений ... 345
Семейство операционных систем OS/2 Warp компании IBM ... 351
Особенности архитектуры и основные возможности ... 354
Особенности интерфейсов ... 357
Серверная операционная система OS/2 Warp 4.5 ... 359
Контрольные вопросы и задачи ... 360

Глава 11. Операционные системы Windows ... 361

Операционные системы Windows 9x ... 363
Краткая историческая справка ... 363
Общие сведения ... 365
Организация многозадачности ... 369
Распределение оперативной памяти ... 373
Операционные системы Windows NT/2000/XP ... 378
Краткая историческая справка ... 378
Основные особенности архитектуры ... 382
Модель безопасности ... 387
Распределение оперативной памяти ... 390
Контрольные вопросы и задачи ... 394
Вопросы для проверки ... 394
Задания ... 395

Список терминов ... 396

Список литературы ... 406

Алфавитный указатель ... 409

Введение

Как известно, процесс проникновения информационных технологий практически во все сферы человеческой деятельности продолжает развиваться и углубляться. Помимо уже привычных и широко распространенных персональных компьютеров, общее число которых достигло многих сотен миллионов, становится все больше и встроенных средств вычислительной техники. Пользователей всей этой разнообразной вычислительной техники становится все больше, причем наблюдается развитие двух вроде бы противоположных тенденций. С одной стороны, информационные технологии все усложняются, и для их применения, и тем более дальнейшего развития, требуется иметь очень глубокие познания. С другой стороны, упрощаются интерфейсы взаимодействия пользователей с компьютерами. Компьютеры и информационные системы становятся все более дружественными и понятными даже для человека, не являющегося специалистом в области информатики и вычислительной техники. Это стало возможным прежде всего потому, что пользователи и их программы взаимодействуют с вычислительной техникой посредством специального (системного) программного обеспечения — через операционную систему.

Операционная система предоставляет интерфейсы и для выполняющихся приложений, и для пользователей. Программы пользователей, да и многие служебные программы запрашивают у операционной системы выполнение тех операций, которые достаточно часто встречаются практически в любой программе. К таким операциям, прежде всего, относятся операции ввода-вывода, запуск или останов какой-нибудь программы, получение дополнительного блока памяти или его освобождение и многие другие. Подобные операции невыгодно каждый раз программировать заново и непосредственно размещать в виде двоичного кода в теле программы, их удобнее собрать вместе и предоставлять для выполнения по запросу из программ. Это и есть одна из важнейших функций операционных систем. Прикладные программы, да и многие системные обрабатывающие программы (такие, например, как системы программирования или системы управления базами данных), не имеют непосредственного доступа к аппаратуре компьютера, а взаимодействуют с ней только через обращения к операционной системе. Пользователи также путем ввода команд операционной системы или выбором возможных дей-

ствий, предлагаемых системой, взаимодействуют с компьютером и своими программами. Такое взаимодействие осуществляется исключительно через операционную систему. Помимо выполнения этой важнейшей функции операционные системы отвечают за эффективное распределение вычислительных ресурсов и организацию надежных вычислений.

Знание основ организации операционных систем и принципов их функционирования позволяет использовать компьютеры более эффективно. Глубокое изучение операционных систем позволяет применить эти знания прежде всего при создании программного обеспечения. Если, к большому сожалению, в нашей стране в последние годы практически не создаются новые операционные системы, то разработки сложных информационных систем, комплексов программ и отдельных приложений, предназначенных для работы в широко распространенных операционных системах, ведутся достаточно интенсивно, причем большим числом организаций. И здесь знание операционных систем, принципов их функционирования, методов организации вычислений является не только желательным, но обязательным.

Дисциплина «Операционные системы» является одной из важнейших. Она включена в Государственный образовательный стандарт по направлению 654600 — «Информатика и вычислительная техника» и отнесена к блоку общепрофессиональных дисциплин. В рамках этого направления имеется несколько специальностей, в том числе 220100 — «Вычислительные машины, комплексы, системы и сети», 220200 — «Автоматизированные системы обработки информации и управления», 220300 — «Системы автоматизированного проектирования», 220400 — «Программное обеспечение вычислительной техники и автоматизированных систем». Именно для студентов вузов, обучающихся по этим специальностям, и предназначается настоящая книга. Однако она может быть востребована и студентами других специальностей, изучающих информатику и вычислительную технику, а также обычными подготовленными пользователями, желающими углубить свои познания в области операционных систем, ибо сегодня уже мало просто уметь работать на компьютере, а желательно понимать, как он работает, как организуются в нем вычисления. Знания основных принципов организации вычислительных процессов, понимание проблем, которые при этом возникают, и методов их решения позволяют обдуманно подходить к использованию компьютера, предусмотреть и предотвратить нежелательные явления. Помимо общетеоретических в книге рассмотрены и отдельные практические вопросы, описаны конкретные реализации отдельных модулей и подсистем.

Учебный материал, ставший основой для настоящей книги, уже в течение нескольких лет читается студентам специальности 220100 в Санкт-Петербургском государственном университете аэрокосмического приборостроения. Материал построен с учетом упомянутого Государственного образовательного стандарта по направлению «Информатика и вычислительная техника», регламентирующего содержание дисциплины «Операционные системы». В основу издания легла переработанная первая часть учебника «Системное программное обеспечение», вышедшего в издательстве «Питер» в 2001 году и используемого в учебном процессе во

многих вузах. При работе над рукописью автор постарался учесть те советы и замечания от коллег по цеху, которые были получены после выхода в свет книги «Системное программное обеспечение».

Напоследок хочется высказать самые теплые слова благодарности всем тем, кто принял участие в подготовке этой книги к изданию. Это и Андрей Васильев, которого, к сожалению, уже больше нет среди нас, и его замечательные коллеги — сотрудники издательства «Питер». Своим кропотливым трудом, вниманием и доброжелательным отношением они помогли преодолеть возникшие трудности. Работа над книгой — длительный процесс: хочется улучшить то одно, то другое, переписать или добавить, а времени свободного, да еще в достаточном количестве, как всегда, нет. В связи с этим хочется также поблагодарить своих родных и близких за долготерпение, доброжелательность и сердечную заботу в течение всего времени работы над рукописью. Без их поддержки эта книга, скорее всего, не состоялась бы.

От издательства

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

Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на web-сайте издательства http://www.piter.com.

Глава 1. Основные понятия

Эта глава является вводной и, пожалуй, самой главной. Любой предмет имеет свои основные понятия и положения. Не является исключением и дисциплина «Операционные системы». К основным понятиям, без которых практически невозможно по-настоящему изучить эту дисциплину, понять основные принципы организации вычислений, взаимодействия прикладных программ с операционной системой и пользователей с компьютерами, следует, прежде всего, отнести понятия вычислительных процессов и ресурсов, системной программы, супервизора, операционной среды, прерываний. Мы также рассмотрим относительно новые понятия, к которым относятся поток выполнения и задача; они дополняют понятие вычислительного процесса и позволяют более эффективно организовать работу компьютера. Поскольку абсолютное большинство операционных систем обеспечивают возможность параллельного выполнения нескольких программ, мы познакомимся с понятием мультипрограммирования. Завершается глава обзором основных общепринятых классификаций.

Назначение и функции операционных систем

Операционные системы относятся к системному программному обеспечению. Как известно, все программное обеспечение разделяется на системное и прикладное. К системному программному обеспечению принято относить такие программы и комплексы программ, которые являются общими, без которых невозможно выполнение или создание других программ. История появления и развития системного программного обеспечения началась с того момента, когда люди осознали, что любая программа требует операций ввода-вывода данных. Это произошло в далекие 50-е годы прошлого столетия. Собственно операционные системы появились чуть позже.

Действительно, если мы не будем иметь возможности изменять исходные данные и получать результаты вычислений, то зачем вообще эти вычисления? Очевидно, что исходные данные могут вводиться различными способами. На практике используются самые разнообразные устройства и методы. Например, мы можем вво-

дить исходные значения с клавиатуры, задавать нужные действия или функции с помощью указателя мыши, считывать записи из файла, снимать оцифрованные значения с датчиков и т. д. Часть исходных данных может быть передана в программу через область памяти, в которую предварительно другая программа занесла свои результаты вычислений. Способов много. Главное — выполнить в программе некоторые действия, связанные с получением исходных данных.

Аналогично, и вывод результатов может быть организован, например, на соответствующие устройства и в форме, удобной для восприятия ее человеком. Либо результаты расчетов будут отправляться программой на какие-нибудь исполнительные устройства, которые управляются компьютером. Наконец, мы можем организовать запись полученных значений на некие устройства хранения данных (с целью их дальнейшей обработки).

Программирование операций ввода-вывода относится к одной из самых трудоемких областей создания программного обеспечения. Здесь речь идет не об использовании операторов типа `READ` или `WRITE` в языках высокого уровня. Речь идет о необходимости создать подпрограмму в машинном виде, уже готовую к выполнению на компьютере, а не написанную с помощью некоторой системы программирования (систем программирования тогда еще не было), подпрограмму, вместо обычных вычислений управляющую тем устройством, которое должно участвовать в операциях ввода исходных данных или вывода результатов. При наличии такой подпрограммы программист может обращаться к ней столько раз, сколько операций ввода-вывода с этим устройством ему требуется. Для выполнения этой работы программисту недостаточно хорошо знать архитектуру вычислительного комплекса и уметь создавать программы на языке ассемблера. Он должен отлично знать и интерфейс, с помощью которого устройство подключено к центральной части компьютера, и алгоритм функционирования устройства управления устройства ввода-вывода.

Очевидно, что имело смысл создать набор подпрограмм управления операциями ввода-вывода и использовать его в своих программах, чтобы не заставлять программистов каждый раз заново программировать все эти операции. С этого и началась история системного программного обеспечения. Впоследствии набор подпрограмм ввода-вывода стали организовывать в виде специальной библиотеки ввода-вывода, а затем появились и сами операционные системы. Основной причиной их появления было желание автоматизировать процесс подготовки вычислительного комплекса к выполнению программы.

В 50-е годы взаимодействие пользователей с вычислительным комплексом было совершенно иным, чем нынче. Программист-кодер (от англ. coder — кодировщик) — специально подготовленный специалист, знающий архитектуру компьютера и язык(и) программирования, — по заказу составлял текст программы, часто по уже готовому алгоритму, разработанному программистом-алгоритмистом. Текст этой программы затем отдавался оператору, который набирал его на специальных устройствах и переносил на соответствующие носители. Чаще всего в качестве носителей использовались перфокарты или перфолента. Далее колода с перфокартами (перфолента) передавалась в вычислительный зал, где для вычислений по этой программе требовалось выполнить следующие действия.

1. Оператор вычислительного комплекса с пульта вводил в рабочие регистры центрального процессора и в оперативную память компьютера ту первоначальную программу, которая позволяла считать в память программу для трансляции исходных кодов и получения машинной (двоичной) программы (проще говоря, *транслятор*, который тоже хранился на перфокартах или перфоленте).
2. Транслятор считывал исходную программу, осуществлял лексический разбор исходного текста, и промежуточные результаты процесса трансляции зачастую так же выводили на перфокарты (перфоленту). Трансляция — сложный процесс, часто требующий нескольких проходов. Порой для выполнения очередного прохода приходилось в память компьютера загружать с перфокарт и следующую часть транслятора, и промежуточные результаты трансляции. Ведь результат трансляции выводился также на носители информации, поскольку объем оперативной памяти был небольшим, а задача трансляции — это очень сложная задача.
3. Оператор загружал в оперативную память компьютера полученные двоичные коды оттранслированной программы и подгружал двоичные коды тех системных подпрограмм, которые реализовывали управление операциями ввода-вывода. После этого готовая программа, расположенная в памяти, могла сама считывать исходные данные и осуществлять необходимые вычисления.

В случае обнаружения ошибок на одном из этих этапов или после анализа полученных результатов весь цикл необходимо было повторить.

Для автоматизации труда программиста (кодера) стали разрабатывать специальные алгоритмические языки высокого уровня, а для автоматизации труда оператора вычислительного комплекса была разработана специальная управляющая программа, загрузив которую в память один раз оператор мог ее далее использовать неоднократно и более не обращаться к процедуре программирования ЭВМ через пульт оператора. Именно эту управляющую программу и стали называть операционной системой. Со временем на нее стали возлагать все больше и больше задач, она стала расти в объеме. Прежде всего разработчики стремились к тому, чтобы операционная система как можно более эффективно распределяла вычислительные ресурсы компьютера, ведь в 60-е годы операционные системы уже позволяли организовать параллельное выполнение нескольких программ. Помимо задач распределения ресурсов появились задачи обеспечения надежности вычислений. К началу 70-х годов диалоговый режим работы с компьютером стал преобладающим, и у операционных систем стремительно начали развиваться интерфейсные возможности. Напомним, что термином *интерфейс* (interface) обозначают целый комплекс спецификаций, определяющих конкретный способ взаимодействия пользователя с компьютером.

На сегодняшний день можно констатировать, что *операционная система* (ОС) представляет собой комплекс системных[1] управляющих и обрабатывающих программ, которые, с одной стороны, выступают как интерфейс между аппаратурой компью-

[1] Системными принято называть такие программы, которые используются всеми остальными программами.

тера и пользователем с его задачами, а с другой стороны, предназначены для наиболее эффективного расходования ресурсов вычислительной системы и организации надежных вычислений.

Можно попробовать перечислить *основные функции операционных систем.*

- Прием от пользователя (или от оператора системы) заданий, или команд, сформулированных на соответствующем языке, и их обработка. Задания могут передаваться в виде текстовых директив (команд) оператора или в форме указаний, выполняемых с помощью манипулятора (например, с помощью мыши). Эти команды связаны, прежде всего, с запуском (приостановкой, остановкой) программ, с операциями над файлами (получить перечень файлов в текущем каталоге, создать, переименовать, скопировать, переместить тот или иной файл и др.), хотя имеются и иные команды.
- Загрузка в оперативную память подлежащих исполнению программ.
- Распределение памяти, а в большинстве современных систем и организация виртуальной памяти.
- Запуск программы (передача ей управления, в результате чего процессор исполняет программу).
- Идентификация всех программ и данных.
- Прием и исполнение различных запросов от выполняющихся приложений. Операционная система умеет выполнять очень большое количество системных функций (сервисов), которые могут быть запрошены из выполняющейся программы. Обращение к этим сервисам осуществляется по соответствующим правилам, которые и определяют *интерфейс прикладного программирования* (Application Program Interface, API) этой операционной системы.
- Обслуживание всех операций ввода-вывода.
- Обеспечение работы систем управлений файлами (СУФ) и/или систем управления базами данных (СУБД), что позволяет резко увеличить эффективность всего программного обеспечения.
- Обеспечение режима мультипрограммирования, то есть организация параллельного выполнения двух или более программ на одном процессоре, создающая видимость их одновременного исполнения.
- Планирование и диспетчеризация задач в соответствии с заданными стратегией и дисциплинами обслуживания.
- Организация механизмов обмена сообщениями и данными между выполняющимися программами.
- Для сетевых операционных систем характерной является функция обеспечения взаимодействия связанных между собой компьютеров.
- Защита одной программы от влияния другой, обеспечение сохранности данных, защита самой операционной системы от исполняющихся на компьютере приложений.
- Аутентификация и авторизация пользователей (для большинства диалоговых операционных систем). Под *аутентификацией* понимается процедура проверки

имени пользователя и его пароля на соответствие тем значениям, которые хранятся в его учетной записи[1]. Очевидно, что если входное имя (login[2]) пользователя и его пароль совпадают, то, скорее всего, это и будет тот самый пользователь. Термин *авторизация* означает, что в соответствии с учетной записью пользователя, который прошел аутентификацию, ему (и всем запросам, которые будут идти к операционной системе от его имени) назначаются определенные права (привилегии), определяющие, что он может, а что не может делать на компьютере.

- Удовлетворение жестким ограничениям на время ответа в режиме реального времени (характерно для операционных систем реального времени).
- Обеспечение работы систем программирования, с помощью которых пользователи готовят свои программы.
- Предоставление услуг на случай частичного сбоя системы.

Операционная система изолирует аппаратное обеспечение компьютера от прикладных программ пользователей. И пользователь, и его программы взаимодействуют с компьютером через интерфейсы операционной системы. Это можно проиллюстрировать, например, рис. 1.1.

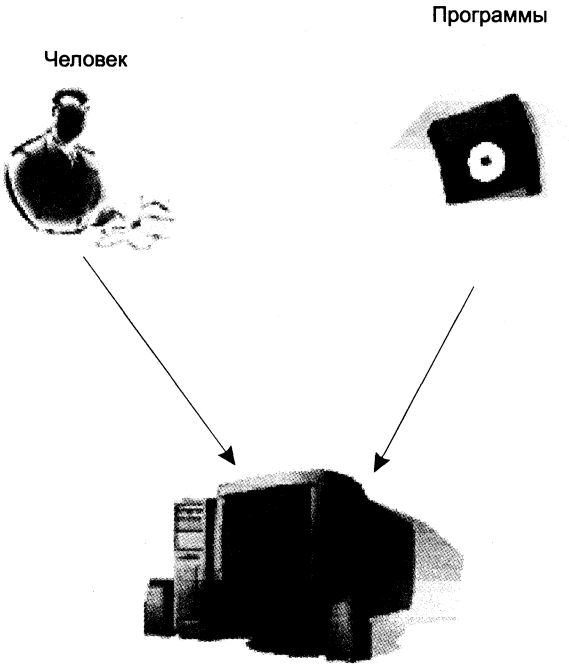


Рис. 1.1. Взаимодействие пользователя и его программ с компьютером через операционную систему

[1] Если операционная система не поддерживает механизм учетных записей, как это имеет место, например, в семействе операционных систем Windows 9x компании Microsoft, то пароль сверяется по специальному файлу, где он хранится в зашифрованном виде.

[2] В 70-е годы пользователи за терминалом писали log in, и это означало процедуру регистрации. Были системы, в которых требовалось набрать команду log on, что означало то же самое.

Понятие операционной среды

Итак, операционная система выполняет функции управления вычислениями в компьютере, распределяет ресурсы вычислительной системы между различными вычислительными процессами и образует ту программную среду, в которой выполняются прикладные программы пользователей. Такая среда называется *операционной*. Последнее следует понимать в том плане, что при запуске программы она будет обращаться к операционной системе с соответствующими запросами на выполнение определенных действий, или функций. Эти функции операционная система выполняет, запуская специальные системные программные модули, входящие в ее состав.

Итак, при создании двоичных машинных программ прикладные программисты могут вообще не знать многих деталей управления конкретными ресурсами вычислительной системы, а должны только обращаться к некоторой программной подсистеме с соответствующими вызовами и получать от нее необходимые функции и сервисы. Эта программная подсистема и есть операционная система, а набор ее функций и сервисов, а также правила обращения к ним как раз и образуют то базовое понятие, которое мы называем операционной средой. Таким образом, можно сказать, что термин «операционная среда» означает, прежде всего, соответствующие интерфейсы, необходимые программам и пользователям для обращения к управляющей (*супервизорной*) части операционной системы с целью получить определенные сервисы.

Системных функций бывает много, они определяют те возможности, которые операционная система предоставляет выполняющимся под ее управлением приложениям. Такого рода *системные запросы* (вызовы системных операций, или функций) либо явно прописываются в тексте программы программистами, либо подставляются автоматически самой системой программирования на этапе трансляции исходного текста разрабатываемой программы. Каждая операционная система имеет свое множество системных функций; они вызываются соответствующим образом, по принятым в системе правилам. Совокупность системных вызовов и правил, по которым их следует использовать, как раз и определяет уже упомянутый нами интерфейс прикладного программирования (API). Очевидно, что программа, созданная для работы в некоторой операционной системе, скорее всего не будет работать в другой операционной системе, поскольку API у этих операционных систем, как правило, различаются. Стараясь преодолеть это ограничение, разработчики операционных систем стали создавать так называемые *программные среды*. Программную (системную) среду следует понимать как некоторое системное программное окружение, позволяющее выполнить все системные запросы от прикладной программы. Та системная программная среда, которая непосредственно образуется кодом операционной системы, называется *основной*, *естественной*, или нативной (native). Помимо основной операционной среды в операционной системе могут быть организованы (путем эмуляции иной операционной среды) дополнительные программные среды. Если в операционной системе организована работа с различными операционными средами, то в такой системе можно выполнять программы, созданные не только для данной, но и для других операционных систем.

Можно сказать, что программы создаются для работы в некоторой заданной операционной среде. Например, можно создать программу для работы в среде DOS. Если такая программа все функции, связанные с операциями ввода-вывода и с запросами памяти, выполняет не сама, а за счет обращения к системным функциям DOS, то она будет (в абсолютном большинстве случаев) успешно выполняться и в MS DOS, и в PC DOS, и в Windows 9x, и в Windows 2000, и в OS/2, и даже в Linux.

Итак, параллельное существование терминов «операционная система» и «операционная среда» вызвано тем, что операционная система (в общем случае) может поддерживать несколько операционных сред. Почти все современные 32-разрядные операционные системы, созданные для персональных компьютеров, поддерживают по нескольку операционных сред. Так, операционная система OS/2 Warp, которая в свое время была одной из лучших в этом отношении, может выполнять следующие программы:

- основные программы, созданные с учетом соответствующего «родного» 32-разряднго программного интерфейса этой операционной системы;
- 16-разрядные программы, созданные для систем OS/2 первого поколения;
- 16-разрядные приложения, разработанные для выполнения в операционной среде MS DOS или PC DOS;
- 16-разрядные приложения, созданные для операционной среды Windows 3.x;
- саму операционную оболочку Windows 3.x и уже в ней — созданные для нее программы.

А операционная система Windows XP позволяет выполнять помимо основных приложений, созданных с использованием Win32API, 16-разрядные приложения для Windows 3.x, 16-разрядные DOS-приложения, 16-разрядные приложения для первой версии OS/2.

Операционная среда может включать несколько интерфейсов: пользовательские и программные. Если говорить о пользовательских, то, например, система Linux имеет для пользователя как интерфейсы командной строки (можно использовать различные «оболочки» — shell), наподобие Norton Commander, например Midnight Commander, так и графические интерфейсы, например X-Window с различными менеджерами окон — KDE, Gnome и др. Если же говорить о программных интерфейсах, то в тех же операционных системах с общим названием Linux программы могут обращаться как к операционной системе за соответствующими сервисами и функциями, так и к графической подсистеме (если она используется). С точки зрения архитектуры процессора (и персонального компьютера в целом) двоичная программа, созданная для работы в среде Linux, использует те же команды и форматы данных, что и программа, созданная для работы в среде Windows NT. Однако в первом случае мы имеем обращение к одной операционной среде, а во втором — к другой. И программа, созданная непосредственно для Windows, не будет выполняться в Linux; однако если в операционной системе Linux организовать полноценную операционную среду Windows, то наша Windows-программа может быть выполнена. Завершая этот раздел, можно еще раз сказать, что операционная среда — это то системное программное окружение, в котором могут выполняться программы, созданные по правилам работы этой среды.

Прерывания

Прерывания представляют собой механизм, позволяющий координировать параллельное функционирование отдельных устройств вычислительной системы и реагировать на особые состояния, возникающие при работе процессора, то есть прерывание — это принудительная передача управления от выполняемой программы к системе (а через нее — к соответствующей программе обработки прерывания), происходящая при возникновении определенного события.

Идея прерывания была предложена также очень давно — в середине 50-х годов, — и можно без преувеличения сказать, что она внесла наиболее весомый вклад в развитие вычислительной техники. Основная цель введения прерываний — реализация асинхронного режима функционирования и распараллеливание работы отдельных устройств вычислительного комплекса.

Механизм прерываний реализуется аппаратно-программными средствами. Структуры систем прерывания (в зависимости от аппаратной архитектуры) могут быть самыми разными, но все они имеют одну общую особенность — *прерывание непременно влечет за собой изменение порядка выполнения команд процессором.*

Механизм обработки прерываний независимо от архитектуры вычислительной системы подразумевает выполнение некоторой последовательности шагов.

1. Установление факта прерывания (прием сигнала запроса на прерывание) и идентификация прерывания (в операционных системах идентификация прерывания иногда осуществляется повторно, на шаге 4).
2. Запоминание состояния прерванного процесса вычислений. Состояние процесса выполнения программы определяется, прежде всего, значением счетчика команд (адресом следующей команды, который, например, в i80x86 определяется регистрами CS и IP — указателем команды [1, 8, 48]), содержимым регистров процессора, и может включать также спецификацию режима (например, режим пользовательский или привилегированный) и другую информацию.
3. Управление аппаратно передается на подпрограмму обработки прерывания. В простейшем случае в счетчик команд заносится начальный адрес подпрограммы обработки прерываний, а в соответствующие регистры — информация из слова состояния. В более развитых процессорах, например в 32-разрядных микропроцессорах фирмы Intel (начиная с i80386 и включая последние процессоры Pentium IV) и им подобных, осуществляются достаточно сложная процедура определения начального адреса соответствующей подпрограммы обработки прерывания и не менее сложная процедура инициализации рабочих регистров процессора (подробно эти вопросы рассматриваются в разделе «Система прерываний 32-разрядных микропроцессоров i80x86» главы 4).
4. Сохранение информации о прерванной программе, которую не удалось спасти на шаге 2 с помощью аппаратуры. В некоторых процессорах предусматривается запоминание довольно большого объема информации о состоянии прерванных вычислений.
5. Собственно выполнение программы, связанной с обработкой прерывания. Эта работа может быть выполнена той же подпрограммой, на которую было переда-

но управление на шаге 3, но в операционных системах достаточно часто она реализуется путем последующего вызова соответствующей подпрограммы.

6. Восстановление информации, относящейся к прерванному процессу (этап, обратный шагу 4).
7. Возврат на прерванную программу.

Шаги 1–3 реализуются аппаратно, шаги 4–7 — программно.

На рис. 1.2 показано, что при возникновении запроса на прерывание естественный ход вычислений нарушается и управление передается на программу обработки возникшего прерывания. При этом средствами аппаратуры сохраняется (как правило, с помощью механизмов стековой памяти) адрес той команды, с которой следует продолжить выполнение прерванной программы. После выполнения программы обработки прерывания управление возвращается на прерванную ранее программу посредством занесения в указатель команд сохраненного адреса команды, которую нужно было бы выполнить, если бы не возникло прерывание. Однако такая схема используется только в самых простых программных средах. В мультипрограммных операционных системах обработка прерываний происходит по более сложным схемам, о чем будет более подробно написано ниже.

Итак, главные функции механизма прерываний — это:

- распознавание или классификация прерываний;
- передача управления соответствующему обработчику прерываний;
- корректное возвращение к прерванной программе.

Переход от прерываемой программы к обработчику и обратно должен выполняться как можно быстрей. Одним из самых простых и быстрых методов является использование таблицы, содержащей перечень всех допустимых для компьютера прерываний и адреса соответствующих обработчиков. Для корректного возвращения к прерванной программе перед передачей управления обработчику прерываний содержимое регистров процессора запоминается либо в памяти с прямым доступом, либо в системном стеке (system stack).

Прерывания, возникающие при работе вычислительной системы, можно разделить на два основных класса: внешние (их иногда называют асинхронными) и внутренние (синхронные).

Внешние прерывания вызываются асинхронными событиями, которые происходят вне прерываемого процесса, например:

- прерывания от таймера;
- прерывания от внешних устройств (прерывания по вводу-выводу);
- прерывания по нарушению питания;
- прерывания с пульта оператора вычислительной системы;
- прерывания от другого процессора или другой вычислительной системы.

Внутренние прерывания вызываются событиями, которые связаны с работой процессора и являются синхронными с его операциями. Примерами являются следующие запросы на прерывания:

- при нарушении адресации (в адресной части выполняемой команды указан запрещенный или несуществующий адрес, обращение к отсутствующему сегменту или странице при организации механизмов виртуальной памяти);
- при наличии в поле кода операции незадействованной двоичной комбинации;
- при делении на ноль;
- вследствие переполнения или исчезновения порядка;
- от средств контроля (например, вследствие обнаружения ошибки четности, ошибок в работе различных устройств).

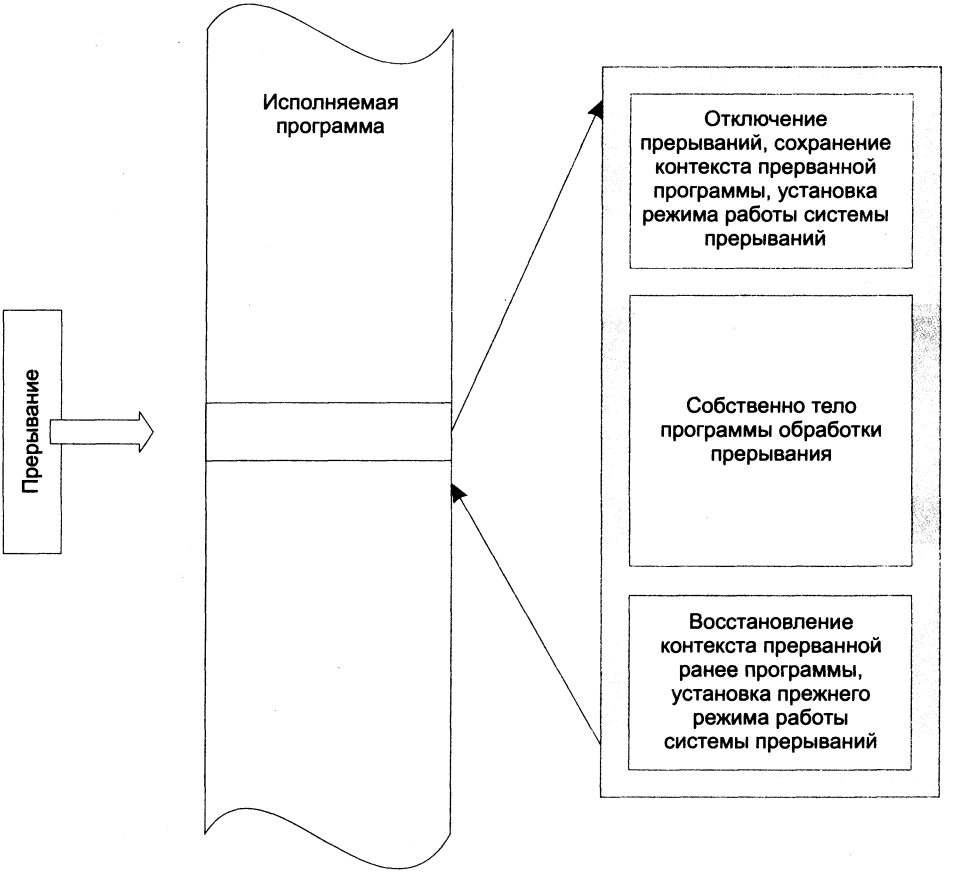


Рис. 1.2. Обработка прерывания

Могут еще существовать прерывания в связи с попыткой выполнить команду, которая сейчас запрещена. Во многих компьютерах часть команд должна выполняться только кодом самой операционной системы, но не прикладными программами. Это делается с целью повышения защищенности выполняемых на компьютере вычислений. Соответственно в аппаратуре предусмотрены различные режимы работы, и пользовательские программы выполняются в режиме, в котором некоторое под-

множество команд, называемых привилегированными, не исполняется. К привилегированным командам помимо команд ввода-вывода относятся и команды переключения режима работа центрального процессора, и команды инициализации некоторых системных регистров процессора. При попытке использовать команду, запрещенную в данном режиме, происходит внутреннее прерывание, и управление передается самой операционной системе.

Наконец, существуют собственно *программные прерывания*. Эти прерывания происходят по соответствующей команде прерывания, то есть по этой команде процессор осуществляет практически те же действия, что и при обычных внутренних прерываниях. Этот механизм был специально введен для того, чтобы переключение на системные программные модули происходило не просто как переход на подпрограмму, а точно таким же образом, как и обычное прерывание. Этим, прежде всего, обеспечивается автоматическое переключение процессора в привилегированный режим с возможностью исполнения любых команд.

Сигналы, вызывающие прерывания, формируются вне процессора или в самом процессоре, они могут возникать одновременно. Выбор одного из них для обработки осуществляется на основе приоритетов, приписанных каждому типу прерывания. Так, со всей очевидностью, прерывания от схем контроля процессора должны обладать наивысшим приоритетом (действительно, если аппаратура работает неправильно, то не имеет смысла продолжать обработку информации). На рис. 1.3 изображен обычный порядок (приоритеты) обработки прерываний в зависимости от типа прерываний. Учет приоритета может быть встроен в технические средства, а также определяться операционной системой, то есть кроме аппаратно реализованных приоритетов прерывания большинство вычислительных машин и комплексов допускают программно-аппаратное управление порядком обработки сигналов прерывания. Второй способ, дополняя первый, позволяет применять различные *дисциплины обслуживания прерываний.*

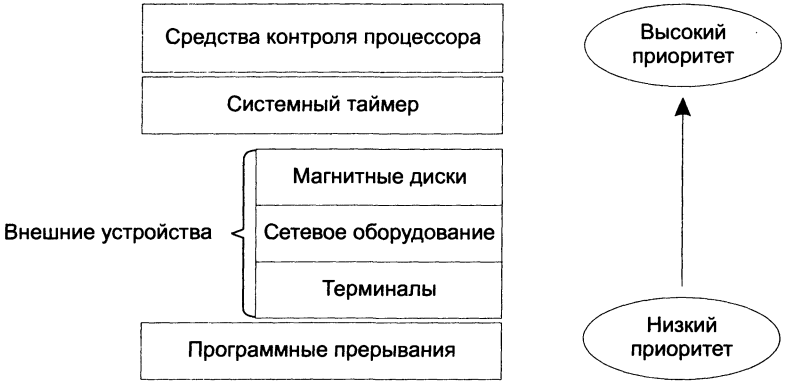


Рис. 1.3. Распределение прерываний по уровням приоритета

Наличие сигнала прерывания не обязательно должно вызывать прерывание исполняющейся программы. Процессор может обладать средствами защиты от прерываний: *отключение* системы прерываний, *маскирование* (запрет) отдельных сиг-

налов прерывания. Программное управление этими средствами (существуют специальные команды для управления работой системы прерываний) позволяет операционной системе регулировать обработку сигналов прерывания, заставляя процессор обрабатывать их сразу по приходу; откладывать обработку на некоторое время; полностью игнорировать прерывания. Обычно операция прерывания выполняется только после завершения выполнения текущей команды. Поскольку сигналы прерывания возникают в произвольные моменты времени, то на момент прерывания может существовать несколько сигналов прерывания, которые могут быть обработаны только последовательно. Чтобы обработать сигналы прерывания в разумном порядке, им (как уже отмечалось) присваиваются приоритеты. Сигнал с более высоким приоритетом обрабатывается в первую очередь, обработка остальных сигналов прерывания откладывается.

Программное управление специальными регистрами маски (маскирование сигналов прерывания) позволяет реализовать различные дисциплины обслуживания.

- *С относительными приоритетами*, то есть обслуживание не прерывается даже при наличии запросов с более высокими приоритетами. После окончания обслуживания данного запроса обслуживается запрос с наивысшим приоритетом. Для организации такой дисциплины необходимо в программе обслуживания данного запроса наложить маски на все остальные сигналы прерывания или просто отключить систему прерываний.
- *С абсолютными приоритетами*, то есть всегда обслуживается прерывание с наивысшим приоритетом. Для реализации этого режима необходимо на время обработки прерывания замаскировать все запросы с более низким приоритетом. При этом возможно многоуровневое прерывание, то есть прерывание программ обработки прерываний. Число уровней прерывания в этом режиме изменяется и зависит от приоритета запроса.
- По *принципу стека*, или, как иногда говорят, *по дисциплине LCFS* (Last Come First Served — последним пришел, первым обслужен), то есть запросы с более низким приоритетом могут прерывать обработку прерывания с более высоким приоритетом. Дли этого необходимо не накладывать маску ни на один из сигналов прерывания и не выключать систему прерываний.

Следует особо отметить, что для правильной реализации последних двух дисциплин нужно обеспечить полное маскирование системы прерываний при выполнении шагов 1–4 и 6–7. Это необходимо для того, чтобы не потерять запрос и правильно его обслужить. Многоуровневое прерывание должно происходить на этапе собственно обработки прерывания, а не на этапе перехода с одного процесса вычислений на другой.

Управление ходом выполнения задач со стороны операционной системы заключается в организации реакций на прерывания, в организации обмена информацией (данными и программами), в предоставлении необходимых ресурсов, в динамике выполнения задачи и в организации сервиса. Причины прерываний определяет операционная система (модуль, который называют *супервизором прерываний*), она же и выполняет действия, необходимые при данном прерывании и в данной ситуации. Поэтому в состав любой операционной системы реального времени прежде

всего входят программы управления системой прерываний, контроля состояний задач и событий, синхронизации задач, средства распределения памяти и управления ею, а уже потом средства организации данных (с помощью файловых систем и т. д. Следует однако заметить, что современная операционная система реального времени должна вносить в аппаратно-программный комплекс нечто большее, нежели просто обеспечение быстрой реакции на прерывания.

Как мы уже знаем, при появлении запроса на прерывание система прерываний идентифицирует сигнал и, если прерывания разрешены, то управление передается на соответствующую подпрограмму обработки. Из рис. 1.2 видно, что в подпрограмме обработки прерывания имеется две служебные секции. Это — первая секция, в которой осуществляется сохранение контекста прерываемых вычислений, который не смог быть сохранен на шаге 2, и последняя, заключительная секция, в которой, наоборот, осуществляется восстановление контекста. Для того чтобы система прерываний не среагировала повторно на сигнал запроса на прерывание, она обычно автоматически «закрывает» (отключает) прерывания, поэтому необходимо потом в подпрограмме обработки прерываний вновь включать систему прерываний. В соответствии с рассмотренными режимами обработки прерываний (с относительными и абсолютными приоритетами и по правилу LCFS) установка этих режимов осуществляется в конце первой секции подпрограммы обработки. Таким образом, на время выполнения центральной секции (в случае работы в режимах с абсолютными приоритетами и по дисциплине LCFS) прерывания разрешены. На время работы заключительной секции подпрограммы обработки система прерываний вновь должна быть отключена и после восстановления контекста опять включена. Поскольку эти действия необходимо выполнять практически в каждой подпрограмме обработки прерываний, во многих операционных системах первые секции подпрограмм обработки прерываний выделяются в уже упоминавшийся специальный системный программный модуль, называемый супервизором прерываний.

Супервизор прерываний прежде всего сохраняет в дескрипторе текущей задачи рабочие регистры процессора, определяющие контекст прерываемого вычислительного процесса. Далее он определяет ту подпрограмму, которая должна выполнить действия, связанные с обслуживанием настоящего (текущего) запроса на прерывание. Наконец, перед тем, как передать управление на эту подпрограмму, супервизор прерываний устанавливает необходимый режим обработки прерывания. После выполнения подпрограммы обработки прерывания управление вновь передается ядру операционной системы. На этот раз уже на тот модуль, который занимается диспетчеризацией задач (см. раздел «Планирование и диспетчеризация процессов и задач» в главе 2). И уже диспетчер задач, в свою очередь, в соответствии с принятой дисциплиной распределения процессорного времени (между выполняющимися вычислительными процессами) восстановит контекст той задачи, которой будет решено выделить процессор. Рассмотренную нами схему иллюстрирует рис. 1.4.

Как мы видим из рисунка, здесь отсутствует возврат в прерванную ранее программу непосредственно из самой подпрограммы обработки прерывания. Для прямого

возврата достаточно адрес возврата сохранить в стеке, что и делает аппаратура процессора. При этом стек легко обеспечивает возможность возврата в случае вложенных прерываний, поскольку он всегда реализует дисциплину LCFS.

Рис. 1.4. Обработка прерывания при участии супервизоров ОС

Однако если бы контекст вычислительных процессов сохранялся просто в стеке, как это обычно реализуется аппаратурой, а не в специальных структурах данных, называемых дескрипторами, о чем будет подробно изложено чуть позже, то у нас не было бы возможности гибко подходить к выбору той задачи, которой нужно передать процессор после завершения работы подпрограммы обработки прерывания. Естественно, что это только общий принцип. В конкретных процессорах и в конкретных операционных системах могут существовать некоторые отступления от рассмотренной схемы и/или дополнения. Например, в современных процессорах часто имеются специальные аппаратные возможности для сохранения контекста прерываемого вычислительного процесса непосредственно в его дескрипторе, то есть дескриптор процесса (по крайней мере его часть) становится структурой данных, которую поддерживает аппаратура.

Для полного понимания принципов создания и механизмов реализации рассматриваемых далее современных операционных систем необходимо знать архитектуру и, в частности, особенности системы прерывания персональных компьютеров. Этот вопрос более подробно рассмотрен в главе 4, посвященной архитектуре микропроцессоров i80x86.

Понятия вычислительного процесса и ресурса

Понятие *последовательного*[1] *вычислительного процесса*, или просто *процесса*, является одним из основных при рассмотрении операционных систем. Как понятие *процесс* является определенным видом абстракции, и мы будем придерживаться следующего неформального определения, приведенного в [47]. Последовательный процесс, иногда называемый *задачей*[2] (task), — это отдельная программа с ее данными, выполняющаяся на последовательном процессоре. Напомним, что под последовательным мы понимаем такой процессор, в котором текущая команда выполняется после завершения предыдущей. В современных процессорах мы сталкиваемся с ситуациями, когда возможно параллельное выполнение нескольких команд. Это делается для повышения скорости вычислений. В этих процессорах параллелизм достигается двумя основными способами — организацией конвейерного механизма выполнения команды и созданием нескольких конвейеров. Однако в подобных процессорах аппаратными решениями обязательно достигается логическая последовательность в выполнении команд, предусмотренная программой. Необходимость этого объясняется в главе 7, посвященной организации параллельных вычислительных процессов.

Концепция процесса предполагает два аспекта: во-первых, он является носителем данных и, во-вторых, он собственно и выполняет операции, связанные с обработкой этих данных.

В качестве примеров процессов (задач) можно назвать прикладные программы пользователей, утилиты и другие системные обрабатывающие программы. Процессом может быть редактирование какого-либо текста, трансляция исходной программы, ее компоновка, исполнение. Причем трансляция какой-нибудь исходной программы является одним процессом, а трансляция следующей исходной программы — другим процессом, поскольку транслятор как объединение программных модулей здесь выступает как одна и та же программа, но данные, которые он обрабатывает, являются разными.

---

[1] Слово «последовательный» в большинстве случаев опускается. Считается, что речь идет о вычислениях, осуществляемых на обычных «последовательных» процессорах, которые выполняют команду за командой, а не параллельно несколько команд за один такт.

[2] В концепции, которая получила наибольшее распространение в 70-е годы, *задача* — это совокупность связанных между собой и образующих единое целое программных модулей и данных, требующая ресурсов вычислительной системы для своей реализации. В последующие годы задачей стали называть единицу работы, для выполнения которой предоставляется центральный процессор. Вычислительный процесс может включать в себя несколько задач.

Концепция процесса преследует цель выработать механизмы распределения и управления ресурсами. Понятие ресурса, так же как и понятие процесса, является, пожалуй, основным при рассмотрении операционных систем. Термин *ресурс* обычно применяется по отношению к многократно используемым, относительно стабильным и часто недостающим объектам, которые запрашиваются, задействуются и освобождаются в период их активности. Другими словами, ресурсом называется всякий объект, который может распределяться внутри системы.

Ресурсы могут быть *разделяемыми*, когда несколько процессов используют их *одновременно* (в один и тот же момент времени) или *параллельно* (попеременно в течение некоторого интервала времени), а могут быть и *неделимыми* (рис. 1.5).

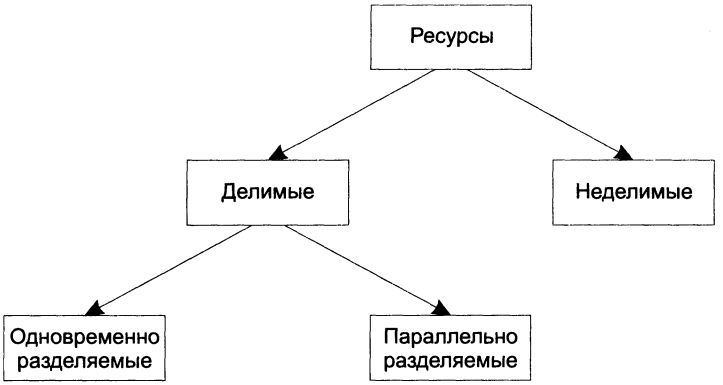


Рис. 1.5. Классификация ресурсов

При разработке первых систем ресурсами считались процессорное время, память, каналы ввода-вывода и периферийные устройства [22, 53]. Однако очень скоро понятие ресурса стало гораздо более универсальным и общим. Различного рода программные и информационные ресурсы также могут быть определены для системы как объекты, которые могут разделяться и распределяться и доступ к которым необходимо соответствующим образом контролировать. В настоящее время понятие ресурса превратилось в абстрактную структуру с целым рядом атрибутов, характеризующих способы доступа к этой структуре и ее физическое представление в системе. Более того, помимо системных ресурсов, о которых мы сейчас говорили, ресурсами стали называть и такие объекты, как сообщения и синхросигналы, которыми обмениваются задачи.

В первых вычислительных системах любая программа могла выполняться только после полного завершения предыдущей. Поскольку эти первые вычислительные системы были построены в соответствии с принципами, изложенными в известной работе Яноша Джона фон Неймана, все подсистемы и устройства компьютера управлялись исключительно центральным процессором. Центральный процессор осуществлял и выполнение вычислений, и управление операциями ввода-вывода данных. Соответственно, пока осуществлялся обмен данными между оперативной памятью и внешними устройствами, процессор не мог выполнять вычисления.

Введение в состав вычислительной машины специальных контроллеров позволило совместить во времени (распараллелить) операции вывода полученных данных и последующие вычисления на центральном процессоре. Однако все равно процессор продолжал часто и долго простаивать, дожидаясь завершения очередной операции ввода-вывода. Поэтому было предложено организовать так называемый *мультипрограммный*, или *мультизадачный*, режим работы вычислительной системы.

Мультипрограммирование, многопользовательский режим работы и режим разделения времени

Вкратце суть мультипрограммного режима работы заключается в том, что пока одна программа (один вычислительный процесс, как мы теперь говорим) ожидает завершения очередной операции ввода-вывода, другая программа (а точнее, другая задача) может быть поставлена на решение (рис. 1.6). Это позволяет более полно использовать имеющиеся ресурсы (например, центральный процессор начинает меньше простаивать, как это видно из рисунка) и уменьшить общее (суммарное) время, необходимое для решения некоторого множества задач.

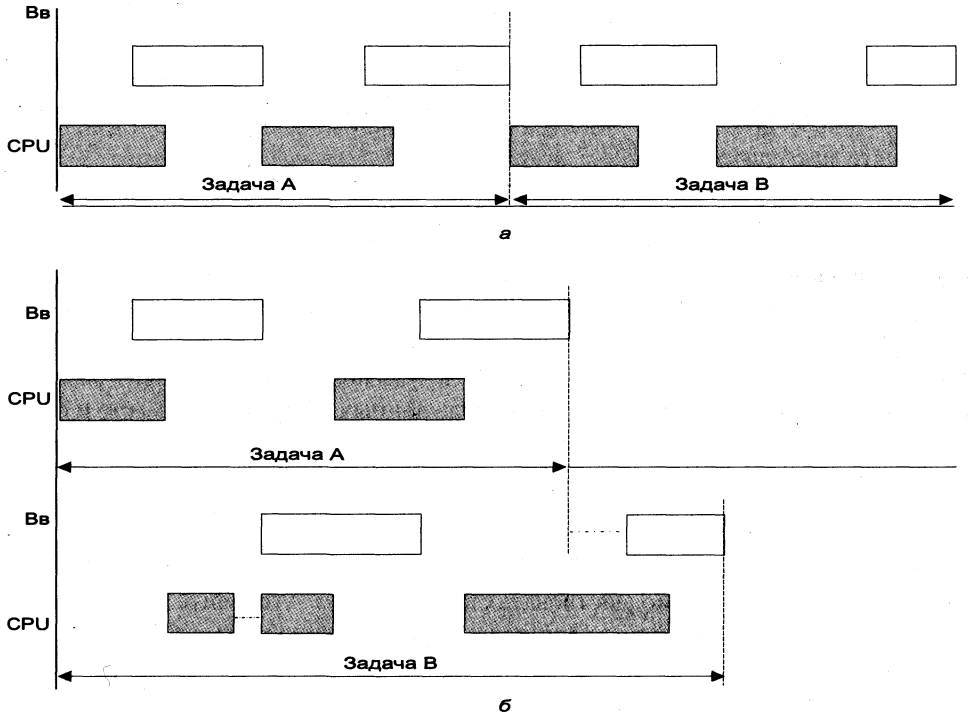


Рис. 1.6. Пример выполнения двух программ в мультипрограммном режиме

На рисунке в качестве примера изображена такая гипотетическая ситуация, при которой благодаря совмещению во времени двух вычислительных процессов об-

щее время их выполнения получается меньше, чем если бы их выполняли по очереди (запуск одного начинался бы только после полного завершения другого). Из этого же рисунка видно, что время выполнения каждого процесса в общем случае больше, чем если бы мы выполняли каждый из них как единственный.

При мультипрограммировании повышается пропускная способность системы, но отдельный процесс никогда не может быть выполнен быстрее, чем если бы он выполнялся в однопрограммном режиме (всякое разделение ресурсов замедляет работу одного из участников за счет дополнительных затрат времени на ожидание освобождения ресурса).

Мультипрограммирование стало применяться все чаще и шире в 60-х годах XX века, когда крупные компании получили, наконец, возможность приобретать в собственность вычислительную технику и использовать ее для решения своих задач. До этого времени вычислительная техника была доступна, прежде всего, для военных целей и решения отдельных задач общегосударственного масштаба. А поскольку стоимость компьютеров в то время была чрезвычайно большой, то компании, вложив свои капиталы в вычислительную технику, захотели за счет продажи машинного времени не только покрыть те расходы, которые сопутствовали ее приобретению и использованию, но и зарабатывать дополнительные деньги, то есть получать прибыль. Машинное время стали активно продавать, сдавая в аренду имеющиеся компьютеры, и потенциальная возможность решать в единицу времени большее количество задач, возможно от разных клиентов, стала выступать основным стимулом в развитии способов организации вычислений и операционных систем.

Задачи пользователей ставились в очередь на решение, и распределение времени центрального процессора и других ресурсов компьютера между несколькими выполняющимися вычислительными процессами позволяло организовать параллельное выполнение сразу нескольких задач. Эти задачи могли относиться и к одному пользователю, и к нескольким. Однако ставил их на решение оператор вычислительной системы.

Приблизительно в то же время, может быть чуть позже, стали активно развиваться всевозможные устройства ввода и вывода данных. Не стояло на месте и системное программное обеспечение. Появилась возможность пользователю самому вводить исходные данные и тут же получать результаты вычислений, причем в удобном для него виде. Упрощение пользовательского интерфейса и развитие интерфейсных функций операционных систем позволило реализовать диалоговый режим работы. Как известно, диалоговый режим предполагает, что пользователь может сам, без посредника, взаимодействовать с компьютером — готовить и запускать свои программы, вводить исходные данные, получать результаты, приостанавливать вычисления и вновь их возобновлять и т. д.

Очевидно, что диалоговый режим работы может быть реализован и без мультипрограммирования. Наглядное тому доказательство — многочисленные дисковые операционные системы, начиная от CP-M и кончая PC-DOS 7.0, которые долгие годы устанавливались на персональные компьютеры и обеспечивали только однопрограммный режим. Однако эти однопрограммные диалоговые системы появи-

лись гораздо позже мультипрограммных. Как это ни кажется странным, им предшествовали многочисленные и разнообразные операционные системы, позволяющие одновременно работать с компьютером большому количеству пользователей и параллельно решать множество задач. Основная причина тому — стоимость компьютера. Только с удешевлением компьютеров появилась возможность иметь свой персональный компьютер, и первое время считалось, что однопрограммного режима работы вполне достаточно. Главным для персональных компьютеров до сих пор считается удобство работы, причем именно в диалоговом режиме, простота интерфейса и его интуитивная понятность.

Совмещение диалогового режима работы с компьютером и режима мультипрограммирования привело к появлению *мультитерминальных*, или многопользовательских, систем. Организовать параллельное выполнение нескольких задач можно разными способами (более подробно об этом см. в главе 2). Если это осуществляется таким образом, что на каждую задачу поочередно выделяется некий квант времени, после чего процессор передается другой задаче, готовой к продолжению вычислений, то такой режим принято называть режимом *разделения времени* (time sharing). Системы разделения времени активно развивались в 60–70 годы, и сам термин означал именно мультитерминальную и мультипрограммную систему.

Итак, операционная система может поддерживать *мультипрограммирование* (многопроцессность). В этом случае она должна стараться эффективно использовать имеющиеся ресурсы путем организации к ним очередей запросов, составляемых тем или иным способом. Это требование достигается поддерживанием в памяти более одного вычислительного процесса, ожидающего процессор, и более одного процесса, готового использовать другие ресурсы, как только последние станут доступными.

Общая схема выделения ресурсов такова. При необходимости использовать какой-либо ресурс (оперативную память, устройство ввода-вывода, массив данных и т. п.) вычислительный процесс (задача) путем обращения к *супервизору*[1] (supervisor) операционной системы посредством специальных вызовов (команд, директив) сообщает о своем требовании. При этом указывается вид ресурса и, если надо, его объем. Например, при запросе оперативной памяти указывается количество адресуемых ячеек, необходимое для дальнейшей работы.

Команда обращения к операционной системе передает ей управление, переводя процессор в привилегированный режим работы (см. раздел «Прерывания»), если такой существует. Большинство компьютеров имеют два (и более) режима работы: *привилегированный* (режим супервизора) и *пользовательский*. Кроме того, могут быть режимы для эмуляции какой-нибудь другой ЭВМ или для организации виртуальной машины, защищенной от остальных вычислений, осуществляемых на этом же компьютере, и т. д. Мы уже говорили об этом, затрагивая вопрос организации прерываний.

[1] Супервизор — центральный (главный) управляющий модуль операционной системы. Может состоять из нескольких модулей, например супервизора ввода-вывода, супервизора прерываний, супервизора программ, диспетчера задач и т. д. В последние годы термин «супервизор» применяется все реже и реже.

Ресурс может быть выделен вычислительному процессу (задаче), обратившемуся к операционной системе с соответствующим запросом, если:

- ресурс свободен и в системе нет запросов от задач более высокого приоритета к этому же ресурсу;
- текущий запрос и ранее выданные запросы допускают совместное использование ресурсов;
- ресурс используется задачей низшего приоритета и может быть временно отобран (разделяемый ресурс).

Получив запрос, операционная система либо удовлетворяет его и возвращает управление задаче, выдавшей данный запрос, либо, если ресурс занят, ставит задачу в очередь к ресурсу, переводя ее в состояние ожидания (блокируя). Очередь к ресурсу может быть организована несколькими способами, но чаще всего она реализуется с помощью списковой структуры.

После окончания работы с ресурсом задача опять с помощью специального вызова супервизора (посредством соответствующей команды) сообщает операционной системе об отказе от ресурса, либо операционная система забирает ресурс сама, если управление возвращается супервизору после выполнения какой-либо системной функции. Супервизор операционной системы, получив управление по этому обращению, освобождает ресурс и проверяет, имеется ли очередь к освободившемуся ресурсу. Если очередь есть, то в зависимости от принятой *дисциплины обслуживания*[1] и приоритетов заявок он выводит из состояния ожидания задачу, ждущую ресурс, и переводит ее в состояние готовности к выполнению, после чего либо передает управление ей, либо возвращает управление задаче, только что освободившей ресурс.

При выдаче запроса на ресурс задача может указать, хочет ли она владеть ресурсом монопольно или допускает совместное использование с другими задачами. Например, с файлом можно работать монопольно, а можно и совместно с другими задачами.

Если в системе имеется некоторая совокупность ресурсов, то управлять их использованием можно на основе некоторой стратегии. Стратегия подразумевает четкую формулировку целей, следуя которым можно добиться эффективного распределения ресурсов.

При организации управления ресурсами всегда требуется принять решение о том, что в данной ситуации выгоднее: быстро обслуживать отдельные наиболее важные запросы, предоставлять всем процессам равные возможности или обслуживать максимально возможное количество процессов и наиболее полно использовать ресурсы [46].

Диаграмма состояний процесса

Необходимо отличать системные управляющие вычислительные процессы, представляющие работу супервизора операционной системы и занимающиеся распреде-

[1] Термин «дисциплина обслуживания» следует понимать как некое правило обслуживания, в том числе и учет каких-либо приоритетов при обслуживании. Например, дисциплина «последний пришедший обслуживается первым» определяет обслуживание в порядке, обратном очередности поступления соответствующих запросов.

лением и управлением ресурсов, от всех других процессов: задач пользователей и системных обрабатывающих процессов. Последние, хоть и относятся к операционной системе, но не входят в ядро операционной системы и требуют общих ресурсов для своей работы, которые получают от супервизора. Для системных управляющих процессов, в отличие от обрабатывающих, в большинстве операционных систем ресурсы распределяются изначально и однозначно. Эти вычислительные процессы сами управляют ресурсами системы, в борьбе за которые конкурируют все остальные процессы. Поэтому исполнение системных управляющих программ не принято называть процессами, и термин «задача» следует употреблять только по отношению к процессам пользователей и к системным обрабатывающим процессам. Однако это справедливо не для всех операционных систем. Например, в так называемых «микроядерных» операционных системах (см. главу 9) большинство управляющих программных модулей самой операционной системы и даже драйверы имеют статус высокоприоритетных процессов, для выполнения которых необходимо выделить соответствующие ресурсы. В качестве примера можно привести хорошо известную операционную систему реального времени QNX фирмы Quantum Software Systems. Аналогично и в UNIX-системах, которые хоть и не относятся к микроядерным, выполнение системных программных модулей тоже имеет статус системных процессов, получающих ресурсы для своего исполнения.

Очевидно, что если некий вычислительный процесс (назовем его первым) в данный конкретный момент времени не исполняется, поскольку процессор занят исполнением какого-то другого процесса, то операционная система должна знать, что вычисления в первом процессе приостановлены. Информация об этом заносится в специальную информационную структуру, сопровождающую каждый вычислительный процесс. Таких приостановленных процессов может быть несколько. Они могут образовывать очередь задач, которые возобновят свои вычисления, как только им будет предоставлен процессор. Некоторые процессы, при своем выполнении требующие ввода или вывода данных, на время выполнения этих запросов могут освобождать процессор. Такие события тоже должны для операционной системы помечаться соответствующим образом. Говорят, что процесс может находиться в одном из нескольких состояний. Информация о состоянии процесса содержится в упомянутой выше информационной структуре, доступной супервизору.

Если обобщать и рассматривать не только традиционные системы общего назначения и привычные всем нам современные мультизадачные операционные системы для персональных компьютеров, но и операционные системы реального времени, то можно сказать, что процесс может находиться в активном и пассивном (не активном) состоянии. В *активном состоянии* процесс может конкурировать за ресурсы вычислительной системы, а в *пассивном состоянии* он известен системе, но за ресурсы не конкурирует (хотя его существование в системе и сопряжено с предоставлением ему оперативной и/или внешней памяти). В свою очередь, *активный процесс* может быть в одном из следующих состояний:

- *выполнения* — все затребованные процессом ресурсы выделены (в этом состоянии в каждый момент времени может находиться только один процесс, если речь идет об однопроцессорной вычислительной системе);

- *готовности к выполнению* — ресурсы могут быть предоставлены, тогда процесс перейдет в состояние выполнения;
- *блокирования*, или *ожидания*, — затребованные ресурсы не могут быть предоставлены, или не завершена операция ввода-вывода.

В большинстве операционных систем последнее состояние, в свою очередь, подразделяется на множество состояний ожидания, соответствующих определенному виду ресурса, из-за отсутствия которого процесс переходит в заблокированное состояние.

В обычных операционных системах, как правило, процесс появляется при запуске какой-нибудь программы. Операционная система организует (порождает, или выделяет) для нового процесса уже упомянутую выше информационную структуру — так называемый *дескриптор процесса*, и процесс (задача) начинает выполняться. Поэтому пассивного состояния в большинстве систем не существует. В операционных системах реального времени (ОСРВ) ситуация иная. Обычно при проектировании системы реального времени состав выполняемых ею программ (задач) известен заранее. Известны и многие их параметры, которые необходимо учитывать при распределении ресурсов (например, объем памяти, приоритет, средняя длительность выполнения, открываемые файлы, используемые устройства и проч.). Поэтому для них заранее заводят дескрипторы задач с тем, чтобы впоследствии не тратить драгоценное время на организацию дескриптора и поиски для него необходимых ресурсов. Таким образом, в ОСРВ многие процессы (задачи) могут находиться в состоянии бездействия, что мы и отобразили на рис. 1.7, отделив это состояние от остальных состояний пунктиром.

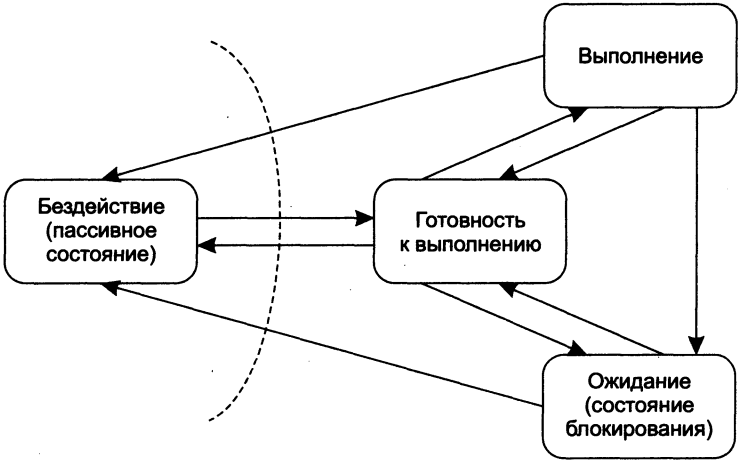


Рис. 1.7. Граф состояний процесса

За время своего существования процесс может неоднократно совершать переходы из одного состояния в другое, обусловленные обращениями к операционной системе с запросами ресурсов и выполнения системных функций, которые предоставляет операционная система, взаимодействием с другими процессами, появле-

нием сигналов прерывания от таймера, каналов и устройств ввода-вывода, других устройств. Возможные переходы процесса из одного состояния в другое отображены на рисунке в виде графа состояний. Рассмотрим эти переходы из одного состояния в другое более подробно.

Процесс из состояния бездействия может перейти в состояние готовности в следующих случаях.

- По команде оператора (пользователя). Имеет место в тех диалоговых операционных системах, где программа может иметь статус задачи, даже оставаясь пассивной, а не просто быть исполняемым файлом и получать статус задачи только на время исполнения (как это имеет место в большинстве современных операционных систем, в том числе и для персональных компьютеров).
- При выборе из очереди планировщиком (характерно для операционных систем, работающих в пакетном режиме).
- При вызове из другой задачи (посредством обращения к супервизору один процесс может создать, инициировать, приостановить, остановить, уничтожить другой процесс).
- По прерыванию от внешнего *инициативного устройства*[1] (сигнал о свершении некоторого события может запускать соответствующую задачу).
- При наступлении запланированного времени запуска программы.

Последние два способа запуска задачи, при которых процесс из состояния бездействия переходит в состояние готовности, наиболее характерны для операционных систем реального времени.

Процесс, который может исполняться, как только ему будет предоставлен процессор (а для диск-резидентных задач в некоторых системах и оперативная память), находится в состоянии готовности. Считается, что такому процессу уже выделены все необходимые ресурсы за исключением процессора.

Из состояния выполнения процесс может выйти по одной из следующих причин.

- Процесс завершается, при этом он посредством обращения к супервизору передает управление операционной системе и сообщает о своем завершении. В результате этих действий супервизор либо переводит его в список бездействующих процессов (процесс переходит в пассивное состояние), либо уничтожает. Уничтожается, естественно, не сама программа, а именно задача, которая соответствовала исполнению этой программы. В состояние бездействия процесс может быть переведен принудительно: по команде оператора или путем обращения к супервизору операционной системы из другой задачи с требованием остановить данный процесс. Само собой, что действие по команде оператора реализуется системным процессом, который «транслирует» эту команду в запрос к супервизору с требованием перевести указанный процесс в состояние бездействия.

[1] Устройство называется инициативным, если по сигналу запроса на прерывание от него должна запускаться некоторая задача.

- Процесс переводится супервизором операционной системы в состояние готовности к исполнению в связи с появлением более приоритетной задачи или в связи с окончанием выделенного ему кванта времени.
- Процесс *блокируется* (переводится в состояние ожидания) либо вследствие запроса операции ввода-вывода (которая должна быть выполнена прежде, чем он сможет продолжить исполнение), либо в силу невозможности предоставить ему ресурс, запрошенный в настоящий момент (причиной перевода в состояние ожидания может быть отсутствие сегмента или страницы в случае организации механизмов виртуальной памяти — см. раздел «Сегментная, страничная и сегментно-страничная организация памяти» в главе 3), либо по команде оператора на приостанов задачи, либо по требованию через супервизор от другой задачи.

При наступлении соответствующего события (завершилась операция ввода-вывода, освободился затребованный ресурс, в оперативную память загружена необходимая страница виртуальной памяти и т. д.) процесс деблокируется и переводится в состояние готовности к исполнению.

Таким образом, движущей силой, меняющей состояния процессов, являются события. Одним из основных видов событий являются уже рассмотренные нами прерывания.

Реализация понятия последовательного процесса в операционных системах

Для того чтобы операционная система могла управлять процессами, она должна располагать всей необходимой для этого информацией. С этой целью на каждый процесс заводится специальная информационная структура, называемая *дескриптором процесса* (описателем задачи, блоком управления задачей). В общем случае дескриптор процесса, как правило, содержит следующую информацию:

- *идентификатор процесса* (Process Identifier, PID);
- *тип* (или класс) *процесса*, который определяет для супервизора некоторые правила предоставления ресурсов;
- *приоритет процесса*, в соответствии с которым супервизор предоставляет ресурсы (в рамках одного класса процессов в первую очередь обслуживаются более приоритетные процессы);
- *переменную состояния*, которая определяет, в каком состоянии находится процесс (готов к работе, выполняется, ожидает устройства ввода-вывода и т. д.);
- *контекст задачи*, то есть защищенную область памяти (или адрес такой области), в которой хранятся текущие значения регистров процессора, когда процесс прерывается, не закончив работы;
- информацию о ресурсах, которыми процесс владеет и/или имеет право пользоваться (указатели на открытые файлы, информация о незавершенных операциях ввода-вывода и др.);
- место (или его адрес) для организации общения с другими процессами;

- параметры времени запуска (момент времени, когда процесс должен активизироваться, и периодичность этой процедуры);
- в случае отсутствия системы управления файлами адрес задачи на диске в ее исходном состоянии и адрес на диске, куда она выгружается из оперативной памяти, если ее вытесняет другая задача (последнее справедливо для *диск-резидентных* задач, которые постоянно находятся во внешней памяти на системном магнитном диске и загружаются в оперативную память только на время выполнения).

Описатели задач, как правило, постоянно располагаются в оперативной памяти с целью ускорить работу супервизора, который организует их в списки (очереди) и отображает изменение состояния процесса перемещением соответствующего описателя из одного списка в другой. Для каждого состояния (за исключением состояния выполнения для однопроцессорной системы) операционная система ведет соответствующий список задач, находящихся в этом состоянии. Однако для состояния ожидания обычно имеется не один список, а столько, сколько различных видов ресурсов могут вызывать состояние ожидания. Например, состояний ожидания завершения операции ввода-вывода может быть столько, сколько устройств ввода-вывода имеется в системе.

В некоторых операционных системах количество описателей определяется жестко и заранее (на этапе генерации варианта операционной системы или в конфигурационном файле, который используется при загрузке ОС), в других по мере необходимости система может выделять участки памяти под новые описатели. Например, в уже мало кому известной системе OS/2, которая несколько лет тому назад многими специалистами считалась одной из лучших ОС для персональных компьютеров, максимально возможное количество описателей задач указывается в конфигурационном файле CONFIG.SYS. Например, строка THREADS=1024 в файле CONFIG.SYS означает, что всего в системе может параллельно существовать и выполняться до 1024 задач, включая вычислительные процессы и их потоки.

В ныне широко распространенных системах Windows NT/2000/XP количество описателей нигде в явном виде не задается. Это переменная величина, и она определяется самой операционной системой. Однако посмотреть на текущее количество таких описателей можно. Если, работая в Windows NT/2000/XP, нажать одновременно комбинацию клавиш Ctrl+Shift+Esc, появится окно Диспетчера задач. На вкладке Быстродействие этой программы мы увидим поле с названием Всего дескрипторов и соответствующее число. Тут же указывается количество дескрипторов для управления потоками (задачами) и число полноценных вычислительных процессов. Более подробно о процессах и потоках см. далее.

В операционных системах реального времени чаще всего количество процессов фиксируется, и, следовательно, целесообразно заранее определять (на этапе генерации или конфигурирования ОС) количество дескрипторов. Для использования таких операционных систем в качестве систем общего назначения (что нынче уже нехарактерно)[1] обычно количество дескрипторов бралось с некоторым запасом,

[1] В недалеком прошлом достаточно часто в качестве вычислительных систем общего назначения приобретались мини-ЭВМ и устанавливали на них ОС реального времени.

и появление новой задачи связывалось с заполнением этой информационной структуры. Поскольку дескрипторы процессов постоянно располагаются в оперативной памяти (с целью ускорить работу диспетчера), то их количество не должно быть очень большим.

Для более эффективной обработки данных в системах реального времени целесообразно иметь постоянные задачи, полностью или частично всегда существующие в системе независимо от того, поступило на них требование или нет. Каждая постоянная задача обладает некоторой собственной областью оперативной памяти (*ОЗУ-резидентная задача*, или просто *резидентная задача*) независимо от того, выполняется задача в данный момент или нет. Эта область, в частности, может использоваться для хранения данных, полученных задачей ранее. Данные могут храниться в ней и тогда, когда задача находится в состоянии ожидания или даже в состоянии бездействия.

Для аппаратной поддержки работы операционных систем с этими информационными структурами (дескрипторами задач) в процессорах могут быть реализованы соответствующие механизмы. Так, например, в микропроцессорах Intel 80x86 (см. главу 4) имеется специальный регистр TR (Task Register), указывающий местонахождение специальной информационной структуры — сегмента состояния задачи (Task State Segment, TSS), в котором при переключении с задачи на задачу автоматически сохраняется содержимое регистров процессора [1, 8, 48].

Поскольку между терминами «процесс» и «задача» со временем появилось существенное различие, мы сейчас подробно рассмотрим этот вопрос.

Процессы и задачи

Хотя понятия *мультипрограммного* и *мультизадачного режимов* работы достаточно близки, это все-таки не одно и то же. К сожалению, здесь до сих пор имеется некоторая путаница. Основные причины тому — не только то, что терминология еще не устоялась и что многие фирмы-разработчики по-разному предпочитали называть одни и те же сущности, но и сложность, неоднозначность ситуации.

Мультипрограммный режим предполагает, что операционная система организует параллельное выполнение нескольких вычислительных процессов на одном компьютере. И каждый вычислительный процесс может, в принципе, никак не зависеть от другого вычислительного процесса. Разве что они могут задержать выполнение друг друга из-за необходимости поочередно разделять ресурсы или сильно задерживать выполнение друг друга при владении неразделяемым ресурсом. У них может не быть ни общих файлов, ни общих переменных. Они вообще могут принадлежать разным пользователям. Просто эти процессы, с позиций внешнего наблюдателя, выполняются на одном и том же компьютере в одно и то же время. Хотя могут выполняться и в разное время, и на разных компьютерах. Главное — это то, что мультипрограммный режим обеспечивает для этих процессов их независимость. Каждому процессу операционная система выделяет затребованные ресурсы, он выполняется как бы на отдельной виртуальной машине. Средства защиты системы должны обеспечить невмешательство одного вычислительного процесса в другой вычислительный процесс. И если такую защиту обеспечить невозможно,

то система не может считаться надежной. Немало методов и конкретных способов было придумано разработчиками для обеспечения надежных вычислений и предотвращения возможности намеренно или по ошибке повлиять на результаты вычислений в другом процессе.

Однако существует и другая потребность: не разделить вычислительные процессы друг от друга, а наоборот совместить их, обеспечить возможность тесного взаимодействия между осуществляемыми вычислениями. Например, результаты вычислений одного вычислительного процесса могут требоваться для начала или продолжения работы другого. Существует огромное количество ситуаций, когда необходимо обеспечить активное взаимодействие между выполняющимися вычислительными процессами. Если нет возможности получить доступ к переменным другого процесса, ибо операционная система построена надежно и защищает адресные пространства одного вычислительного процесса от вмешательства другого вычислительного процесса, то возникают очень серьезные препятствия на пути передачи каких бы то ни было данных между процессами.

Термин *мультизадачный режим* работы стали применять как раз для тех случаев, когда необходимо обеспечить взаимодействие между вычислениями. Мультизадачный режим означает, что операционная система позволяет организовать параллельное выполнение вычислений, и имеются специальные механизмы для передачи данных, синхросигналов, каких-либо сообщений между этими взаимодействующими вычислениями. Это можно сделать за счет того, что такие вычисления не должны системой изолироваться друг от друга. Операционная система не должна для них в обязательном порядке задействовать все механизмы защиты вычислений от невмешательства друг в друга. При мультизадачном режиме разработчик программы должен позаботиться о разделении ресурсов между его задачами. Операционная система будет всего лишь разделять процессорное время между задачами.

Понятие *процесса* было введено для реализации идей мультипрограммирования. Термин *задача* тоже, к сожалению, в большинстве случаев применялся для того же. В свое время различали термины «мультизадачность» и «мультипрограммирование», но потом они стали заменять друг друга, и это вносило немалую путаницу. Таким образом, для реализации мультизадачности в ее исходном толковании необходимо было ввести соответствующую сущность. Такой сущностью стали *легковесные* (thin) *процессы*, или, как их теперь преимущественно называют, *потоки выполнения*[1], *нити*, или *треды* (threads).

Когда говорят о *процессах* (process), то тем самым хотят отметить, что операционная система поддерживает их обособленность: у каждого процесса имеется свое виртуальное адресное пространство, каждому процессу назначаются свои ресурсы — файлы, окна, семафоры и т. д. Такая обособленность нужна для того, чтобы защитить один процесс от другого, поскольку они, совместно используя все ресурсы вычислительной системы, конкурируют друг с другом за доступ к ресурсам. В общем случае процессы просто никак не связаны между собой и могут принадлежать даже разным пользователям, разделяющим одну вычислительную систему.

[1] Поток выполнения (thread) не следует путать с потоком данных (stream).

Другими словами, в случае процессов операционная система считает их совершенно несвязанными и независимыми. При этом именно операционная система берет на себя роль арбитра в спорах конкурирующих процессов за ресурсы. Она же и обеспечивает защиту выполняющихся вычислений.

Однако желательно иметь еще и возможность задействовать внутренний параллелизм, который может быть в самих процессах. Такой внутренний параллелизм встречается достаточно часто и позволяет ускорить вычисления. Например, некоторые операции, выполняемые приложением, могут требовать для своего исполнения достаточно длительное использование центрального процессора. В этом случае при интерактивной работе с приложением пользователь вынужден долго ожидать завершения заказанной операции и не может управлять приложением до тех пор, пока операция не выполнится до самого конца. Такие ситуации встречаются достаточно часто, например, при работе с графическими редакторами при обработке больших изображений с высокой степенью детализации. Если же программные модули, исполняющие такие длительные операции, оформлять в виде самостоятельных «подпроцессов» (легковесных процессов, потоков выполнения, или задач), которые могут выполняться параллельно с другими подпроцессами (потоками, задачами), то у пользователя появляется возможность параллельно выполнять несколько операций в рамках одного приложения (процесса). Легковесными эти процессы называют потому, что операционная система не должна для них организовывать полноценную виртуальную машину, то есть эти задачи, прежде всего, не имеют своих собственных ресурсов, а развиваются в том же виртуальном адресном пространстве, могут пользоваться теми же файлами, виртуальными устройствами и иными ресурсами, выделенными ОС данному процессу. Единственное, что они имеют свое — это процессорный ресурс. В однопроцессорной системе потоки выполнения (задачи) разделяют между собой процессорное время так же, как это делают обычные процессы, а в мультипроцессорной системе они могут выполняться одновременно, если не встречают конкуренции из-за обращения к иным ресурсам.

Главное, что обеспечивает многопоточность — это возможность параллельно выполнять несколько видов операций в одной прикладной программе. Параллельные вычисления (а следовательно, и более эффективное использование ресурсов центрального процессора, и меньшее суммарное время выполнения задач) гораздо удобнее реализовать не на уровне процессов, но на уровне задач (потоков, тредов). И программа, разработанная с использованием механизма потоков, представляемая как некоторое множество задач в рамках одного процесса, может быть выполнена быстрее за счет параллельного функционирования ее отдельных частей. Особенно это выгодно при наличии нескольких процессоров, ибо каждая задача может выполняться на отдельном процессоре. Например, если электронная таблица, текстовый процессор или графический редактор были разработаны с учетом возможностей многопоточной обработки, то пользователь может запросить пересчет своего рабочего листа, слияние нескольких документов или преобразование изображения и одновременно продолжать заполнять таблицу, открывать для редактирования следующий документ, изменять другое изображение.

Особенно эффективно можно использовать многопоточность для выполнения распределенных приложений. Например, многопоточный сервер может параллельно выполнять запросы сразу нескольких клиентов. Как известно, операционная система OS/2 была одной из первых систем, используемых в персональных компьютерах, которая поддерживала многопоточность. В середине 90-х годов для этой операционной системы было создано большое количество приложений, в которых наличие механизмов многопоточной обработки реально приводило к существенному повышению скорости вычислений. Для систем Windows, с которыми мы все имеем дело, ярко выраженной многопоточностью обладают такие продукты, как SQL Server, Oracle. И хотя те же Word, Excel, Internet Explorer также при своей работе образуют потоки, явного параллелизма в этих программах почти не поддерживается. Поэтому при увеличении числа процессоров в компьютере такие программы не начинают выполняться быстрее.

Итак, сущность «поток выполнения» была введена для того, чтобы именно с помощью этих единиц распределять процессорное время между возможными работами. Сущность «процесс» предполагает, что при диспетчеризации нужно учитывать все ресурсы, закрепленные за ним. При манипулировании задачами-потоками можно менять только контекст задачи, если мы переключаемся с одной задачи на другую в рамках одного процесса. Все остальные вычислительные ресурсы при этом не затрагиваются. Каждый процесс всегда состоит, по крайней мере, из одного потока выполнения, и только если имеется внутренний параллелизм, программист может «расщепить» один поток на несколько параллельных. Потребность в потоках возникла еще в однопроцессорных вычислительных системах, поскольку они позволяли организовать вычисления более эффективно. Для использования достоинств многопроцессорных систем с общей памятью потоки уже просто необходимы, так как позволяют не только реально ускорить выполнение тех задач, которые допускают их естественное распараллеливание, но и загрузить процессорные элементы работой, с тем чтобы они не простаивали. Заметим, однако, что желательно, чтобы можно было свести к минимуму взаимодействие потоков между собой, ибо ускорение от одновременного выполнения параллельных потоков может быть сведено к минимуму из-за задержек синхронизации и обмена данными.

Каждый поток выполняется строго последовательно и имеет свой собственный программный счетчик и стек. Потоки, как и процессы, могут порождать потоки-потомки, поскольку любой процесс состоит по крайней мере из одного потока. Подобно традиционным процессам (то есть процессам, состоящим из одного потока), каждый поток может находиться в одном из активных состояний. Пока один поток заблокирован (или просто находится в очереди готовых к исполнению задач), другой поток того же процесса может выполняться. Потоки разделяют процессорное время так же, как это делают обычные процессы, в соответствии с различными вариантами диспетчеризации.

Как уже упоминалось, иногда потоки выполнения называют легковесными процессами. Как мы уже знаем, все потоки имеют одно и то же виртуальное адресное пространство своего процесса. Это означает, что они разделяют одни и те же глобальные переменные. Поскольку каждый поток может иметь доступ к каждому

виртуальному адресу, один поток может использовать стек другого потока. Между потоками нет полной защиты, потому что, во-первых, это невозможно, а во-вторых, не нужно. Все потоки одного процесса всегда решают общую задачу одного пользователя, и механизм потоков используется здесь для более быстрого решения задачи путем ее распараллеливания. При этом программисту очень важно получить в свое распоряжение удобные средства организации взаимодействия частей одной программы. Повторим, что кроме разделения адресного пространства, все потоки разделяют также набор открытых файлов, устройства, выделенные процессу, имеют одни и те же наборы сигналов, семафоры и т. п. А что у потоков является их собственным? Собственными являются программный счетчик, стек, рабочие регистры процессора, потоки-потомки, состояние.

Вследствие того, что потоки, относящиеся к одному процессу, выполняются в одном и том же виртуальном адресном пространстве, между ними легко организовать тесное взаимодействие, в отличие от процессов, для которых нужны специальные механизмы обмена сообщениями и данными. Более того, программист, создающий многопоточное приложение, может заранее продумать работу множества потоков процесса таким образом, чтобы они могли взаимодействовать наиболее выгодным способом, а не конкурировать за ресурсы тогда, когда этого можно избежать.

Для того чтобы можно было эффективно организовать параллельное выполнение рассмотренных сущностей (процессов и потоков), в архитектуру современных процессоров включены средства для работы со специальной информационной структурой, описывающей ту или иную сущность. Для этого уже на уровне архитектуры микропроцессора используется понятие *задача* (task). Оно как бы объединяет в себе и обычный процесс, и поток выполнения (тред). Это понятие и поддерживаемая для него на уровне аппаратуры информационная структура позволяют в дальнейшем при разработке операционной системы строить соответствующие дескрипторы как для задач, так и для процессов. И отличаться эти дескрипторы будут прежде всего тем, что дескриптор задачи может хранить только контекст приостановленного вычислительного процесса, тогда как дескриптор процесса должен содержать поля, описывающие тем или иным способом ресурсы, выделенные этому процессу. Для хранения контекста задачи в микропроцессорах с архитектурой i32 имеется специальный сегмент состояния задачи (Task State Segment, TSS). А для отображения информации о процессе используется уже иная информационная структура, однако она включает в себя TSS. Другими словами, сегмент состояния задачи, подробно рассматриваемый в разделе «Адресация в 32-разрядных микропроцессорах i80x86 при работе в защищенном режиме» главы 4, используется как основа для дескриптора процесса. Таким образом, дескриптор процесса больше по размеру, чем TSS, и включает в себя такие традиционные поля, как идентификатор процесса, его имя, тип, приоритет и проч.

Каждый поток (в случае использования так называемой «плоской» модели памяти — см. раздел «Сегментная, страничная и сегментно-страничная организация памяти» в главе 3) может быть оформлен в виде самостоятельного сегмента, что приводит к тому, что простая (не многопоточная) программа будет иметь всего один сегмент кода в виртуальном адресном пространстве.

Теперь, если вернуться к уже упомянутому файлу CONFIG.SYS, в котором для операционной системы OS/2 указываются наиболее важные параметры, определяющие ее работу, стоит заметить, что в этом файле строка THREADS=1024 указывает на количество не процессов, а именно задач. И под задачей в данном случае понимается как процесс, так и поток этого процесса.

К большому сожалению, практически невозможно использовать термины «задача» и «процесс» с однозначным толкованием, чтобы под задачей обязательно понимать поток, в то время как термин «процесс» означал бы множество потоков. Значение этих терминов по-прежнему сильно зависит от контекста. И это характерно практически для каждой книги, в том числе и для учебной литературы. Грешен этим и автор. Остается надеяться, что со временем все же ситуация изменится, и толкование этих слов будет более четким и строгим.

В завершение можно привести несколько советов по использованию потоков выполнения при создании приложений, заимствованных из [28].

- В случае однопроцессорной системы множество параллельных потоков часто не ускоряет работу приложения, поскольку в каждый отдельно взятый промежуток времени возможно выполнение только одного потока. Кроме того, чем больше у вас потоков, тем больше нагрузки на систему, относящиеся к переключению между ними. Мультизадачность из более двух постоянно работающих потоков в вашем проекте не сделает программу быстрее, если каждый из потоков не будет требовать частого ввода-вывода.
- Вначале нужно понять, для чего необходим поток. Поток, осуществляющий обработку, может помешать системе быстро реагировать на запросы ввода-вывода. Потоки позволяют программе отзываться на просьбы пользователя и устройств, но при этом (в том числе) сильно загружают процессор. Потоки позволяют компьютеру одновременно обслуживать множество устройств, и созданный вами поток, отвечающий за обработку специфического устройства, как минимум может потребовать столько времени, сколько системе необходимо для обработки запросов от всех устройств.
- Потокам можно назначать разные приоритеты для того, чтобы наименее значимые процессы выполнялись в фоновом режиме. Это путь честного разделения ресурсов процессора. Однако необходимо осознать тот факт, что процессор один на всех, а потоков много. Если в вашей программе главная процедура передает нечто для обработки в низкоприоритетный поток, то сама программа становится просто неуправляемой.
- Потоки хорошо работают, когда они независимы. Но они начинают работать непродуктивно, когда вынуждены часто синхронизироваться для доступа к общим ресурсам. Взаимные блокировки и критические секции отнюдь не добавляют скорости работы системы, хотя без использования этих механизмов взаимодействующие вычисления организовывать нельзя.
- Помните, что память виртуальна. Механизм виртуальной памяти (см. раздел «Память и отображения, виртуальное адресное пространство» в главе 3) следит за тем, какая часть виртуального адресного пространства должна находиться в оперативной памяти, а какая должна быть сброшена в файл подкачки.

Потоки усложняют ситуацию, если они обращаются в одно и то же время к разным адресам виртуального адресного пространства приложения. Это значительно увеличивает нагрузку на систему, особенно при небольшом объеме кэш-памяти. Помните, что реально память не всегда «свободна», как это пишут в информационных окошках «О системе». Всегда отождествляйте доступ к памяти с доступом к файлу на диске и создавайте приложение с учетом вышесказанного.

- Всякий раз, когда любой из ваших потоков пытается воспользоваться общим ресурсом вычислительного процесса, которому он принадлежит, вы обязаны каким-то образом легализовать и защитить вашу деятельность. Хорошим средством для этого являются критические секции, семафоры и очереди сообщений (см. главу 7). Если вы протестировали ваше приложение и не обнаружили ошибок синхронизации, то это еще не значит, что их там нет. Пользователь может создавать самые непредсказуемые ситуации. Это очень ответственный момент в разработке многопоточных приложений.
- Не возлагайте на поток несколько функций. Сложные функциональные отношения затрудняют понимание общей структуры приложения, его алгоритм. Чем проще и однозначнее каждая из рассматриваемых ситуаций, тем больше вероятность, что ошибок удастся избежать.

Основные виды ресурсов и возможности их разделения

Рассмотрим кратко основные виды ресурсов вычислительной системы и способы их разделения (см. рис. 1.5). Прежде всего, одним из важнейших ресурсов является сам процессор[1], точнее — процессорное время. Процессорное время делится попеременно (параллельно). Имеется множество методов разделения этого ресурса (см. раздел «Планирование и диспетчеризация процессов и задач» в главе 2).

Вторым видом ресурсов вычислительной системы можно считать память. Оперативная память может делиться и одновременно (то есть в памяти одновременно может располагаться несколько задач или, по крайней мере, текущих фрагментов, участвующих в вычислениях), и попеременно (в разные моменты оперативная память может предоставляться для разных вычислительных процессов). Память — очень интересный вид ресурса. Дело в том, что в каждый конкретный момент времени процессор при выполнении вычислений обращается к очень ограниченному числу ячеек оперативной памяти. С этой точки зрения желательно память выделять для возможно большего числа параллельно исполняемых задач. С другой стороны, как правило, чем больше оперативной памяти может быть выделено для конкретного текущего вычислительного процесса, тем лучше будут условия его выполнения. Поэтому проблема эффективного разделения оперативной памяти между параллельно выполняемыми вычислительными процессами является од-

[1] Разговор о процессоре как об одном из ресурсов более характерен для мультипроцессорных систем. В случае однопроцессорных систем чаще говорят о процессорном времени.

ной из самых актуальных. Достаточно подробно вопросы распределения памяти между параллельно выполняющимися процессами рассмотрены в главе 3.

Внешняя память тоже является ресурсом, который часто необходим для выполнения вычислений. Когда говорят о внешней памяти (например, памяти на магнитных дисках), то собственно память и доступ[1] к ней считаются разными видами ресурса. Каждый из этих ресурсов может предоставляться независимо от другого. Но для полноценной работы с внешней памятью необходимо иметь оба этих ресурса. Собственно внешняя память может разделяться и одновременно, а вот доступ к ней всегда разделяется попеременно.

Если говорить о внешних устройствах, то они, как правило, могут разделяться параллельно, если используются механизмы прямого доступа. Если же устройство работает с последовательным доступом, то оно не может считаться разделяемым ресурсом. Простыми и наглядными примерами внешних устройств, которые не могут быть разделяемыми, являются принтер и накопитель на магнитной ленте. Действительно, если допустить, что принтер можно разделять между двумя процессами, которые смогут его использовать (управлять его работой) попеременно, то результаты печати, скорее всего, окажутся негодными — фрагменты выведенного текста могут перемешаться таким образом, что будет не понятно, что есть что. Аналогично и для накопителя на магнитной ленте. Если один процесс начнет что-то читать или писать, а второй при этом запросит перемотку ленты на ее начало, то оба вычислительных процесса не смогут выполнить свои вычисления. Здесь следует заметить, что при работе с устройствами печати мы, тем не менее, явно наблюдаем возможность печатать из разных программ, выполняющихся параллельно. Однако необходимо знать, что это реализуется за счет того, что каждый вычислительный процесс получает свой виртуальный принтер, который он ни с кем не разделяет. А операционная система, получив задания на печать от выполняющихся задач, сама упорядочивает эти задания и передает очередное задание на принтер только после полного завершения предыдущего задания.

Очень важным видом ресурсов являются *программные модули*. Прежде всего, мы будем рассматривать системные программные модули, поскольку именно они обычно считаются программными ресурсами и поэтому в принципе могут распределяться между выполняющимися процессами.

Как известно, программные модули могут быть однократно используемыми и многократно (или повторно) используемыми. *Однократно используемыми* называют такие программные модули, которые могут быть правильно выполнены только один раз, то есть в процессе своего выполнения они могут испортить себя: либо повреждается часть кода, либо исходные данные, от которых зависит ход вычислений. Очевидно, что однократно используемые программные модули являются неделимым ресурсом. Более того, их, как правило, вообще не распределяют как ресурс системы. Системные однократно используемые программные модули, как правило, задействуются только на этапе загрузки операционной системы. При этом следует иметь в виду тот очевидный факт, что собственно двоичные файлы, которые

[1] Процесс обращения к данным.

обычно хранятся на системном диске и в которых и записаны эти модули, не портятся, а потому могут быть повторно использованы при следующем запуске операционной системы.

Повторно используемые программные модули, в свою очередь, могут быть непривилегированными, привилегированными и реентерабельными. Все они допускают корректное повторное выполнение программного кода при обращении к нему из другой программы.

Привилегированные программные модули работают в так называемом привилегированном режиме, то есть при отключенной системе прерываний (часто говорят, что прерывания закрыты), когда никакие внешние события не могут нарушить естественный порядок вычислений. Как результат, программный модуль выполняется до своего конца, после чего он может быть вновь вызван на исполнение из другой задачи (другого вычислительного процесса). С позиций стороннего наблюдателя по отношению к вычислительным процессам, которые попеременно (причем, возможно, неоднократно) в течение срока своей «жизни» вызывают некоторый привилегированный программный модуль, такой модуль будет выступать как попеременно разделяемый ресурс. Структура привилегированных программных модулей изображена на рис. 1.8. Здесь в первой секции программного модуля выключается система прерываний. Следовательно, при выполнении вычислений в первой секции ничто не может их прервать, и беспокоиться о промежуточных переменных нет необходимости. В последней секции, напротив, система прерываний включается. Даже если тут же возникнет прерывание и другой процесс запросит этот же привилегированный модуль, все равно все вычисления уже выполнены и ничто не сможет их испортить.

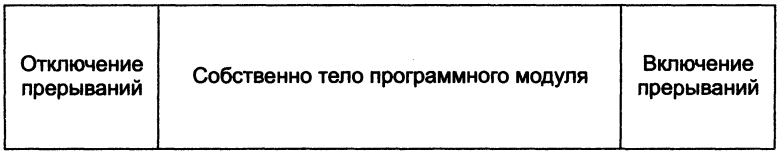


Рис. 1.8. Структура привилегированного программного модуля

Непривилегированные программные модули — это обычные программные модули, которые могут быть прерваны во время своей работы. Следовательно, такие модули в общем случае нельзя считать разделяемыми, потому что если после прерывания выполнения такого модуля, исполняемого в рамках одного вычислительного процесса, запустить его еще раз по требованию другого вычислительного процесса, то промежуточные результаты для прерванных вычислений могут быть потеряны.

В противоположность этому, *реентерабельные программные модули*[1] допускают повторное многократное прерывание своего исполнения и повторный их запуск по обращению из других задач (вычислительных процессов). Для этого реентерабельные программные модули должны быть созданы таким образом, чтобы было

[1] Реентерабельный — допускающий повторные прерывания (дословный перевод с английского слова «re-enterable»).

обеспечено сохранение промежуточных результатов для прерываемых вычислений и возврат к ним, когда вычислительный процесс возобновляется с прерванной ранее точки. Это может быть реализовано двумя способами: с помощью статических и динамических методов выделения памяти под сохраняемые значения. Основным и наиболее часто используемым является динамический способ выделения памяти для сохранения всех промежуточных результатов вычисления, относящихся к реентерабельному программному модулю (рис. 1.9).

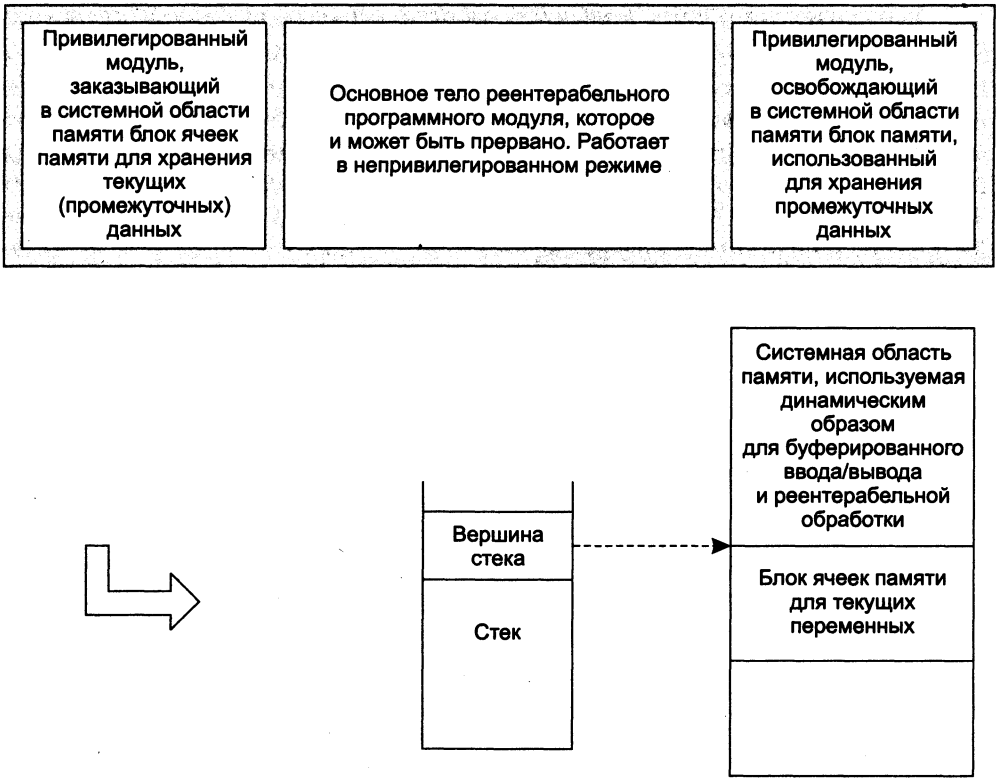


Рис. 1.9. Структура реентерабельного программного модуля

Основная идея построения и работы реентерабельного программного модуля заключается в том, что в первой (головной) своей части путем обращения из системной привилегированной секции осуществляется запрос на получение в системной области памяти блока ячеек, необходимого для размещения всех текущих (промежуточных) данных. При этом на вершину стека помещается указатель на начало области данных и ее объем. Все текущие переменные реентерабельного программного модуля в этом случае располагаются в системной области памяти. Адресация этих переменных осуществляется относительно вершины стека. Поскольку в конце привилегированной секции система прерываний включается, то во время работы центральной (основной) части реентерабельного модуля возможно ее прерывание. Если прерывания не возникает, то в третьей (заключительной) секции

осуществляется запрос на освобождение используемого блока системной области памяти. При освобождении этой области памяти модифицируется значение стека. Если же во время работы центральной секции возникает прерывание, и другой вычислительный процесс обращается к тому же самому реентерабельному программному модулю, то для этого нового процесса вновь заказывается новый блок памяти в системной области памяти, и на вершину стека записывается новый указатель. Очевидно, что возможно многократное повторное вхождение в реентерабельный программный модуль до тех пор, пока в области системной памяти, выделяемой специально для реентерабельной обработки, есть свободные области, объема которых достаточно для выделения нового блока.

Что касается статического способа выделения памяти, то здесь речь может идти, например, о том, что заранее для фиксированного числа вычислительных процессов резервируются области памяти, в которых будут располагаться переменные реентерабельных программных модулей: для каждого процесса — своя область памяти. Чаще всего в качестве таких процессов выступают процессы ввода-вывода, и речь идет о реентерабельных драйверах[1].

Кроме реентерабельных программных модулей существуют еще *повторно входимые* (re-entrance). Этим термином называют программные модули, которые тоже допускают свое многократное параллельное использование, но, в отличие от реентерабельных, их нельзя прерывать. Повторно входимые программные модули состоят из привилегированных секций, и повторное обращение к ним возможно только после завершения какой-нибудь из таких секций. После выполнения очередной такой привилегированной секции управление может быть передано супервизору, который может предоставить возможность выполняться другой задаче, а значит, возможно повторное вхождение в рассматриваемый программный модуль. Другими словами, в повторно входимых программных модулях четко предопределены все допустимые (возможные) точки входа. Следует отметить, что повторно входимые программные модули встречаются гораздо чаще реентерабельных (повторно прерываемых).

Наконец, имеются и информационные ресурсы, то есть в качестве ресурсов могут выступать данные. Информационные ресурсы могут существовать как в виде переменных, находящихся в оперативной памяти, так и в виде файлов. Если процессы используют данные только для чтения, то такие информационные ресурсы можно разделять. Если же процессы могут изменять информационные ресурсы, то необходимо специальным образом организовывать работу с такими данными. Это одна из наиболее сложных проблем, достаточно подробно она обсуждается в главах 9 и 10.

Классификация операционных систем

Выше мы уже дали определение операционной системы (ОС). Поэтому просто повторим, что основным предназначением ОС является организация эффективных и надежных вычислений, создание различных интерфейсов для взаимодействия с этими вычислениями и с самой вычислительной системой.

[1] Реентерабельный драйвер может управлять параллельно несколькими однотипными устройствами — более подробно см. в главе 5.

Широко известно высказывание, согласно которому любая наука начинается с классификации. Само собой, что вариантов классификации может быть очень много, здесь все будет зависеть от выбранного признака, по которому один объект мы будем отличать от другого. Однако, что касается ОС, здесь уже давно сформировалось относительно небольшое количество классификаций: по назначению, по режиму обработки задач, по способу взаимодействия с системой и, наконец, по способам построения (архитектурным особенностям системы).

Прежде всего, традиционно различают ОС общего и специального назначения. ОС специального назначения, в свою очередь, подразделяются на ОС для носимых микрокомпьютеров и различных встроенных систем, организации и ведения баз данных, решения задач реального времени и т. п. Еще не так давно операционные системы для персональных компьютеров относили к ОС специального назначения. Сегодня современные мультизадачные ОС для персональных компьютеров уже многими относятся к ОС общего назначения, поскольку их можно использовать для самых разнообразных целей — так велики их возможности.

По режиму обработки задач различают ОС, обеспечивающие однопрограммный и мультипрограммный (мультизадачный) режимы. К однопрограммным ОС относится, например, всем известная, хотя нынче уже практически и не используемая MS DOS. Напомним, что под *мультипрограммированием* понимается способ организации вычислений, когда на однопроцессорной вычислительной системе создается видимость одновременного выполнения нескольких программ. Любая задержка в решении программы (например, для осуществления операций ввода-вывода данных) используется для выполнения других (таких же либо менее важных) программ. Иногда при этом говорят о мультизадачном режиме, причем, вообще говоря, термины «мультипрограммный режим» и «мультизадачный режим» — это не синонимы, хотя и близкие понятия. Основное принципиальное отличие этих терминов заключается в том, что мультипрограммный режим обеспечивает параллельное выполнение нескольких приложений, и при этом программисты, создающие эти программы, не должны заботиться о механизмах организации их параллельной работы (эти функции берет на себя сама ОС; именно она распределяет между выполняющимися приложениями ресурсы вычислительной системы, осуществляет необходимую синхронизацию вычислений и взаимодействие). Мультизадачный режим, наоборот, предполагает, что забота о параллельном выполнении и взаимодействии приложений ложится как раз на прикладных программистов. Хотя в современной технической и тем более научно-популярной литературе об этом различии часто забывают и тем самым вносят некоторую путаницу. Можно, однако, заметить, что современные ОС для персональных компьютеров реализуют и мультипрограммный, и мультизадачный режимы.

Если принимать во внимание способ взаимодействия с компьютером, то можно говорить о диалоговых системах и системах пакетной обработки. Доля последних хоть и не убывает в абсолютном исчислении, но в процентном отношении она существенно сократилась по сравнению с диалоговыми системами.

При организации работы с вычислительной системой в диалоговом режиме можно говорить об однопользовательских (однотерминальных) и мультитерминаль-

ных ОС. В мультитерминальных ОС с одной вычислительной системой одновременно могут работать несколько пользователей, каждый со своего терминала. При этом у пользователей возникает иллюзия, что у каждого из них имеется собственная вычислительная система. Очевидно, что для организации мультитерминального доступа к вычислительной системе необходимо обеспечить мультипрограммный режим работы. В качестве одного из примеров мультитерминальных операционных систем для персональных компьютеров можно назвать Linux. Некая имитация мультитерминальных возможностей имеется и в системе Windows XP. В этой операционной системе каждый пользователь после регистрации (входа в систему) получает свою виртуальную машину. Если необходимо временно предоставить компьютер другому пользователю, вычислительные процессы первого можно не завершать, а просто для этого другого пользователя система создает новую виртуальную машину. В результате компьютер будет выполнять задачи и первого, и второго пользователя. Количество параллельно работающих виртуальных машин определяется имеющимися ресурсами.

Основной особенностью *операционных систем реального времени* (ОСРВ) является обеспечение обработки поступающих заданий в течение заданных интервалов времени, которые нельзя превышать. Поток заданий в общем случае не является планомерным и не может регулироваться оператором (характер следования событий можно предсказать лишь в редких случаях), то есть задания поступают в непредсказуемые моменты времени и без всякой очередности. В то время как в ОС, не предназначенных для решения задач реального времени, имеются некоторые накладные расходы процессорного времени на этапе инициирования задач (в ходе которого ОС распознает все пожелания пользователей относительно решения своих задач, загружает в оперативную память нужную программу и выделяет другие необходимые для ее выполнения ресурсы), в ОСРВ подобные затраты могут отсутствовать, так как набор задач обычно фиксирован, и вся информация о задачах известна еще до поступления запросов. Для подлинной реализации режима реального времени необходима (хотя этого и недостаточно) организация мультипрограммирования. Мультипрограммирование является основным средством повышения производительности вычислительной системы, а для решения задач реального времени производительность становится важнейшим фактором. Лучшие характеристики по производительности для систем реального времени обеспечиваются однотерминальными ОСРВ. Средства организации мультитерминального режима всегда замедляют работу системы в целом, но расширяют функциональные возможности системы. Одной из наиболее известных ОСРВ для персональных компьютеров является ОС QNX.

По основному архитектурному принципу операционные системы разделяются на *микроядерные* и *макроядерные* (*монолитные*). В некоторой степени это разделение тоже условно, однако можно в качестве яркого примера микроядерной ОС привести ОСРВ QNX, тогда как в качестве монолитной можно назвать Windows 95/98 или ОС Linux. Если ядро ОС Windows мы не можем изменить, нам недоступны его исходные коды и у нас нет программы для сборки (компиляции) этого ядра, то в случае с Linux мы можем сами собрать то ядро, которое нам необходимо, включив в него те программные модули и драйверы, которые мы считаем целесообразным включить именно в ядро (ведь к ним можно обращаться и из ядра).

Контрольные вопросы и задачи

1. Что такое операционная система? Перечислите основные функции операционных систем.
2. Что означает термин «авторизация»? Что означает термин «аутентификация»? Какая из этих операций выполняется раньше и почему?
3. Что такое операционная среда? Какие основные, наиболее известные операционные среды вы можете перечислить?
4. Что такое прерывание? Какие шаги выполняет система прерываний при возникновении запроса на прерывание? Какие бывают прерывания?
5. Перечислите известные дисциплины обслуживания прерываний; объясните, как можно реализовать каждую из этих дисциплин.
6. С какой целью в операционные системы вводится специальный системный модуль, иногда называемый супервизором прерываний?
7. Как можно и как следует толковать процесс — одно из основных понятий операционных систем? Объясните, в чем заключается различие между такими понятиями, как «процесс» и «задача»?
8. Изобразите диаграмму состояний процесса, поясните все возможные переходы из одного состояния в другое.
9. Объясните значения терминов «задача», «процесс», «поток выполнения»? Как они между собой соотносятся?
10. Для чего каждая задача получает соответствующий дескриптор? Какие поля, как правило, содержатся в дескрипторе процесса (задачи)? Что такое «контекст задачи»?
11. Объясните понятие ресурса. Почему понятие ресурса является одним из фундаментальных при рассмотрении операционных систем? Какие виды и типы ресурсов вы знаете?
12. Как вы считаете, сколько и каких списков дескрипторов задач может быть в системе? От чего должно зависеть это число?
13. В чем заключается различие между повторно входимыми и реентерабельными программными модулями? Как они реализуются?
14. Что такое привилегированный программный модуль? Почему нельзя создать мультипрограммную операционную систему, в которой бы не было привилегированных программных модулей?

Глава 2. Управление задачами

Понятия *процесса* (process) и *потока выполнения* (thread) нам уже известны. Мы теперь знаем, в чем здесь имеется сходство, а в чем — существенное различие. Однако в данной главе при рассмотрении вопросов распределения процессорного времени мы не всегда будем разделять эти понятия. Дело в том, что по отношению к этому ресурсу — процессорному времени — оба этих понятия практически эквиваленты. Они выступают просто как некоторая работа, для выполнения которой необходимо предоставить центральный процессор. Поэтому мы будем в основном использовать термин *задача* (task), который является как бы обобщающим. Ведь каждый поток выполнения на самом деле получает статус задачи, и для него создается соответствующий дескриптор. Но мы должны помнить о различиях между дескриптором процесса и дескриптором задачи. Даже если процесс состоит из единственного потока, мы говорим о дескрипторе процесса, содержащем информацию, с помощью которой операционная система отслеживает все ресурсы, необходимые процессу для его выполнения. Один из основных модулей супервизора операционной системы — *диспетчер задач* — переводит процессы в одно из состояний в зависимости от того, доступен тот или иной ресурс или не доступен. И поскольку в мультизадачной системе любой процесс содержит хотя бы один поток, то потоку (то есть задаче) ставится в соответствие дескриптор задачи, в котором сохраняется контекст этих вычислений. Сказанное справедливо для мультипрограммных систем, поддерживающих мультизадачный режим. В мультипрограммных системах, не поддерживающих мультизадачность, контекст прерванного процесса хранится в дескрипторе этого процесса. Заметим, что повсеместно распространенные системы Windows 9x/NT/2000/XP являются и мультипрограммными, и мультизадачными. Не случайно начиная с Windows NT и Windows 95 компания Microsoft отказалась от термина «задача» и стала использовать понятия процесса и потока выполнения (треда, нити). Правда, для изложения вопросов диспетчеризации это становится неудобным, ибо здесь чаще используется обобщающее понятие.

Еще одним доводом в пользу термина «задача» при рассмотрении вопросов организации распределения процессорного времени между выполняющимися вычислениями является аналогичный выбор этой сущности разработчиками про-

цессоров. Именно для отображения этой ситуации и обеспечения дополнительными возможностями системных программистов в решении вопросов распределения процессорного времени они вводят специальные информационные структуры и аппаратную поддержку для работы с ними. Во многих современных микропроцессорах, предназначенных для построения на их основе мощных мультипрограммных и мультизадачных систем, имеются *дескрипторы задач*. Примером, подтверждающим этот тезис, являются микропроцессоры, совместимые с архитектурой ia32, то есть с 32-разрядными процессорами фирмы Intel. Основные архитектурные особенности этих микропроцессоров, специально проработанные для организации мультизадачных операционных систем, рассматриваются достаточно подробно в главе 4. Здесь мы лишь отметим тот факт, что в этих процессорах имеется специальная аппаратная поддержка организации мультизадачного (и мультипрограммного) режима. Речь идет о *сегменте состояния задачи* (Task State Segment, TSS), который предназначен, прежде всего, для сохранения контекста потока или процесса и который легко позволяет организовать и мультипрограммный, и мультизадачный режимы. Не случайно был введен термин «задача», ибо он здесь применим и по отношению к полноценному вычислительному процессу, и по отношению к легковесному процессу (потоку выполнения, треду, нити). На самом деле этот аппаратный механизм применяется гораздо реже, чем об этом думали разработчики архитектуры ia32. На практике оказалось, что для сохранения контекста потоков эффективнее использовать программные механизмы, хотя они и не обеспечивают такой же надежности, как аппаратные.

Итак, операционная система выполняет следующие основные функции, связанные с управлением процессами и задачами:

- создание и удаление задач;
- планирование процессов и диспетчеризация задач;
- синхронизация задач, обеспечение их средствами коммуникации.

Создание задачи сопряжено с формированием соответствующей информационной структуры, а ее удаление — с расформированием. Создание и удаление задач осуществляется по соответствующим запросам от пользователей или от самих задач. Задача может породить новую задачу. При этом между задачами появляются «родственные» отношения. Порождающая задача называется «отцом», «родителем», а порожденная — «потомком». Отец может приостановить или удалить свою дочернюю задачу, тогда как потомок не может управлять отцом.

Процессор является одним из самых необходимых ресурсов для выполнения вычислений. Поэтому способы распределения времени центрального процессора между выполняющимися задачами сильно влияют и на скорость выполнения отдельных вычислений, и на общую эффективность вычислительной системы.

Основным подходом в организации того или иного метода управления процессами, обеспечивающего эффективную загрузку ресурсов или выполнение каких-либо иных целей, является организация очередей процессов и ресурсов. При распреде-

лении процессорного времени между задачами также используется механизм очередей.

Решение вопросов, связанных с тем, какой задаче следует предоставить процессорное время в данный момент, возлагается на специальный модуль операционный системы, чаще всего называемый *диспетчером задач*. Вопросы же подбора вычислительных процессов, которые не только можно, но и целесообразно решать параллельно, возлагаются на *планировщик процессов*.

Вопросы синхронизации задач и обеспечение их различными средствами передачи сообщений и данных между ними вынесены в отдельную главу, и сейчас мы их рассматривать не будем.

Планирование и диспетчеризация процессов и задач

Когда говорят о диспетчеризации, то всегда в явном или неявном виде подразумевают понятие задачи (потока выполнения). Если операционная система не поддерживает механизм потоковых вычислений, то можно заменять понятие задачи понятием процесса. Ко всему прочему, часто понятие задачи используется в таком контексте, что для его трактовки приходится использовать термин «процесс».

Очевидно, что на распределение ресурсов влияют конкретные потребности тех задач, которые должны выполняться параллельно. Другими словами, можно столкнуться с ситуациями, когда невозможно эффективно распределять ресурсы с тем, чтобы они не простаивали. Например, пусть всем выполняющимся процессам требуется некоторое устройство с последовательным доступом. Но поскольку, как мы уже знаем, оно не может разделяться между параллельно выполняющимися процессами, то процессы вынуждены будут очень долго ждать своей очереди, то есть недоступность одного ресурса может привести к тому, что длительное время не будут использоваться многие другие ресурсы.

Если же мы возьмем такой набор процессов, что они не будут конкурировать между собой за неразделяемые ресурсы при своем параллельном выполнении, то, скорее всего, процессы смогут выполниться быстрее (из-за отсутствия дополнительных ожиданий), да и имеющиеся в системе ресурсы, скорее всего, будут использоваться более эффективно. Таким образом, возникает задача подбора такого множества процессов, которые при своем выполнении будут как можно реже конфликтовать за имеющиеся в системе ресурсы. Такая задача называется *планированием вычислительных процессов*.

Задача планирования процессов возникла очень давно — в первых пакетных операционных системах при планировании пакетов задач, которые должны были выполняться на компьютере и по возможности бесконфликтно и оптимально использовать его ресурсы. В настоящее время актуальность этой задачи стала меньше. На первый план уже очень давно вышли задачи динамического (или краткосрочного) планирования, то есть текущего наиболее эффективного распределения ресурсов, возникающего практически по каждому событию. Задачи динамического

планирования стали называть *диспетчеризацией*[1]. Очевидно, что планирование процессов осуществляется гораздо реже, чем текущее распределение ресурсов между уже выполняющимися задачами. Основное различие между долгосрочным и краткосрочным планировщиками заключается в частоте их запуска, например: краткосрочный планировщик может запускаться каждые 30 или 100 мс, долгосрочный — один раз в несколько минут (или чаще; тут многое зависит от общей длительности решения заданий пользователей).

Долгосрочный планировщик решает, какой из процессов, находящихся во входной очереди, в случае освобождения ресурсов памяти должен быть переведен в очередь процессов, готовых к выполнению. Долгосрочный планировщик выбирает процесс из входной очереди с целью создания неоднородной мультипрограммной смеси. Это означает, что в очереди готовых к выполнению процессов должны находиться в разной пропорции как процессы, ориентированные на ввод-вывод, так и процессы, ориентированные преимущественно на активное использование центрального процессора.

Краткосрочный планировщик решает, какая из задач, находящихся в очереди готовых к выполнению, должна быть передана на исполнение. В большинстве современных операционных систем, с которыми мы сталкиваемся, долгосрочный планировщик отсутствует.

Планирование вычислительных процессов и стратегии планирования

Прежде всего, следует отметить, что при рассмотрении стратегий планирования, как правило, идет речь о краткосрочном планировании, то есть о диспетчеризации. Долгосрочное планирование, как мы уже отметили, заключается в подборе таких вычислительных процессов, которые бы меньше всего конкурировали между собой за ресурсы вычислительной системы. Иногда используется термин *стратегия обслуживания*.

Стратегия планирования определяет, какие процессы мы планируем на выполнение для того, чтобы достичь поставленной цели. Известно большое количество различных стратегий выбора процесса, которому необходимо предоставить процессор. Среди них, прежде всего, можно выбрать следующие:

- по возможности заканчивать вычисления (вычислительные процессы) в том же самом порядке, в котором они были начаты;
- отдавать предпочтение более коротким вычислительным задачам;
- предоставлять всем пользователям (процессам пользователей) одинаковые услуги, в том числе и одинаковое время ожидания.

---

[1] К сожалению, здесь наблюдается терминологическая несогласованность. Собственно модули супервизора, отвечающие за диспетчеризацию задач, часто называют планировщиками (scheduler). Однако фактически, говоря о тех же планировщиках памяти или о каких-нибудь других модулях, отвечающих за динамическое распределение ресурсов, имеют в виду, что эти планировщики осуществляют диспетчеризацию. Наконец, иногда диспетчеризацию называют краткосрочным планированием.

Когда говорят о стратегии обслуживания, всегда имеют в виду понятие процесса, а не понятие задачи, поскольку процесс, как мы уже знаем, может состоять из нескольких потоков выполнения (задач).

На сегодняшний день абсолютное большинство компьютеров — это персональные IBM-совместимые компьютеры, работающие на платформах Windows компании Microsoft. Это однопользовательские диалоговые мультипрограммные и мультизадачные системы. При создании операционных систем для персональных компьютеров разработчики, прежде всего, стараются обеспечить комфортную работу с системой, то есть основные усилия уходят на проработку пользовательского интерфейса. Что касается эффективности организации вычислений, то она, видимо, тоже должна оцениваться с этих позиций. Если же считать системы Windows операционными системами общего назначения, что тоже возможно, ибо эти системы повсеместно используют для решения самых разнообразных задач автоматизации, то также следует признать, что принятые в системах Windows стратегии обслуживания приводят к достаточно высокой эффективности вычислений. Некоторым даже удается использовать системы Windows NT/2000 для решения задач реального времени. Однако выбор этих операционных систем для таких задач скорее всего делается либо вследствие некомпетентности, либо из-за невысоких требований ко времени отклика и гарантиям обслуживания со стороны самих систем реального времени, которые реализуются на Windows NT/2000.

Прежде всего, система, ориентированная на однопользовательский режим, должна обеспечить хорошую реакцию системы на запросы от того приложения, с которым сейчас пользователь работает. Мало пользователей, которые могут параллельно работать с большим числом приложений. Поэтому по умолчанию для задачи, с которой пользователь непосредственно работает и которую называют *задачей переднего плана* (foreground task), система устанавливает более высокий уровень приоритета. В результате процессорное время прежде всего предоставляется текущей задаче пользователя, и он не будет испытывать лишний раз дискомфорт из-за медленной реакции системы на его запросы. Для обеспечения надлежащей работы коммуникационных процессов и для возможности выполнять системные функции приоритет задач пользователя должен быть ниже, чем у тех задач, которые реализуют операции ввода-вывода и иные управляющие функции.

Например, в Windows 2000 можно открыть окно **Свойства системы**, перейти на вкладку **Дополнительно**, щелчком на кнопке **Параметры быстродействия** открыть одноименное окно и с помощью переключателя в разделе **Отклик приложений** установить режим **Оптимизировать быстродействие приложений**. Это будет соответствовать выбору такой стратегии диспетчеризации задач, в соответствии с которой приоритет на получение процессорного времени будут иметь задачи пользователя, а не фоновые служебные вычисления. В предыдущей версии ОС — Windows NT 4.0 — для выбора нужной ему стратегии пользователь должен был на вкладке **Быстродействие** окна **Свойства системы** установить желаемое значение в поле **Ускорение приложения переднего плана**. Это ускорение можно сделать максимальным (по умолчанию), а можно его свести к нулю. Последний вариант означал бы, что все запущенные пользователем приложения будут иметь одинаковый приори-

тет. Последнее важно, если пользователь часто запускает сразу по нескольку задач, каждая из которых требует длительных вычислений, причем эти приложения часто используют операции ввода-вывода. Например, если нужно обработать несколько десятков музыкальных или графических файлов, причем каждый файл имеет большие размеры, то выполнение всей этой работы как множества параллельно исполняющихся задач будет завершено за меньшее время, если указать стратегию равенства обслуживания. Должно быть очевидным, что любой другой вариант решения этой задачи потребует больше времени. Например, последовательное выполнение задач обработки каждого файла (то есть обработка следующего файла может начинаться только по окончании обработки предыдущего) приведет к самому длительному варианту. Стратегия предоставления процессорного времени в первую очередь текущей задаче пользователя, которая установлена в системах Windows по умолчанию, приведет нас к промежуточному (по затратам времени) результату.

Очевидно, что в идеале в очереди готовых к выполнению задач должны находиться в разной пропорции как задачи, ориентированные на ввод-вывод, так и задачи, ориентированные преимущественно на работу с центральным процессором. Практически все операционные системы стараются учесть это требование, однако не всегда оно выполняется настолько удачно, что пользователь получает превосходное время реакции системы на свои запросы и при этом видит, что его ресурсоемкие приложения выполняются достаточно быстро.

Дисциплины диспетчеризации

Известно большое количество *дисциплин диспетчеризации*, то есть правил формирования очереди готовых к выполнению задач, в соответствии с которыми формируется эта очередь (список). Иногда их называют *дисциплинами обслуживания*, опуская тот факт, что речь идет о распределении процессорного времени. Одни дисциплины диспетчеризации дают наилучшие результаты для одной стратегии обслуживания, в то время как для другой стратегии они могут быть вовсе неприемлемыми. Известно большое количество дисциплин диспетчеризации. Мы же, несмотря на статус этой книги, рассмотрим далеко не все, а только те, которые признаны наиболее эффективными и до сих пор имеют применение.

Прежде всего, различают два больших класса дисциплин обслуживания: *бесприоритетные* и *приоритетные*. При бесприоритетном обслуживании выбор задач производится в некотором заранее установленном порядке без учета их относительной важности и времени обслуживания. При реализации приоритетных дисциплин обслуживания отдельным задачам предоставляется преимущественное право попасть в состояние исполнения. Перечень дисциплин обслуживания и их классификация приведены на рис. 2.1.

В концепции приоритетов имеем следующие варианты:

- приоритет, присвоенный задаче, является величиной постоянной;
- приоритет изменяется в течение времени решения задачи (*динамический приоритет*).

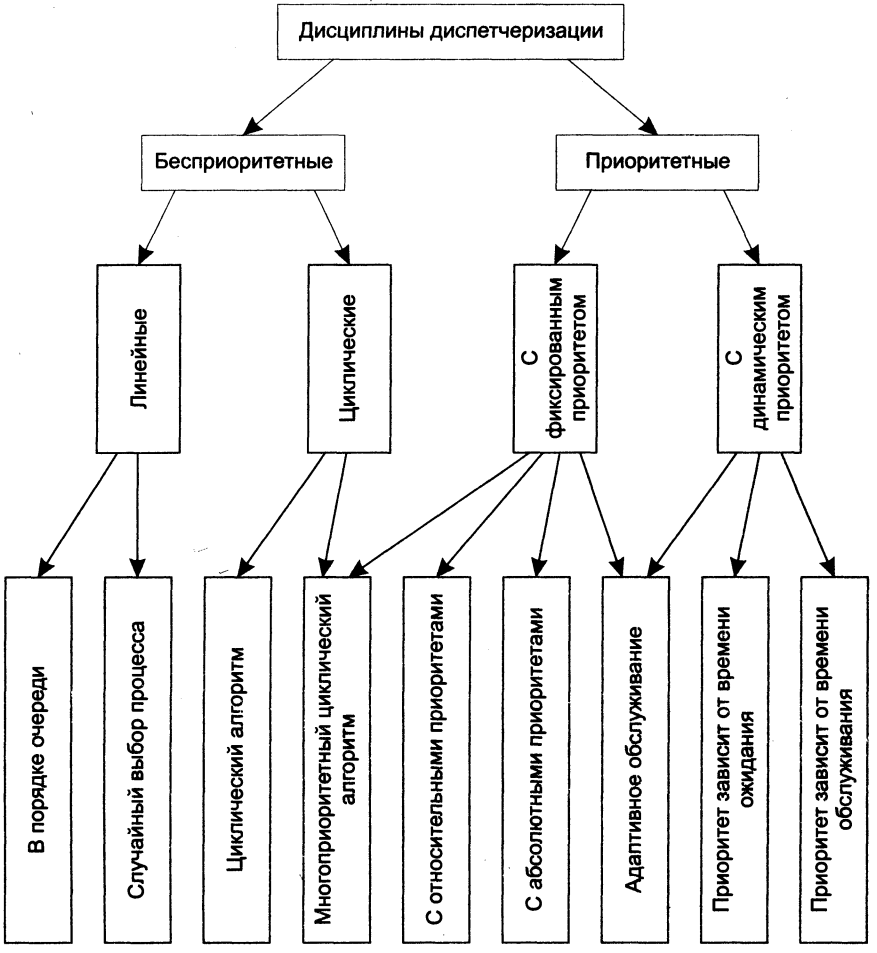


Рис. 2.1. Дисциплины диспетчеризации

Диспетчеризация с динамическими приоритетами требует дополнительных расходов на вычисление значений приоритетов исполняющихся задач, поэтому во многих операционных системах реального времени используются методы диспетчеризации на основе *абсолютных приоритетов*. Это позволяет сократить время реакции системы на очередное событие, однако требует детального анализа всей системы для правильного присвоения соответствующих приоритетов всем исполняющимся задачам с тем, чтобы гарантировать обслуживание. Проблему гарантии обслуживания мы рассмотрим ниже.

Рассмотрим некоторые основные (наиболее часто используемые) дисциплины диспетчеризации.

Самой простой в реализации является дисциплина *FCFS* (First Come First Served — первым пришел, первым обслужен), согласно которой задачи обслуживаются «в порядке очереди», то есть в порядке их появления. Те задачи, которые были заблоки-

рованы в процессе работы (попали в какое-либо из состояний ожидания, например из-за операций ввода-вывода), после перехода в состояние готовности вновь ставятся в эту очередь готовности. При этом возможны два варианта. Первый (самый простой) — это ставить разблокированную задачу в конец очереди готовых к выполнению задач. Этот вариант применяется чаще всего. Второй вариант заключается в том, что диспетчер помещает разблокированную задачу перед теми задачами, которые еще не выполнялись. Другими словами, в этом случае образуется две очереди (рис. 2.2): одна очередь образуется из новых задач, а вторая очередь — из ранее выполнявшихся, но попавших в состояние ожидания. Такой подход позволяет реализовать стратегию обслуживания «по возможности заканчивать вычисления в порядке их появления». Эта дисциплина обслуживания не требует внешнего вмешательства в ход вычислений, при ней не происходит перераспределения процессорного времени. Про нее можно сказать, что она относится к не вытесняющим дисциплинам[1].

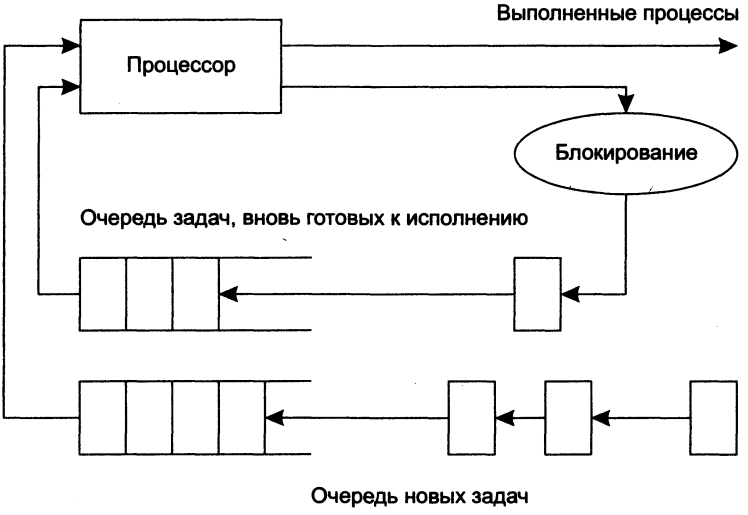


Рис. 2.2. Дисциплина диспетчеризации FCFS

К достоинствам этой дисциплины прежде всего можно отнести простоту реализации и малые расходы системных ресурсов на формирование очереди задач.

Однако эта дисциплина приводит к тому, что при увеличении загрузки вычислительной системы растет и среднее время ожидания обслуживания, причем короткие задания (требующие небольших затрат машинного времени) вынуждены ожидать

[1] Существующие дисциплины диспетчеризации процессов могут быть разбиты на два класса: вытесняющие (preemptive) и не вытесняющие (non-preemptive). В первых пакетных операционных системах часто реализовывали параллельное выполнение заданий без принудительного перераспределения процессора между задачами. В большинстве современных ОС для мощных вычислительных систем, а также в ОС для персональных компьютеров, ориентированных на высокопроизводительное выполнение приложений (Windows 9x/NT/2000/XP, Linux, OS/2), реализованы вытесняющие дисциплины диспетчеризации (вытесняющая многозадачность).

столько же, сколько трудоемкие задания. Избежать этого недостатка позволяют дисциплины SJN и SRT. Правило FCFS применяется и в более сложных дисциплинах диспетчеризации. Например, в приоритетных дисциплинах диспетчеризации, если имеется несколько задач с одинаковым приоритетом, которые стоят в очереди готовых к выполнению задач, то попадают они в эту очередь с учетом времени.

Дисциплина обслуживания *SJN* (Shortest Job Next — следующим выполняется самое короткое задание) требует, чтобы для каждого задания была известна оценка в потребностях машинного времени. Необходимость сообщать операционной системе характеристики задач с описанием потребностей в ресурсах вычислительной системы привела к тому, что были разработаны соответствующие языковые средства. В частности, ныне уже забытый *язык JCL* (Job Control Language — язык управления заданиями) был одним из наиболее известных. Пользователи вынуждены были указывать предполагаемое время выполнения задачи и для того, чтобы они не злоупотребляли возможностью указать заведомо меньшее время выполнения (с целью возможности получить результаты раньше других), ввели подсчет реальных потребностей. Диспетчер задач сравнивал заказанное время и время выполнения и в случае превышения указанной оценки потребности в данном ресурсе ставил данное задание не в начало, а в конец очереди. Еще в некоторых операционных системах в таких случаях использовалась система штрафов, при которой в случае превышения заказанного машинного времени оплата вычислительных ресурсов осуществлялась уже по другим расценкам.

Дисциплина обслуживания SJN предполагает, что имеется только одна очередь заданий, готовых к выполнению. Задания, которые в процессе своего исполнения были временно заблокированы (например, ожидали завершения операций ввода-вывода), вновь попадали в конец очереди готовых к выполнению наравне с вновь поступающими. Это приводило к тому, что задания, которым требовалось очень немного времени для своего завершения, вынуждены были ожидать процессор наравне с длительными работами, что не всегда хорошо.

Для устранения этого недостатка и была предложена дисциплина *SRT* (Shortest Remaining Time) — следующим будет выполняться задание, которому осталось меньше всего выполняться на процессоре.

Все эти три дисциплины обслуживания могут использоваться для пакетных режимов обработки, когда пользователю не нужно ждать реакции системы — он просто сдает свое задание и через несколько часов получает результаты вычислений. Для интерактивных же вычислений желательно прежде всего обеспечить приемлемое время реакции системы. Если же система является мультитерминальной, то помимо малого времени реакции системы на запрос пользователя желательно, чтобы она обеспечивала и равенство в обслуживании. Можно сказать, что стратегия обслуживания, согласно которой главным является равенство обслуживания при приемлемом времени обслуживания, является главной для систем разделения времени. Кстати, UNIX-системы реализуют дисциплины обслуживания, соответствующие именно этой стратегии.

Если же это однопользовательская система, но с возможностью мультипрограммной обработки, то желательно, чтобы те программы, с которыми непосредственно

работает пользователь, имели лучшее время реакции, нежели фоновые задания. При этом желательно, чтобы некоторые приложения, выполняясь без непосредственного участия пользователя (например, программа получения электронной почты, использующая модем и коммутируемые линии для передачи данных), тем не менее, гарантированно получали необходимую им долю процессорного времени.

Для решения перечисленных проблем используется дисциплина обслуживания, называемая *карусельной* (Round Robin, RR), и приоритетные методы обслуживания.

Дисциплина обслуживания RR предполагает, что каждая задача получает процессорное время порциями или, как говорят, *квантами времени* (time slice) q. После окончания кванта времени q задача снимается с процессора, и он передается следующей задаче. Снятая задача ставится в конец очереди задач, готовых к выполнению. Эту дисциплину обслуживания иллюстрирует рис. 2.3. Для оптимальной работы системы необходимо правильно выбрать закон, по которому кванты времени выделяются задачам.

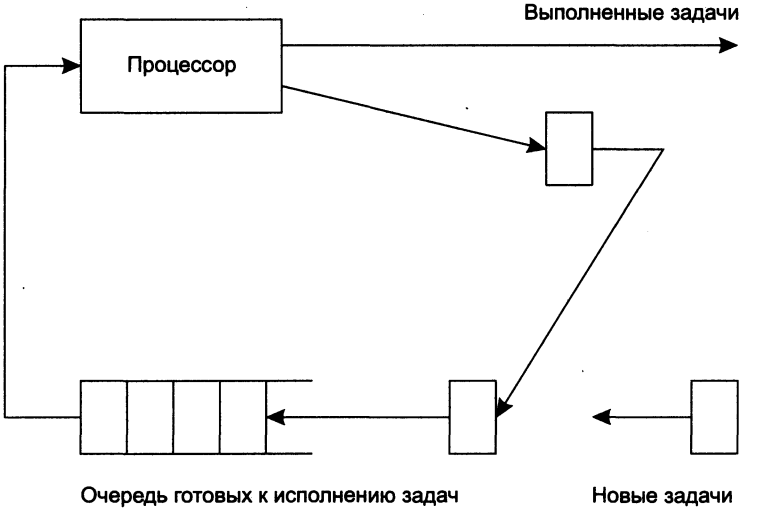


Рис. 2.3. Карусельная дисциплина диспетчеризации

Величина кванта времени q выбирается как компромисс между приемлемым временем реакции системы на запросы пользователей (с тем, чтобы их простейшие запросы не вызывали длительного ожидания) и накладными расходами на частую смену контекста задач. Очевидно, что при прерываниях операционная система вынуждена выполнять большой объем работы, связанной со сменой контекста. Она должна сохранить достаточно большой объем информации о текущем (прерываемом) процессе, поставить дескриптор снятой задачи в очередь, занести в рабочие регистры процессора соответствующие значения для той задачи, которая теперь будет выполняться (ее дескриптор расположен первым в очереди готовых к исполнению задач). Если величина q велика, то при увеличении очереди готовых к выполнению задач реакция системы станет медленной. Если же величина q мала, то относительная доля

накладных расходов на переключения контекста между исполняющимися задачами увеличится, и это ухудшит производительность системы.

В некоторых операционных системах есть возможность указывать в явном виде величину кванта времени или диапазон возможных значений, тогда система будет стараться выбирать оптимальное значение сама. Например, в операционной системе OS/2 в файле CONFIG.SYS есть возможность с помощью оператора TIMESLICE указать минимальное и максимальное значения для кванта q. Так, например, строка TIMESLICE=32,256 указывает, что минимальное значение равно 32 мс, а максимальное — 256. Если некоторая задача прервана, поскольку израсходован выделенный ей квант времени q, то следующий выделенный ей интервал будет увеличен на время, равное одному периоду таймера (около 32 мс), и так до тех пор, пока квант времени не станет равным максимальному значению, указанному в операторе TIMESLICE. Этот метод позволяет OS/2 уменьшить накладные расходы на переключение задач в том случае, если несколько задач параллельно выполняют длительные вычисления. Следует заметить, что диспетчеризация задач в этой операционной системе реализована, пожалуй, наилучшим образом с точки зрения реакции системы и эффективности использования процессорного времени.

Дисциплина карусельной диспетчеризации более всего подходит для случая, когда все задачи имеют одинаковые права на использование ресурсов центрального процессора. Однако как мы знаем, равенства в жизни гораздо меньше, чем неравенства. Одни задачи всегда нужно решать в первую очередь, тогда как остальные могут подождать. Это можно реализовать за счет того, что одной задаче мы (или диспетчер задач) присваиваем один приоритет, а другой задаче — другой. Задачи в очереди будут располагаться в соответствии с их приоритетами. Формирует очередь диспетчер задач. Процессор в первую очередь будет предоставляться задаче с самым высоким приоритетом, и только если ее потребности в процессоре удовлетворены или она попала в состояние ожидания некоторого события, диспетчер может предоставить его следующей задаче. Многие дисциплины диспетчеризации по-разному используют основную идею карусельной диспетчеризации и механизм приоритетов.

Дисциплина диспетчеризации RR — это одна из самых распространенных дисциплин. Однако бывают ситуации, когда операционная система не поддерживает в явном виде дисциплину карусельной диспетчеризации. Например, в некоторых операционных системах реального времени используется диспетчер задач, работающий по принципу абсолютных приоритетов (процессор предоставляется задаче с максимальным приоритетом, а при равенстве приоритетов он действует по принципу очередности) [7, 11]. Другими словами, причиной снятия задачи с выполнения может быть только появление задачи с более высоким приоритетом. Поэтому если нужно организовать обслуживание задач таким образом, чтобы все они получали процессорное время равномерно и равноправно, то системный оператор может сам организовать эту дисциплину. Для этого достаточно всем пользовательским задачам присвоить одинаковые приоритеты и создать одну высокоприоритетную задачу, которая не должна ничего делать, но которая, тем не менее, будет по таймеру (через указанные интервалы времени) планироваться на выполнение. Благодаря высокому приоритету этой задачи текущее приложение будет сниматься с выполнения и ставиться в конец очереди, а поскольку этой высокоприоритетной задаче

на самом деле ничего делать не надо, то она тут же освободит процессор, и из очереди готовности будет взята следующая задача.

В своей простейшей реализации дисциплина карусельной диспетчеризации предполагает, что все задачи имеют одинаковый приоритет. Если же необходимо ввести механизм приоритетного обслуживания, то это, как правило, делается за счет *организации нескольких очередей*. Процессорное время предоставляется в первую очередь тем задачам, которые стоят в самой привилегированной очереди. Если она пустая, то диспетчер задач начинает просматривать остальные очереди. Именно по такому алгоритму действует диспетчер задач в операционных системах OS/2, Windows 9x, Windows NT/2000/XP и многих других. Отличия в основном заключаются в количестве очередей и в правилах, касающихся перемещения задач из одной очереди в другую.

Известные дисциплины диспетчеризации (мы здесь рассмотрели только основные) могут применять или не применять еще одно правило, касающееся перераспределения процессора между выполняющимися задачами.

Есть дисциплины, в которых процессор принудительно может быть отобран у текущей задачи. Такие дисциплины обслуживания называют *вытесняющими*, поскольку одна задача вытесняется другой. Другими словами, возможно принудительное перераспределение процессорного времени между выполняющимися задачами. Оно осуществляется самой операционной системой, отбирающей периодически процессор у выполняющейся задачи.

А есть дисциплины диспетчеризации, в которых ничто не может отобрать у задачи процессор, пока она сама его не освободит. Освобождение процессора в этом случае, как правило, связано с тем, что задача попадает в состояние ожидания некоторого события.

Итак, диспетчеризация без перераспределения процессорного времени, то есть *не вытесняющая* (non-preemptive multitasking), или *кооперативная, многозадачность* (cooperative multitasking), — это такой способ диспетчеризации задач, при котором активная задача выполняется до тех пор, пока она сама, что называется «по собственной инициативе», не отдаст управление диспетчеру задач для того, чтобы тот выбрал из очереди другой, готовый к выполнению процесс или поток. Дисциплины обслуживания FCFS, SJN, SRT относятся к не вытесняющим.

Диспетчеризация с перераспределением процессорного времени между задачами, то есть *вытесняющая многозадачность* (preemptive multitasking), — это такой способ, при котором решение о переключении процессора с выполнения одной задачи на выполнение другой принимается диспетчером задач, а не самой активной задачей. При вытесняющей многозадачности механизм диспетчеризации задач целиком сосредоточен в операционной системе, и программист может писать свое приложение, не заботясь о том, как оно будет выполняться параллельно с другими задачами (процессами и потоками). При этом операционная система выполняет следующие функции: определяет момент снятия с выполнения текущей задачи, сохраняет ее контекст в дескрипторе задачи, выбирает из очереди готовых задач следующую и запускает ее на выполнение, загружая ее контекст. Дисциплина RR и многие другие, построенные на ее основе, относятся к вытесняющим.

При не вытесняющей многозадачности процессорное время распределено между системой и прикладными программами. Прикладная программа, получив управление от операционной системы, сама должна определить момент завершения своей очередной итерации и передачи управления супервизору операционной системы с помощью соответствующего системного вызова. При этом естественно, что диспетчер задач, так же как и в случае вытесняющей мультизадачности, формирует очереди задач и выбирает в соответствии с некоторым алгоритмом (например, с учетом порядка поступления задач или их приоритетов) следующую задачу на выполнение. Такой механизм создает некоторые проблемы как для пользователей, так и для разработчиков.

Для пользователей это означает, что управление системой может теряться на некоторый произвольный период времени, который определяется процессом выполнения приложения (а не системой, старающейся всегда обеспечить приемлемое время реакции на запросы пользователей) [27]. Если приложение тратит слишком много времени на выполнение какой-либо работы (например, на форматирование диска), пользователь не может переключиться с этой на другую задачу (например, на текстовый или графический редактор, в то время как форматирование продолжалось бы в фоновом режиме). Эта ситуация нежелательна, так как пользователи обычно не хотят долго ждать, когда машина завершит свою задачу.

Поэтому разработчики приложений для не вытесняющей операционной среды, возлагая на себя функции диспетчера задач, должны создавать приложения так, чтобы они выполняли свои задачи небольшими частями. Так, упомянутая выше программа форматирования может отформатировать одну дорожку дискеты и вернуть управление системе. После выполнения других задач система возвратит управление программе форматирования, чтобы та отформатировала следующую дорожку. Подобный метод разделения времени между задачами работает, но он существенно затрудняет разработку программ и предъявляет повышенные требования к квалификации программиста.

Например, в ныне уже забытой операционной среде Windows 3.x нативные 16-разрядные приложения этой системы разделяли между собой процессорное время именно таким образом. И именно программисты должны были обеспечивать «дружественное» отношение своей программы к другим выполняемым одновременно с ней программам, достаточно часто отдавая управление ядру системы. Крайним проявлением «недружественности» приложения является его зависание, приводящее к общему краху системы — прекращению всех вычислений. В системах с вытесняющей многозадачностью такие ситуации, как правило, исключены, так как центральный механизм диспетчеризации, во-первых, обеспечивает все задачи процессорным временем, во-вторых, дает возможность иметь надежные механизмы мониторинга вычислений и, в-третьих, позволяет снять зависшую задачу с выполнения.

Однако распределение функций диспетчеризации между системой и приложениями не всегда является недостатком, а при определенных условиях может быть и достоинством, потому что дает возможность разработчику приложений самому планировать распределение процессорного времени наиболее подходящим для

данного фиксированного набора задач образом [27, 44, 46]. Так как разработчик сам определяет в программе момент времени передачи управления, то при этом исключаются нерациональные прерывания программ в «неудобные» для них моменты времени. Кроме того, легко разрешаются проблемы совместного использования данных: задача во время каждой итерации использует их монопольно и уверена, что на протяжении этого периода никто другой их не изменит. Примером эффективного применения не вытесняющей многозадачности является сетевая операционная система Novell NetWare, в которой в значительной степени благодаря этому достигнута высокая скорость выполнения файловых операций. Менее удачным оказалось использование не вытесняющей многозадачности в операционной среде Windows 3.x. К счастью, на сегодня эта операционная система уже нигде не применяется, ее с успехом заменила сначала Windows 95, а затем и Windows 98. Правда, следует заметить, что при выполнении в этих операционных системах старых 16-разрядных приложений, разработанных в свое время для операционной среды Win16 API, создается виртуальная машина, работающая по принципам не вытесняющей многозадачности.

Качество диспетчеризации и гарантии обслуживания

Одна из проблем, которая возникает при выборе подходящей дисциплины обслуживания — это *гарантия обслуживания*. Дело в том, что в некоторых дисциплинах, например в дисциплине абсолютных приоритетов, низкоприоритетные процессы получаются обделенными многими ресурсами и, прежде всего, процессорным временем. Возникает реальная дискриминация низкоприоритетных задач, в результате чего они достаточно длительное время могут не получать процессорное время. В конце концов, некоторые процессы и задачи вообще могут быть не выполнены к заданному сроку. Известны случаи, когда вследствие высокой загрузки вычислительной системы отдельные процессы вообще не выполнились, несмотря на то что прошло несколько лет (!) с момента их планирования. Поэтому вопрос гарантии обслуживания является очень актуальным.

Более жестким требованием к системе, чем просто гарантированное завершение процесса, является его гарантированное завершение к указанному моменту времени или за указанный интервал времени. Существуют различные дисциплины диспетчеризации, учитывающие жесткие временные ограничения, но не существует дисциплин, которые могли бы предоставить больше процессорного времени, чем может быть в принципе выделено.

Планирование с учетом жестких временных ограничений легко реализовать, организуя очередь готовых к выполнению задач в порядке возрастания их временных ограничений. Основным недостатком такого простого упорядочения является то, что задача (за счет других задач) может быть обслужена быстрее, чем это ей реально необходимо. Чтобы избежать этого, проще всего процессорное время выделять все-таки квантами. А после получения задачей своего кванта времени операционная система, оценив некоторое множество факторов (важных с точки зрения опти-

мизации распределения процессорного времени и гарантий обслуживания к заданному сроку), может переназначить приоритет задаче. Это позволит ей более гибко использовать механизм приоритетов и иметь механизмы гарантии обслуживания.

Гарантировать обслуживание можно, например, следующими тремя способами.

- Выделять минимальную долю процессорного времени некоторому классу процессов, если по крайней мере один из них готов к исполнению. Например, можно отводить 20 % от каждых 10 мс процессам реального времени, 40 % от каждых 2 с — интерактивным процессам и 10 % от каждых 5 мин — пакетным (фоновым) процессам.
- Выделять минимальную долю процессорного времени некоторому конкретному процессу, если он готов к выполнению.
- Выделять столько процессорного времени некоторому процессу, чтобы он мог выполнить свои вычисления к сроку.

Для сравнения алгоритмов диспетчеризации обычно используются некоторые критерии.

- *Загрузка центрального процессора* (CPU utilization). В большинстве персональных систем средняя загрузка процессора не превышает 2–3 %, доходя в моменты выполнения сложных вычислений и до 100 %. В реальных системах, где компьютеры (например, серверы) выполняют очень много работы, загрузка процессора колеблется в пределах от 15–40 % (для легко загруженного процессора) до 90–100 % (для тяжело загруженного процессора).
- *Пропускная способность центрального процессора* (CPU throughput). Пропускная способность процессора может измеряться количеством процессов, которые выполняются в единицу времени.
- *Время оборота* (turnaround time). Для некоторых процессов важным критерием является полное время выполнения, то есть интервал от момента появления процесса во входной очереди до момента его завершения. Это время названо временем оборота и включает время ожидания во входной очереди, время ожидания в очереди готовых процессов, время ожидания в очередях к оборудованию, время выполнения в процессоре и время ввода-вывода.
- *Время ожидания* (waiting time). Под временем ожидания понимается суммарное время нахождения процесса в очереди готовых процессов.
- *Время отклика* (response time). Для интерактивных программ важным показателем является время отклика, или время, прошедшее от момента попадания процесса во входную очередь до момента первого обращения к терминалу.

Очевидно, что простейшая стратегия краткосрочного планировщика должна быть направлена на максимизацию средних значений загруженности и пропускной способности, времени ожидания и времени отклика.

Правильное планирование процессов в значительной степени влияет на производительность всей системы. Можно выделить следующие главные причины, приводящие к снижению производительности системы.

- Накладные расходы на переключение процессора. Они определяются не только переключениями контекстов задач, но и (при переключении на потоки другого приложения) перемещениями страниц виртуальной памяти, а также необходимостью обновления данных в кэше (коды и данные одной задачи, находящиеся в кэше, не нужны другой задаче и будут заменены, что приведет к дополнительным задержкам).
- Переключение на другую задачу в тот момент, когда текущая задача выполняет критическую секцию, а другие задачи активно ожидают входа в свою критическую секцию (см. главу 7). В этом случае потери будут особо велики (хотя вероятность прерывания выполнения коротких критических секций мала).

В случае мультипроцессорных систем применяются следующие методы повышения производительности системы:

- совместное планирование, при котором все потоки одного приложения (неблокированные) одновременно ставятся на выполнение процессорами и одновременно снимаются с выполнения (для сокращения переключений контекста);
- планирование, при котором находящиеся в критической секции задачи не прерываются, а активно ожидающие входа в критическую секцию задачи не ставятся на выполнение до тех пор, пока вход в секцию не освободится;
- планирование с учетом так называемых *подсказок* (hints) программы (во время ее выполнения), например, в известной своими новациями ОС Mach имелось два класса таких подсказок: во-первых, указания (разной степени категоричности) о снятии текущего процесса с процессора, во-вторых, указания о том процессе, который должен быть выбран взамен текущего.

Одним из основных методов гарантии обслуживания является использование динамических приоритетов.

Диспетчеризация задач с использованием динамических приоритетов

При выполнении программ, реализующих какие-нибудь задачи контроля и управления (что характерно, прежде всего, для систем реального времени), может случиться такая ситуация, когда одна или несколько задач не могут быть реализованы (решены) в течение длительного промежутка времени из-за возросшей нагрузки в вычислительной системе. Потери, связанные с невыполнением таких задач, могут оказаться больше, чем потери от невыполнения программ с более высоким приоритетом. При этом оказывается целесообразным временно изменить приоритет «аварийных» задач (для которых истекает отпущенное для них время обработки). После выполнения этих задач их приоритет восстанавливается. Поэтому почти в любой операционной системе реального времени (ОС РВ) имеются средства для *динамического изменения приоритета* (dynamic priority variation) задачи. Есть такие средства и во многих операционных системах, которые не относятся к классу ОС РВ.

Рассмотрим, например, как реализован механизм динамических приоритетов в операционной системе UNIX, которая, как известно, не относится к ОС РВ. Операцион-

ные системы класса UNIX относятся к мультитерминальным диалоговым системам. Основная стратегия обслуживания, применяемая в UNIX-системах, — это равенство в обслуживании и обеспечение приемлемого времени реакции системы. Реализуется эта стратегия за счет дисциплины диспетчеризации RR с несколькими очередями и механизма динамических приоритетов. Приоритет процесса вычисляется следующим образом [39]. Во-первых, в вычислении участвуют значения двух полей дескриптора процесса — p_nice и p_cpu. Первое из них назначается пользователем явно или формируется по умолчанию с помощью системы программирования. Второе поле формируется диспетчером задач (планировщиком разделения времени) и называется системной составляющей или текущим приоритетом. Другими словами, каждый процесс имеет два атрибута приоритета. С учетом этого приоритета и распределяется между исполняющимися задачами процессорное время: *текущий приоритет*, на основании которого происходит планирование, и заказанный *относительный приоритет* (называемый *nice number*, или просто *nice*).

Схема нумерации текущих приоритетов различна для различных версий UNIX. Например, более высокому значению текущего приоритета может соответствовать более низкий фактический приоритет планирования. Разделение между приоритетами режима ядра и задачи также зависит от версии. Рассмотрим частный случай, когда текущий приоритет процесса варьируется в диапазоне от 0 (низкий приоритет) до 127 (наивысший приоритет). Процессы, выполняющиеся в режиме задачи, имеют более низкий приоритет, чем в режиме ядра. Для режима задачи приоритет меняется в диапазоне 0–65, для режима ядра — 66–95 (системный диапазон). Процессы, приоритеты которых лежат в диапазоне 96–127, являются процессами с фиксированным приоритетом, не изменяемым операционной системой, и предназначены для поддержки приложений реального времени.

Процессу, ожидающему недоступного в данный момент ресурса, система определяет значение *приоритета сна*, выбираемое ядром из диапазона системных приоритетов и связанное с событием, вызвавшим это состояние. Когда процесс пробуждается, ядро устанавливает значение текущего приоритета процесса равным приоритету сна. Поскольку приоритет такого процесса находится в системном диапазоне и выше, чем приоритет режима задачи, вероятность предоставления процессу вычислительных ресурсов весьма велика. Такой подход позволяет, в частности, быстро завершить системный вызов, в ходе выполнения которого могут блокироваться некоторые системные ресурсы.

После завершения системного вызова перед возвращением в режим задачи ядро восстанавливает приоритет режима задачи, сохраненный перед выполнением системного вызова. Это может привести к понижению приоритета, что, в свою очередь, вызовет переключение контекста.

Текущий приоритет процесса в режиме задачи p_priuser, как мы только что отмечали, зависит от значения относительного приоритета p_nice и степени использования вычислительных ресурсов p_cpu:

```
p_priuser = a × p_nice - b × p_cpu
```

Задача планировщика разделения времени — справедливо распределить вычислительный ресурс между конкурирующими процессами. Для принятия решения о

выборе следующего запускаемого процесса планировщику необходима информация об использовании процессора. Эта составляющая приоритета уменьшается обработчиком прерываний таймера каждый тик. Таким образом, пока процесс выполняется в режиме задачи, его текущий приоритет линейно уменьшается.

Каждую секунду ядро пересчитывает текущие приоритеты процессов, готовых к запуску (приоритеты которых меньше некоторого порогового значения; в нашем примере эта величина равна 65), последовательно увеличивая их за счет последовательного уменьшения отрицательного компонента времени использования процессора. Как результат, эти действия приводят к перемещению процессов в более приоритетные очереди и повышению вероятности их последующего выполнения.

Возможно использование следующей формулы:

```
p_cpu = p_cpu/2
```

В этом правиле проявляется недостаток нивелирования приоритетов при повышении загрузки системы. Происходит это потому, что в таком случае каждый процесс получает незначительный объем вычислительных ресурсов и, следовательно, имеет малую составляющую p_cpu, которая еще более уменьшается благодаря формуле пересчета величины p_cpu. В результате загрузка процессора перестает оказывать заметное влияние на приоритет, и низкоприоритетные процессы (то есть процессы с высоким значением nice number) практически «отлучаются» от вычислительных ресурсов системы.

В некоторых версиях UNIX для пересчета значения p_cpu используется другая формула:

```
p_cpu = p_cpu × (2 × load)/(2 × load + 1)
```

Здесь параметр load равен среднему числу процессов, находившихся в очереди на выполнение за последнюю секунду, и характеризует среднюю загрузку системы за этот период времени. Этот алгоритм позволяет частично избавиться от недостатка планирования по формуле p_cpu = p_cpu/2, поскольку при значительной загрузке системы уменьшение p_cpu при пересчете будет происходить медленнее.

Описанные алгоритмы диспетчеризации позволяют учесть интересы низкоприоритетных процессов, так как в результате длительного ожидания очереди на запуск приоритет таких процессов увеличивается, соответственно повышается и вероятность их запуска. Эти алгоритмы также обеспечивают более вероятный выбор планировщиком интерактивных процессов по отношению к сугубо вычислительным (фоновым). Такие задачи, как командный интерпретатор или редактор, большую часть времени проводят в ожидании ввода, имея, таким образом, высокий приоритет (приоритет сна). При наступлении ожидаемого события (например, пользователь осуществил ввод данных) им сразу же предоставляются вычислительные ресурсы. Фоновые процессы, потребляющие значительные ресурсы процессора, имеют высокую составляющую p_cpu и, как следствие, более низкий приоритет.

Аналогичные механизмы имеют место и в таких операционных системах, как OS/2 или Windows NT/2000/XP. Правда, алгоритмы изменения приоритета задач в этих системах иные. Например, в Windows NT/2000/XP каждый поток выполнения имеет базовый уровень приоритета, который лежит в диапазоне от двух уровней

ниже базового приоритета процесса, его породившего, до двух уровней выше этого приоритета, как показано на рис. 2.4. Базовый приоритет процесса определяет, сколь сильно могут различаться приоритеты потоков этого процесса и как они соотносятся с приоритетами потоков других процессов. Поток наследует этот базовый приоритет и может изменять его так, чтобы он стал немного больше или немного меньше. В результате получается приоритет планирования, с которым поток и начинает исполняться. В процессе исполнения потока его приоритет может отклоняться от базового.

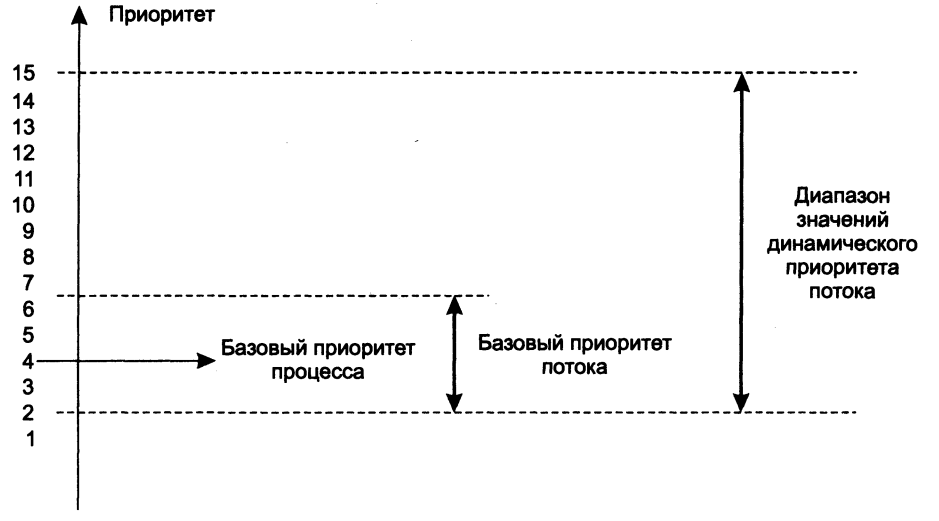


Рис. 2.4. Схема динамического изменения приоритетов в Windows NT/2000/XP

На рисунке также показан динамический приоритет потока, нижней границей которого является базовый приоритет потока, а верхняя зависит от вида работ, исполняемых потоком. Например, если поток обрабатывает текущие результаты операций ввода пользователем своих данных, диспетчер задач Windows поднимает его динамический приоритет; если же он выполняет вычисления, то диспетчер задач постепенно снижает его приоритет до базового. Снижая приоритет одной задачи и поднимая приоритет другой, подсистемы могут управлять относительной приоритетностью потоков внутри процесса.

Для определения порядка выполнения потоков диспетчер задач использует систему приоритетов, направляя на выполнение задачи с высоким приоритетом раньше задач с низким приоритетом. Система прекращает исполнение, или *вытесняет* (preempts), текущий поток, если становится готовым к выполнению другой поток с более высоким приоритетом.

Имеется группа очередей — по одной для каждого приоритета. В операционных системах Windows NT/2000/XP используется один и тот же диспетчер задач. Он поддерживает 32 уровня приоритета. Задачи делятся на два класса: реального времени и переменного приоритета. Задачи реального времени, имеющие приоритеты от 16 до 31, — это высокоприоритетные потоки, используемые программами,

критическими по времени выполнения, то есть требующими немедленного внимания системы (по терминологии Microsoft).

Диспетчер задач просматривает очереди, начиная с самой приоритетной. При этом если очередь пустая, то есть в ней нет готовых к выполнению задач с таким приоритетом, то осуществляется переход к следующей очереди. Следовательно, если есть задачи, требующие процессор немедленно, они будут обслужены в первую очередь. Для собственно системных модулей, функционирующих в статусе задачи, зарезервирована очередь с номером 0.

Большинство задач в системе относятся к классу переменного приоритета с уровнями приоритета (номером очереди) от 1 до 15. Эти очереди используются задачами с *переменным приоритетом* (variable priority), так как диспетчер задач для оптимизации отклика системы корректирует их приоритеты по мере выполнения. Диспетчер приостанавливает исполнение текущей задачи, после того как та израсходует свой квант времени. При этом если прерванная задача — это поток переменного приоритета, то диспетчер задач понижает приоритет этого потока выполнения на единицу и перемещает в другую очередь. Таким образом, приоритет задачи, выполняющей много вычислений, постепенно понижается (до значения его базового приоритета). С другой стороны, диспетчер повышает приоритет задачи после ее освобождения из состояния ожидания. Обычно добавка к приоритету задачи определяется кодом исполнительной системы, находящимся вне ядра операционной системы, однако величина этой добавки зависит от типа события, которого ожидала заблокированная задача. Так, например, поток, ожидавший ввода очередного байта с клавиатуры, получает большую добавку к значению своего приоритета, чем поток ввода-вывода, работавший с дисковым накопителем. Однако в любом случае значение приоритета не может достигнуть 16.

В операционной системе OS/2 схема динамической приоритетной диспетчеризации несколько иная, хоть и похожа[1]. В OS/2 также имеется четыре класса задач. И для каждого класса задач имеется своя группа приоритетов с интервалом значений от 0 до 31. Итого, 128 различных уровней и, соответственно, 128 возможных очередей готовых к выполнению задач (потоков).

Задачи, имеющие самые высокие значения приоритета, называются *критическими по времени* (time critical). В этот класс входят задачи, которые мы в обиходе называем *задачами реального времени*, то есть для них должен быть обязательно предоставлен определенный минимум процессорного времени. Наиболее часто встречающимися задачами этого класса являются задачи коммуникаций (например, задача управления последовательным портом, на который приходят биты по коммутируемой линии с подключенным модемом, или задачи управления сетевым оборудованием). Если такие задачи не получат управление в нужный момент времени, то сеанс связи может прерваться.

[1] Как известно, одно время компания Microsoft принимала активное участие в разработке OS/2 совместно с IBM. Поэтому после прекращения совместных работ над этой операционной системой и начале нового проекта многие решения из OS/2 были унаследованы в варианте OS/2 ver. 3.0, названной впоследствии Windows NT.

Следующий класс задач имеет название *приоритетного*. Поскольку к этому классу относят задачи, которые выполняют по отношению к остальным задачам функции сервера (о модели клиент-сервер, по которой строятся современные операционные системы с микроядерной архитектурой, см. главы 9 и 10), то его еще иногда называют *серверным*. Приоритет таких задач должен быть выше, поскольку это позволяет гарантировать, что запрос на некоторую функцию со стороны обычных задач выполнится сразу, а не будет дожидаться, пока до него дойдет очередь на фоне других пользовательских приложений.

Большинство задач относят к обычному классу, его еще называют *регулярным* (regular), или *стандартным*. По умолчанию система программирования порождает задачу, относящуюся именно к этому классу.

Наконец, существует еще класс фоновых задач, называемый в OS/2 *остаточным*. Программы этого класса получают процессорное время только тогда, когда нет задач из других классов, требующих процессор. В качестве примера такой задачи можно привести программу обновления индексного файла, используемого при поиске файлов, или программу проверки электронной почты.

Внутри каждого из вышеописанных классов задачи, имеющие одинаковый уровень приоритета, выполняются в соответствии с дисциплиной RR. Переход от одного потока к другому происходит либо по окончании отпущенного ему кванта времени, либо по системному прерыванию, передающему управление задаче с более высоким приоритетом (таким образом система вытесняет задачи с более низким приоритетом для выполнения задач с более высоким приоритетом и может обеспечить быструю реакцию на важные события).

OS/2 самостоятельно изменяет приоритет выполняющихся программ независимо от уровня, установленного самим приложением. Этот механизм называется *повышением приоритета* (priority boost). Операционная система изменяет приоритет задачи в трех случаях [26].

- Повышение приоритета активной задачи (foreground boost). Приоритет задачи автоматически повышается, когда она становится активной. Это снижает время реакции активного приложения на действия пользователя по сравнению с фоновыми программами.
- Повышение приоритета ввода-вывода (Input/Output boost). По завершении операции ввода-вывода задача получает самый высокий уровень приоритета ее класса. Таким образом обеспечивается завершение всех незаконченных операций ввода-вывода.
- Повышение приоритета «забытой» задачи (starvation boost). Если задача не получает управление в течение достаточно долгого времени (этот промежуток времени задает оператор `MAXWAIT` в файле `CONFIG.SYS`[1]), диспетчер задач OS/2 временно присваивает ей уровень приоритета, не превышающий критический. В результате переключение на такую «забытую» программу происходит быстрее. После выполнения приложения в течение одного кванта времени его при-

[1] Строка MAXWAIT = 1 означает, что приоритет задачи при переключении на нее будет поднят до максимального не позже чем через одну секунду.

оритет вновь снижается до остаточного. В сильно загруженных системах этот механизм позволяет программам с остаточным приоритетом работать хотя бы в краткие интервалы времени. В противном случае они вообще никогда бы не получили управление.

Если нет необходимости использовать метод динамического изменения приоритета, то с помощью оператора PRIOPITY = ABSOLUTE в файле CONFIG.SYS можно ввести дисциплину абсолютных приоритетов; по умолчанию оператор PRIOPITY имеет значение DYNAMIC.

Контрольные вопросы и задачи

1. Перечислите и поясните основные функции операционных систем, которые связаны с управлением задачами.
2. В чем заключается основное различие между планированием процессов и диспетчеризацией задач?
3. Что такое стратегия обслуживания? Перечислите известные вам стратегии обслуживания.
4. Какие дисциплины диспетчеризации задач вы знаете? Поясните их основные идеи, перечислите достоинства и недостатки.
5. Расскажите, какие дисциплины диспетчеризации следует отнести к вытесняющим, а какие — к не вытесняющим.
6. Как можно реализовать механизм разделения времени, если диспетчер задач работает только по принципу предоставления процессорного времени задаче с максимальным приоритетом?
7. Что такое «гарантия обслуживания»? Как ее можно реализовать?
8. Опишите механизм динамической диспетчеризации, реализованный в UNIX-системах.
9. Сравните механизмы диспетчеризации задач в операционных системах Windows NT и OS/2. В чем они похожи друг на друга и в чем заключаются основные различия?

Глава 3. Управление памятью в операционных системах

Оперативная память — это важнейший ресурс любой вычислительной системы, поскольку без нее (как, впрочем, и без центрального процессора) невозможно выполнение ни одной программы. В главе 1 мы уже отмечали, что память является разделяемым ресурсом. От выбранных механизмов распределения памяти между выполняющимися процессорами в значительной степени зависит эффективность использования ресурсов системы, ее производительность, а также возможности, которыми могут пользоваться программисты при создании своих программ. Желательно так распределять память, чтобы выполняющаяся задача имела возможность обратиться по любому адресу в пределах адресного пространства той программы, в которой идут вычисления. С другой стороны, поскольку любой процесс имеет потребности в операциях ввода-вывода, и процессор достаточно часто переключается с одной задачи на другую, желательно в оперативной памяти расположить достаточное количество активных задач с тем, чтобы процессор не останавливал вычисления из-за отсутствия очередной команды или операнда. Некоторые ресурсы, которые относятся к неразделяемым, из-за невозможности их совместного использования делают *виртуальными*. Таким образом, чтобы иметь возможность выполняться, каждый процесс может получить некий виртуальный ресурс. Виртуализация ресурсов делается программным способом средствами операционной системы, а значит, для них тоже нужно иметь ресурс памяти. Поэтому вопросы организации разделения памяти для выполняющихся процессов и потоков являются очень актуальными, ибо выбранные и реализованные алгоритмы решения этих вопросов в значительной степени определяют и потенциальные возможности системы, и общую ее производительность, и эффективность использования имеющихся ресурсов.

Память и отображения, виртуальное адресное пространство

Если не принимать во внимание программирование на машинном языке (эта технология практически не используется уже очень давно), то можно сказать, что программист обращается к памяти с помощью некоторого набора логических имен, которые чаще всего являются символьными, а не числовыми, и для которого отсутствует отношение порядка. Другими словами, в общем случае множество переменных в программе не упорядочено, хотя отдельные переменные могут иметь частичную упорядоченность (например, элементы массива). Имена переменных и входных точек программных модулей составляют пространство символьных имен. Иногда это адресное пространство называют *логическим*.

С другой стороны, при выполнении программы мы имеем дело с физической оперативной памятью, собственно с которой и работает процессор, извлекая из нее команды и данные и помещая в нее результаты вычислений. *Физическая память* представляет собой упорядоченное множество ячеек реально существующей оперативной памяти, и все они пронумерованы, то есть к каждой из них можно обратиться, указав ее порядковый номер (адрес). Количество ячеек физической памяти ограниченно и фиксированно.

Системное программное обеспечение должно связать каждое указанное пользователем символьное имя с физической ячейкой памяти, то есть осуществить отображение пространства имен на физическую память компьютера. В общем случае это отображение осуществляется в два этапа (рис. 3.1): сначала системой программирования, а затем операционной системой. Это второе отображение осуществляется с помощью соответствующих аппаратных средств процессора — подсистемы управления памятью, которая использует дополнительную информацию, подготавливаемую и обрабатываемую операционной системой. Между этими этапами обращения к памяти имеют форму *виртуального* адреса. При этом можно сказать, что множество всех допустимых значений виртуального адреса для некоторой программы определяет ее *виртуальное адресное пространство*, или *виртуальную память*. Виртуальное адресное пространство программы зависит, прежде всего, от архитектуры процессора и от системы программирования и практически не зависит от объема реальной физической памяти компьютера. Можно еще сказать, что адреса команд и переменных в машинной программе, подготовленной к выполнению системой программирования, как раз и являются виртуальными адресами.

Как мы знаем, система программирования осуществляет трансляцию и компоновку программы, используя библиотечные программные модули. В результате работы системы программирования полученные виртуальные адреса могут иметь как двоичную форму, так и символьно-двоичную. Это означает, что некоторые программные модули (их, как правило, большинство) и их переменные получают ка-

кие-то числовые значения, а те модули, адреса для которых пока не могут быть определены, имеют по-прежнему символьную форму, и их окончательная привязка к физическим ячейкам будет осуществлена на этапе загрузки программы в память перед ее непосредственным выполнением.

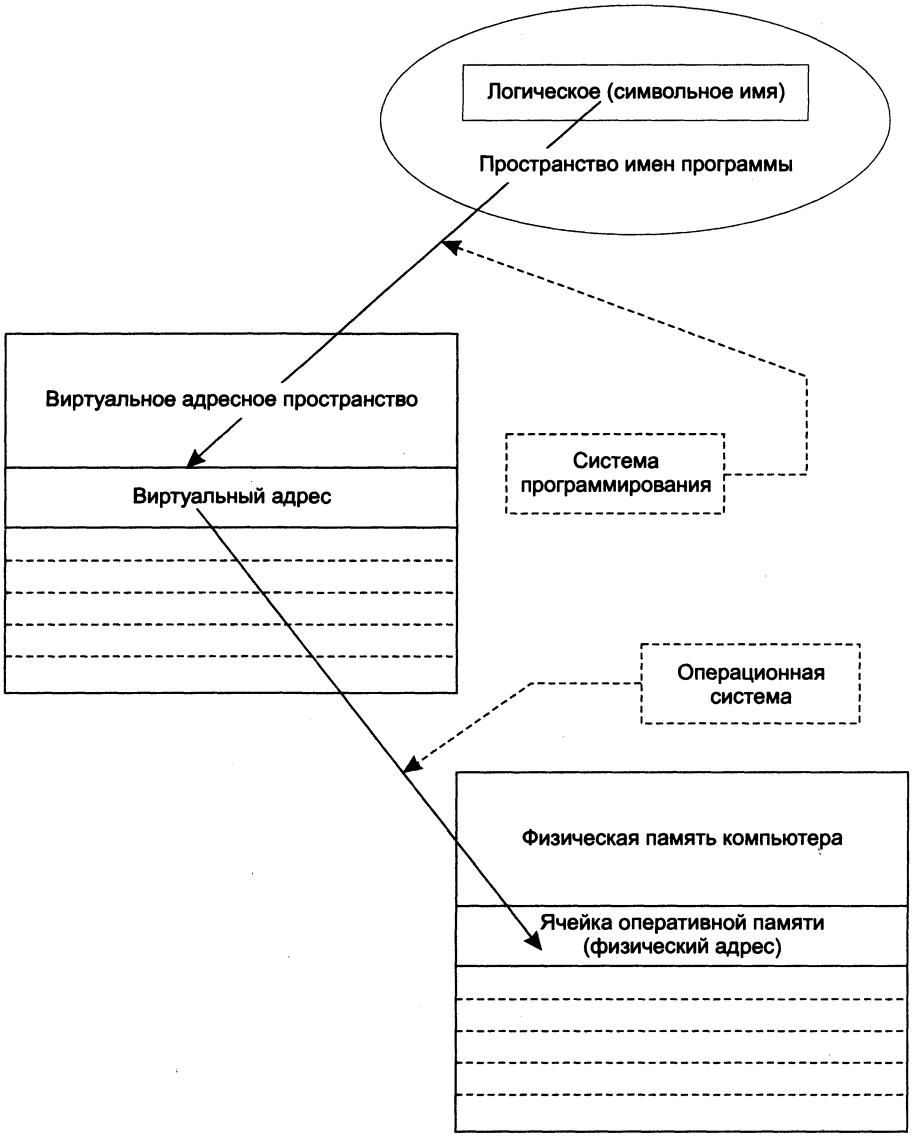


Рис. 3.1. Память и отображения

Одним из частных случаев отображения пространства символьных имен на физическую память является полная тождественность виртуального адресного пространства физической памяти. При этом нет необходимости осуществлять второе ото-

бражение. В таком случае говорят, что система программирования генерирует *абсолютную двоичную программу*; в этой программе все двоичные адреса таковы, что программа может исполняться только тогда, когда ее виртуальные адреса будут точно соответствовать физическим. Часть программных модулей любой операционной системы обязательно должна быть абсолютными двоичными программами. Эти программы размещаются по фиксированным адресам физической памяти, и с их помощью уже можно впоследствии реализовывать размещение остальных программ, подготовленных системой программирования таким образом, что они могут работать на различных физических адресах (то есть на тех адресах, на которые их разместит операционная система). В качестве примера таких программ можно назвать программы загрузки операционной системы.

Другим частным случаем этой общей схемы трансляции адресного пространства является тождественность виртуального адресного пространства исходному логическому пространству имен. Здесь уже отображение выполняется самой операционной системой, которая во время исполнения использует таблицу символьных имен. Такая схема отображения используется чрезвычайно редко, так как отображение имен на адреса необходимо выполнять для каждого вхождения имени (каждого нового имени), и особенно много времени тратится на квалификацию имен. Данную схему можно было встретить в интерпретаторах, в которых стадии трансляции и исполнения практически неразличимы. Это характерно для простейших компьютерных систем, в которых вместо операционной системы использовался встроенный интерпретатор (например, Basic).

Возможны и промежуточные варианты. В простейшем случае транслятор-компилятор генерирует относительные адреса, которые, по сути, являются виртуальными адресами, с последующей настройкой программы на один из непрерывных разделов. Второе отображение осуществляется перемещающим загрузчиком. После загрузки программы виртуальный адрес теряется, и доступ выполняется непосредственно к физическим ячейкам. Более эффективное решение достигается в том случае, когда транслятор вырабатывает в качестве виртуального адреса относительный адрес и информацию о начальном адресе, а процессор, используя подготавливаемую операционной системой адресную информацию, выполняет второе отображение не один раз (при загрузке программы), а при каждом обращении к памяти.

Термин *виртуальная память* фактически относится к системам, которые сохраняют виртуальные адреса во время исполнения. Так как второе отображение осуществляется в процессе исполнения задачи, то адреса физических ячеек могут изменяться. При правильном применении такие изменения улучшают использование памяти, избавляя программиста от деталей управления ею, и даже повышают надежность вычислений.

Если рассматривать общую схему двухэтапного отображения адресов, то с позиции соотношения объемов упомянутых адресных пространств можно отметить наличие следующих трех ситуаций:

- объем виртуального адресного пространства программы V_v меньше объема физической памяти V_p ($V_v < V_p$);
- объем виртуального адресного пространства программы V_v равен объему физической памяти V_p ($V_v = V_p$);

- объем виртуального адресного пространства программы V_v больше объема физической памяти V_p ($V_v > V_p$).

Первая ситуация ($V_v < V_p$) ныне практически не встречается, но, тем не менее, это реальное соотношение. Скажем, не так давно 16-разрядные мини-ЭВМ имели систему команд, в которых пользователи-программисты могли адресовать до 2^{16} = 64 Кбайт адресов (обычно в качестве адресуемой единицы выступала ячейка памяти размером с байт). А физически старшие модели этих мини-ЭВМ могли иметь объем оперативной памяти в несколько мегабайтов. Обращение к памяти столь большого объема осуществлялось с помощью специальных регистров, содержимое которых складывалось с адресом операнда (или команды), извлекаемым из поля операнда или указателя команды (и/или определяемым по значению поля операнда или указателя команды). Соответствующие значения в эти специальные регистры, выступающие как базовое смещение в памяти, заносила операционная система. Для одной задачи в регистр заносилось одно значение, а для второй (третьей, четвертой и т. д.) задачи, размещаемой одновременно с первой, но в другой области памяти, заносилось, соответственно, другое значение. Вся физическая память таким образом разбивалась на разделы объемом по 64 Кбайт, и на каждый такой раздел осуществлялось отображение своего виртуального адресного пространства.

Вторая ситуация ($V_v = V_p$) встречается очень часто, особенно характерна она была для недорогих вычислительных комплексов. Для этого случая имеется большое количество методов распределения оперативной памяти.

Наконец, в наше время мы уже достигли того, что ситуация превышения объема виртуального адресного пространства программы над объемом физической памяти ($V_v > V_p$) характерна даже для персональных компьютеров, то есть для самых распространенных и недорогих машин. Теперь это самая обычная ситуация, и для нее имеется несколько методов распределения памяти, отличающихся как сложностью, так и эффективностью.

Простое непрерывное распределение и распределение с перекрытием

Общие принципы управления памятью в однопрограммных операционных системах

Простое непрерывное распределение — это самая простая схема, согласно которой вся память условно может быть разделена на три области:

- область, занимаемая операционной системой;
- область, в которой размещается исполняемая задача;
- незанятая ничем (свободная) область памяти.

Изначально являясь самой первой схемой, схема простого непрерывного распределения памяти продолжает и сегодня быть достаточно распространенной. Эта схема предполагает, что операционная система не поддерживает мультипрограммирование, поэтому не возникает проблемы распределения памяти между несколькими

задачами. Программные модули, необходимые для всех программ, располагаются в области самой операционной системы, а вся оставшаяся память может быть предоставлена задаче. Эта область памяти получается непрерывной, что облегчает работу системы программирования. Поскольку в различных однотипных вычислительных комплексах может быть разный состав внешних устройств (и, соответственно, они содержат различное количество драйверов), для системных нужд могут быть отведены отличающиеся объемы оперативной памяти, и получается, что можно не привязывать жестко виртуальные адреса программы к физическому адресному пространству. Эта привязка осуществляется на этапе загрузки задачи в память.

Для того чтобы для задач отвести как можно больший объем памяти, операционная система строится таким образом, чтобы постоянно в оперативной памяти располагалась только самая нужная ее часть. Эту часть операционной системы стали называть *ядром*. Прежде всего, в ядро операционной системы входят основные модули супервизора. Для однопрограммных систем понятие супервизора вырождается в модули, получающие и выполняющие первичную обработку запросов от обрабатывающих и прикладных программ, и в модули подсистемы памяти. Ведь если программа по ходу своего выполнения запрашивает некоторое множество ячеек памяти, то подсистема памяти должна их выделить (если они есть), а после освобождения этой памяти подсистема памяти должна выполнить действия, связанные с возвратом памяти в систему. Остальные модули операционной системы, не относящиеся к ее ядру, могут быть обычными диск-резидентными (или транзитными), то есть загружаться в оперативную память только по необходимости, и после своего выполнения вновь освобождать память.

Такая схема распределения влечет за собой два вида потерь вычислительных ресурсов — потеря процессорного времени, потому что процессор простаивает, пока задача ожидает завершения операций ввода-вывода, и потеря самой оперативной памяти, потому что далеко не каждая программа использует всю память, а режим работы в этом случае однопрограммный. Однако это очень недорогая реализация, которая позволяет отказаться от многих функций операционной системы. В частности, такая сложная проблема, как защита памяти, здесь почти не стоит. Единственное, что желательно защищать — это программные модули и области памяти самой операционной системы.

Если есть необходимость создать программу, логическое адресное пространство которой должно быть больше, чем свободная область памяти, или даже больше, чем весь возможный объем оперативной памяти, то используется распределение с перекрытием — так называемые *оверлейные структуры* (от *overlay* — перекрытие, расположение поверх чего-то). Этот метод распределения предполагает, что вся программа может быть разбита на части — сегменты. Каждая оверлейная программа имеет одну главную (main) часть и несколько сегментов (segments), причем в памяти машины одновременно могут находиться только ее главная часть и один или несколько не перекрывающихся сегментов.

Пока в оперативной памяти располагаются выполняющиеся сегменты, остальные находятся во внешней памяти. После того как текущий (выполняющийся) сегмент завершит свое выполнение, возможны два варианта: либо он сам (если данный сег-

мент не нужно сохранить во внешней памяти в его текущем состоянии) обращается к операционной системе с указанием, какой сегмент должен быть загружен в память следующим; либо он возвращает управление главному сегменту задачи, и уже тот обращается к операционной системе с указанием, какой сегмент сохранить (если это нужно), а какой сегмент загрузить в оперативную память, и вновь отдает управление одному из сегментов, располагающихся в памяти. Простейшие схемы сегментирования предполагают, что в памяти в каждый конкретный момент времени может располагаться только один сегмент (вместе с главным модулем). Более сложные схемы, используемые в больших вычислительных системах, позволяют располагать в памяти несколько сегментов. В некоторых вычислительных комплексах могли существовать отдельно сегменты кода и сегменты данных. Сегменты кода, как правило, не претерпевают изменений в процессе своего исполнения, поэтому при загрузке нового сегмента кода на место отработавшего последний можно не сохранять во внешней памяти, в отличие от сегментов данных, которые сохранять необходимо.

Первоначально программисты сами должны были включать в тексты своих программ соответствующие обращения к операционной системе (их называют системными вызовами) и тщательно планировать, какие сегменты могут находиться в оперативной памяти одновременно, чтобы их адресные пространства не пересекались. Однако с некоторых пор такого рода обращения к операционной системе системы программирования стали подставлять в код программы сами, автоматически, если в том возникает необходимость. Так, в известной и популярной в недалеком прошлом системе программирования Turbo Pascal программист просто указывал, что данный модуль является оверлейным. И при обращении к нему из основной программы модуль загружался в память и получал управление. Все адреса определялись системой программирования автоматически, обращения к DOS для загрузки оверлеев тоже генерировались системой Turbo Pascal.

Распределение оперативной памяти в MS DOS

Как известно, MS DOS[1] — это однопрограммная операционная система для персонального компьютера типа IBM PC. В ней, конечно, можно организовать запуск резидентных, или TSR-задач[2], в результате которого в памяти будет находиться не одна программа, но в целом система MS DOS предназначена для выполнения только одного вычислительного процесса. Поэтому распределение памяти в ней построено по схеме простого непрерывного распределения. Система поддерживает механизм распределения памяти с перекрытием (оверлейные структуры).

Как известно, в IBM PC использовался 16-разрядный микропроцессор i8088, который за счет введения сегментного способа адресации позволял указывать

[1] Версий однопрограммных дисковых операционных систем (Disks Operating System, DOS) для персональных компьютеров было много. Одних только MS DOS (систем от Microsoft) более 10. Однако несмотря на существенные различия все их чаще всего именуют одинаково — MS DOS.

[2] TSR (Terminate and Stay Resident) — резидентная в памяти программа, которая благодаря изменениям в таблице векторов прерываний позволяет перехватывать прерывания и в случае обращения к ней выполнять необходимые действия. Подробно об этом можно прочесть, например, в [3, 23, 24, 35].

адрес ячейки памяти в пространстве объемом до 1 Мбайт. В последующих персональных компьютерах (IBM PC AT, AT386 и др.) было принято решение поддерживать совместимость с первыми, поэтому при работе в DOS прежде всего рассматривают первый мегабайт. Вся эта память разделялась на несколько областей, что иллюстрирует рис. 3.2. На этом рисунке показано, что памяти может быть и больше, чем 1 Мбайт, но более подробное рассмотрение этого вопроса мы здесь опустим, отослав желающих изучить данную тему глубже к монографии [2].

Если не вдаваться в детали, можно сказать, что в состав MS DOS входят следующие основные компоненты.

- Подсистема BIOS (Base Input-Output System — базовая подсистема ввода-вывода), включающая в себя помимо программы POST (Power On Self Test — самотестирование при включении компьютера)[1] программные модули обработки прерываний, с помощью которых можно управлять основными контроллерами на материнской плате компьютера и устройствами ввода-вывода. Эти модули часто называют обработчиками прерываний. По своей функциональной сути они представляют собой драйверы. BIOS располагается в постоянном запоминающем устройстве компьютера. В конечном итоге почти все остальные модули MS DOS обращаются к BIOS. Если и не напрямую, то через модули более высокого уровня иерархии.
- Модуль расширения BIOS — файл `IO.SYS` (в других DOS-системах он может называться иначе, например `_BIO.COM`).
- Основной, или базовый, модуль обработки прерываний DOS — файл `MSDOS.SYS`. Именно этот модуль в основном реализует работу с файловой системой.
- Командный процессор (интерпретатор команд) — файл `COMMAND.COM`.
- Утилиты и драйверы, расширяющие возможности системы.
- Программа загрузки MS DOS — загрузочная запись (Boot Record, BR), расположенная на дискете или на жестком диске (подробнее о загрузочной записи и о других загрузчиках см. главу 6).

Вся память в соответствии с архитектурой IBM PC условно может быть разбита на следующие три части.

- В самых младших адресах памяти (первые 1024 ячейки) размещается таблица векторов прерывания (см. раздел «Система прерываний 32-разрядных микропроцессоров i80x86» в главе 4). Это связано с аппаратной реализацией процессора i8088. В последующих процессорах (начиная с i80286) адрес таблицы прерываний определяется через содержимое соответствующего регистра, но для обеспечения полной совместимости с первым процессором при включении или аппаратном сбросе в этот регистр заносятся нули. При желании, однако, в случае использования современных микропроцессоров i80x86 вектора прерываний можно размещать и в других областях.

[1] После выполнения программы POST, входящей в состав ROM BIOS, опрашиваются устройства, которые могут содержать программы для загрузки операционной системы.

0000-003FF	1 Кбайт	Таблица векторов прерываний	
00400-005FF	512 байт	Глобальные переменные BIOS; глобальные переменные DOS	В ранних версиях здесь располагались глобальные переменные интерпретатора Бейсик
00600-0A000	35-60 Кбайт	Модуль IO. SYS; Модуль MSDOS. SYS: - обслуживающие функции; - буферы, рабочие и управляющие области; - устанавливаемые драйверы; Резидентная часть COMMAND. COM: - обработка программных прерываний; - системная программа загрузки; - программа загрузки транзитной части COMMAND. COM	Размер этой области зависит от версии MSDOS и, главное, от конфигурационного файла CONFIG. SYS
	580 Кбайт	Область памяти для выполнения программ пользователя и утилит MS DOS. В эту область попадают программы типа *.COM и *.EXE	Объем этой области очень зависит от объема, занимаемого ядром ОС. Программа может перекрывать транзитную область COMMAND. COM
		Область расположения стека исполняющейся программы	Стек «растет» снизу вверх
	18 Кбайт	Транзитная часть командного процессора COMMAND. COM	Собственно командный интерпретатор
A0000-C7FFF	160 Кбайт	Видеопамять. Область и размер используемого видеобуфера зависят от текущего режима	При работе в текстовом режиме область памяти A0000-B0000 свободна и может быть использована в программе
C8000-E0000	96 Кбайт	Зарезервировано для расширения BIOS	
F0000-FFFF	64 Кбайт	Область ROM BIOS (System BIOS)	Обычно объем этой области равен 32 Кбайт, но может достигать и 128 Кбайт, занимая младшие адреса
Более 100000		High Memory Area. При наличии драйвера HIMEM. SYS здесь можно расположить основные системные файлы MS DOS, освобождая тем самым область основной памяти в первом мегабайте	Может использоваться при наличии специальных драйверов. Используются спецификации XMS и EMS

Рис. 3.2. Распределение оперативной памяти в MS DOS

- Вторая часть памяти отводится для программных модулей самой системы MS DOS и для программ пользователя. Эту область памяти мы рассмотрим чуть

позже, здесь только заметим, что она называется основной, или стандартной, памятью (conventional memory).

- Наконец, третья часть адресного пространства отведена для постоянных запоминающих устройств и функционирования некоторых устройств ввода-вывода. Эта область памяти получила название UMA (Upper Memory Area — область памяти, адрес которой выше основной).

В младших адресах основной памяти размещается то, что можно условно назвать ядром этой операционной системы — системные переменные, основные программные модули, блоки данных для буферизации операций ввода-вывода. Для управления устройствами, драйверы которых не входят в базовую подсистему ввода-вывода, загружаются так называемые *загружаемые*, или *устанавливаемые*, драйверы. Перечень устанавливаемых драйверов определяется специальным конфигурационным файлом CONFIG.SYS. После загрузки расширения BIOS — файла IO.SYS — последний (загрузив модуль MSDOS.SYS) считывает файл CONFIG.SYS и уже в соответствии с ним подгружает в память необходимые драйверы. Кстати, в конфигурационном файле CONFIG.SYS могут иметься операторы, указывающие на количество буферов, отводимых для ускорения операций ввода-вывода, и на количество файлов, которые могут обрабатываться (для работы с файлами необходимо зарезервировать место в памяти для хранения управляющих структур, с помощью которых выполняются операции с записями файла). В случае использования микропроцессоров i80x86 и наличия в памяти драйвера HIMEM.SYS модули IO.SYS и MSDOS.SYS могут быть размещены за пределами первого мегабайта в области, которая получила название *HMA* (High Memory Area — область памяти с большими адресами).

Память с адресами, большими чем 10FFFFh, может быть использована в DOS-программах при выполнении их на микропроцессорах, имеющих такую возможность (например, микропроцессор i80286 имел 24-разрядную шину адреса, а i80386 — уже 32-разрядную). Но для этого с помощью специальных драйверов необходимо переключать процессор в другой режим работы, при котором он сможет использовать адреса выше 10FFFFh. Широкое распространение получили две основные спецификации: *XMS* (Extended Memory Specification) и *EMS* (Expanded Memory Specification). Последние годы система MS DOS практически перестала применяться. Теперь ее используют в основном для запуска некоторых утилит, с помощью которых подготавливают дисковые устройства, или для установки других операционных систем. И поскольку основным утилитам, необходимым для обслуживания персонального компьютера, спецификации EMS и XMS, как правило, не нужны, мы не будем здесь их рассматривать.

Остальные программные модули MS DOS (в принципе, большинство из них является утилитами) оформлены как обычные исполняемые файлы. Например, утилита форматирования диска представляет собой и двоичный исполняемый файл, и команду операционной системы. В основном такого рода утилиты являются транзитными модулями, то есть загружаются в память только на время своей работы, хотя среди них имеются и TSR-программы.

Для того чтобы предоставить больше памяти программам пользователя, в MS DOS применено то же решение, что и во многих других простейших операционных

системах, — командный процессор COMMAND.COM состоит из двух частей. Первая часть является резидентной и размещается в области ядра, вторая часть транзитная и размещается в области старших адресов раздела памяти, выделяемой для программ пользователя. И если программа пользователя перекрывает собой область, в которой была расположена транзитная часть командного процессора, то последний при необходимости восстанавливает в памяти свою транзитную часть, поскольку после выполнения программы она возвращает управление резидентной части COMMAND.COM.

Поскольку размер основной памяти относительно небольшой, то очень часто системы программирования реализуют оверлейные структуры. Для этого в MS DOS поддерживаются специальные вызовы.

Распределение памяти статическими и динамическими разделами

Для организации мультипрограммного и/или мультизадачного режима необходимо обеспечить одновременное расположение в оперативной памяти нескольких задач (целиком или частями). Память задаче может выделяться одним сплошным участком (в этом случае говорят о методах *неразрывного распределения памяти*) или несколькими порциями, которые могут быть размещены в разных областях памяти (тогда говорят о методах *разрывного распределения*).

Начнем с методов неразрывного распределения памяти. Самая простая схема распределения памяти между несколькими задачами предполагает, что память, не занятая ядром операционной системы, может быть разбита на несколько непрерывных частей — *разделов* (partitions, regions). Разделы характеризуются именем, типом, границами (как правило, указываются начало раздела и его длина).

Разбиение памяти на несколько непрерывных (неразрывных) разделов может быть фиксированным (статическим) либо динамическим (то есть процесс выделения нового раздела памяти происходит непосредственно при появлении новой задачи). Вначале мы кратко рассмотрим статическое распределение памяти на разделы.

Разделы с фиксированными границами

Разбиение всего объема оперативной памяти на несколько разделов может осуществляться единовременно (то есть в процессе генерации варианта операционной системы, который потом и эксплуатируется) или по мере необходимости оператором системы. Однако и во втором случае при разбиении памяти на разделы вычислительная система более ни для каких целей в этот момент не используется. Пример разбиения памяти на несколько разделов приведен на рис. 3.3.

В каждом разделе в каждый момент времени может располагаться по одной программе (задаче). В этом случае по отношению к каждому разделу можно применить все те методы создания программ, которые используются для однопрограммных систем. Возможно использование оверлейных структур, что позволяет создавать большие сложные программы и в то же время поддерживать *коэффициент мульти-*

программирования[1] на должном уровне. Первые мультипрограммные операционные системы строились по этой схеме. Использовалась эта схема и много лет спустя при создании недорогих вычислительных систем, поскольку является несложной и обеспечивает возможность параллельного выполнения программ. Иногда в некотором разделе размещалось по нескольку небольших программ, которые постоянно в нем и находились. Такие программы назывались *ОЗУ-резидентными* (или просто *резидентными*). Та же схема используется и в современных встроенных системах; правда, для них характерно, что все программы являются резидентными, и внешняя память во время работы вычислительного оборудования не используется.

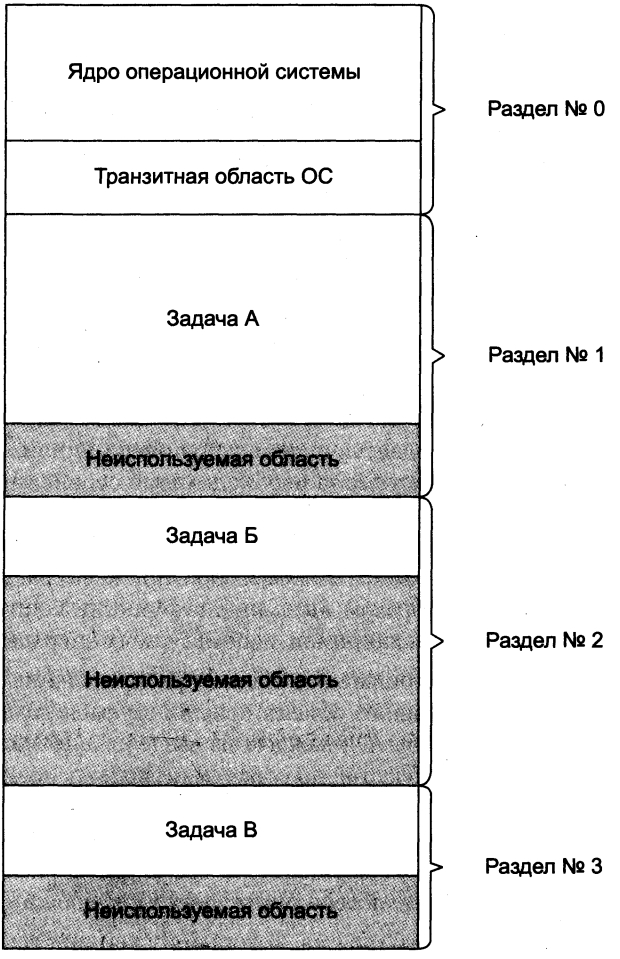


Рис. 3.3. Распределение памяти разделами с фиксированными границами

[1] Под коэффициентом мультипрограммирования (m) понимают количество параллельно выполняемых программ. Обычно на практике для загрузки центрального процессора до уровня 90 % необходимо, чтобы коэффициент мультипрограммирования был не менее 4–5. А для того чтобы наиболее полно использовать и остальные ресурсы системы, желательно иметь m на уровне 10–15.

При небольшом объеме памяти и, следовательно, небольшом количестве разделов увеличить число параллельно выполняемых приложений (особенно когда эти приложения интерактивны и во время своей работы фактически не используют процессорное время, а в основном ожидают операций ввода-вывода) можно за счет *замены их в памяти*, или *свопинга* (swapping). При свопинге задача может быть целиком выгружена на магнитный диск (перемещена во внешнюю память), а на ее место загружается либо более привилегированная, либо просто готовая к выполнению другая задача, находившаяся на диске в приостановленном состоянии. При свопинге из основной памяти во внешнюю (обратно) перемещается вся программа, а не ее отдельная часть.

Серьезная проблема, которая возникает при организации мультипрограммного режима работы вычислительной системы, — защита как самой операционной системы от ошибок и преднамеренного вмешательства процессов в ее работу, так и самих процессов друг от друга.

В самом деле, программа может обращаться к любым ячейкам в пределах своего виртуального адресного пространства. Если система отображения памяти не содержит ошибок, и в самой программе их тоже нет, то возникать ошибок при выполнении программы не должно. Однако в случае ошибок адресации, что случается не так уж и редко, исполняющаяся программа может начать «обработку» чужих данных или кодов с непредсказуемыми последствиями. Одной из простейших, но достаточно эффективных мер является введение регистров защиты памяти. В эти регистры операционная система заносит граничные значения области памяти раздела текущего исполняющегося процесса. При нарушении адресации возникает прерывание, и управление передается супервизору операционной системы. Обращения задач к операционной системе за необходимыми сервисами осуществляются не напрямую, а через команды программных прерываний, что обеспечивает передачу управления только в предопределенные входные точки кода операционной системы и в системном режиме работы процессора, при котором регистры защиты памяти игнорируются. Таким образом, выполнение функции защиты требует введения специальных аппаратных механизмов, используемых операционной системой.

Основным недостатком рассматриваемого способа распределения памяти является наличие порой достаточно большого объема неиспользуемой памяти (см. рис. 3.3). Неиспользуемая память может быть в каждом из разделов. Поскольку разделов несколько, то и неиспользуемых областей получается несколько, поэтому такие потери стали называть *фрагментацией памяти*. В отдельных разделах потери памяти могут быть очень значительными, однако использовать фрагменты свободной памяти при таком способе распределения не представляется возможным. Желание разработчиков сократить столь значительные потери привело их к следующим двум решениям:

- выделять раздел ровно такого объема, который нужен под текущую задачу;
- размещать задачу не в одной непрерывной области памяти, а в нескольких областях.

Второе решение было реализовано в нескольких способах организации виртуальной памяти. Мы их обсудим в следующем разделе, а сейчас кратко рассмотрим первое решение.

Разделы с подвижными границами

Чтобы избавиться от фрагментации, можно попробовать размещать в оперативной памяти задачи плотно, одну за другой, выделяя ровно столько памяти, сколько задача требует. Одной из первых операционных систем, в которой был реализован такой способ распределения памяти, была OS MVT[1] (Multiprogramming with a Variable number of Tasks — мультипрограммирование с переменным числом задач). В этой операционной системе специальный планировщик (диспетчер памяти) ведет список адресов свободной оперативной памяти. При появлении новой задачи диспетчер памяти просматривает этот список и выделяет для задачи раздел, объем которой либо равен необходимому, либо чуть больше, если память выделяется не ячейками, а некими дискретными единицами. При этом модифицируется список свободных областей памяти. При освобождении раздела диспетчер памяти пытается объединить освобождающийся раздел с одним из свободных участков, если таковой является смежным.

При этом список свободных участков памяти может быть упорядочен либо по адресам, либо по объему. Выделение памяти под новый раздел может осуществляться одним из трех основных способов:

- первый подходящий участок;
- самый подходящий участок;
- самый неподходящий участок.

В первом случае список свободных областей упорядочивается по адресам (например, по возрастанию адресов). Диспетчер просматривает список и выделяет задаче раздел в той области, которая первой подойдет по объему. В этом случае, если такой фрагмент имеется, то в среднем необходимо просмотреть половину списка. При освобождении раздела также необходимо просмотреть половину списка. Правило «первый подходящий» приводит к тому, что память для небольших задач преимущественно будет выделяться в области младших адресов, и, следовательно, это увеличит вероятность того, что в области старших адресов будут образовываться фрагменты достаточно большого объема.

Способ «самый подходящий» предполагает, что список свободных областей упорядочен по возрастанию объема фрагментов. В этом случае при просмотре списка для нового раздела будет использован фрагмент свободной памяти, объем которой наиболее точно соответствует требуемому. Требуемый раздел будет определяться по-прежнему в результате просмотра в среднем половины списка. Однако оставшийся фрагмент оказывается настолько малым, что в нем уже вряд ли удастся разместить еще какой-либо раздел. При этом получается, что вновь образованный фрагмент попадет в начало списка, и в последующем его придется каждый раз проверять на пригодность, тогда как его малый размер вряд ли окажется подходящим. Поэтому в целом такую дисциплину нельзя назвать эффективной.

Как ни странно, самым эффективным способом, как правило, является последний, по которому для нового раздела выделяется «самый неподходящий» фрагмент сво-

[1] Эта операционная система была одной из самых распространенных в больших ЭВМ класса IBM 360 (370).

бодной памяти. Для этой дисциплины список свободных областей упорядочивается по убыванию объема свободного фрагмента. Очевидно, что если есть такой фрагмент памяти, то он сразу же и будет найден, и, поскольку этот фрагмент является самым большим, то, скорее всего, после выделения из него раздела памяти для задачи оставшуюся область памяти можно будет использовать в дальнейшем.

Однако очевидно, что при любой дисциплине обслуживания, по которой работает диспетчер памяти, из-за того что задачи появляются и завершаются в произвольные моменты времени и при этом имеют разные объемы, в памяти всегда будет наблюдаться сильная фрагментация. При этом возможны ситуации, когда из-за сильной фрагментации памяти диспетчер задач не сможет образовать новый раздел, хотя суммарный объем свободных областей будет больше, чем необходимо для задачи. В этой ситуации можно организовать так называемое *уплотнение памяти*. Для уплотнения памяти все вычисления приостанавливаются, и диспетчер памяти корректирует свои списки, перемещая разделы в начало памяти (или, наоборот, в область старших адресов). При определении физических адресов задачи будут участвовать новые значения базовых регистров, с помощью которых и осуществляется преобразование виртуальных адресов в физические. Недостатком этого решения является потеря времени на уплотнение и, что самое главное, невозможность при этом выполнять сами вычислительные процессы.

Данный способ распределения памяти, тем не менее, применялся достаточно длительное время в нескольких операционных системах, поскольку в нем для задач выделяется непрерывное адресное пространство, а это упрощает создание систем программирования и их работу. Применяется этот способ и ныне при создании систем на базе контроллеров с упрощенной (по отношению к мощным современным процессорам) архитектурой. Например, при разработке операционной системы для современных цифровых АТС, которая использует 16-разрядные микропроцессоры Intel.

Сегментная, страничная и сегментно-страничная организация памяти

Методы распределения памяти, при которых задаче уже может не предоставляться сплошная (непрерывная) область памяти, называют *разрывными*. Идея выделять память задаче не одной сплошной областью, а фрагментами позволяет уменьшить фрагментацию памяти, однако этот подход требует для своей реализации больше ресурсов, он намного сложнее. Если задать адрес начала текущего фрагмента программы и величину смещения относительно этого начального адреса, то можно указать необходимую нам переменную или команду. Таким образом, виртуальный адрес можно представить состоящим из двух полей. Первое поле будет указывать на ту часть программы, к которой обращается процессор, для определения местоположения этой части в памяти, а второе поле виртуального адреса позволит найти нужную нам ячейку относительно найденного адреса. Программист может либо самостоятельно разбивать программу на фрагменты, либо можно автоматизировать эту задачу, возложив ее на систему программирования.

Сегментный способ организации виртуальной памяти

Первым среди разрывных методов распределения памяти был *сегментный*. Для этого метода программу необходимо разбивать на части и уже каждой такой части выделять физическую память. Естественным способом разбиения программы на части является разбиение ее на логические элементы — так называемые *сегменты*. В принципе, каждый программный модуль (или их совокупность, если мы того пожелаем) может быть воспринят как отдельный сегмент, и вся программа тогда будет представлять собой множество сегментов. Каждый сегмент размещается в памяти как до определенной степени самостоятельная единица. Логически обращение к элементам программы в этом случае будет состоять из имени сегмента и смещения относительно начала этого сегмента. Физически имя (или порядковый номер) сегмента будет соответствовать некоторому адресу, с которого этот сегмент начинается при его размещении в памяти, и смещение должно прибавляться к этому базовому адресу.

Преобразование имени сегмента в его порядковый номер осуществит система программирования. Для каждого сегмента система программирования указывает его объем. Он должен быть известен операционной системе, чтобы она могла выделять ему необходимый объем памяти. Операционная система будет размещать сегменты в памяти и для каждого сегмента она должна вести учет о местонахождении этого сегмента. Вся информация о текущем размещении сегментов задачи в памяти обычно сводится в *таблицу сегментов*, чаще такую таблицу называют *таблицей дескрипторов сегментов задачи*. Каждая задача имеет свою таблицу сегментов. Достаточно часто эти таблицы называют таблицами дескрипторов сегментов, поскольку по своей сути элемент таблицы описывает расположение сегмента.

Таким образом, виртуальный адрес для этого способа будет состоять из двух полей — номера сегмента и смещения относительно начала сегмента. Соответствующая иллюстрация приведена на рис. 3.4 для случая обращения к ячейке, виртуальный адрес которой равен сегменту с номером 11 со смещением от начала этого сегмента, равным 612. Как мы видим, операционная система разместила данный сегмент в памяти, начиная с ячейки с номером 19700.

Итак, каждый сегмент, размещаемый в памяти, имеет соответствующую информационную структуру, часто называемую *дескриптором сегмента*. Именно операционная система строит для каждого исполняемого процесса соответствующую таблицу дескрипторов сегментов, и при размещении каждого из сегментов в оперативной или внешней памяти отмечает в дескрипторе текущее местоположение сегмента. Если сегмент задачи в данный момент находится в оперативной памяти, то об этом делается пометка в дескрипторе. Как правило, для этого используется *бит присутствия* P (от слова «present»). В этом случае в поле адреса диспетчер памяти записывает адрес физической памяти, с которого сегмент начинается, а в поле длины сегмента (limit) указывается количество адресуемых ячеек памяти. Это поле используется не только для того, чтобы размещать сегменты без наложения друг на друга, но и для того, чтобы контролировать, не обращается ли код исполняющейся задачи за пределы текущего сегмента. В случае превышения длины сегмен-

та вследствие ошибок программирования мы можем говорить о нарушении адресации и с помощью введения специальных аппаратных средств генерировать сигналы прерывания, которые позволят фиксировать (обнаруживать) такого рода ошибки.

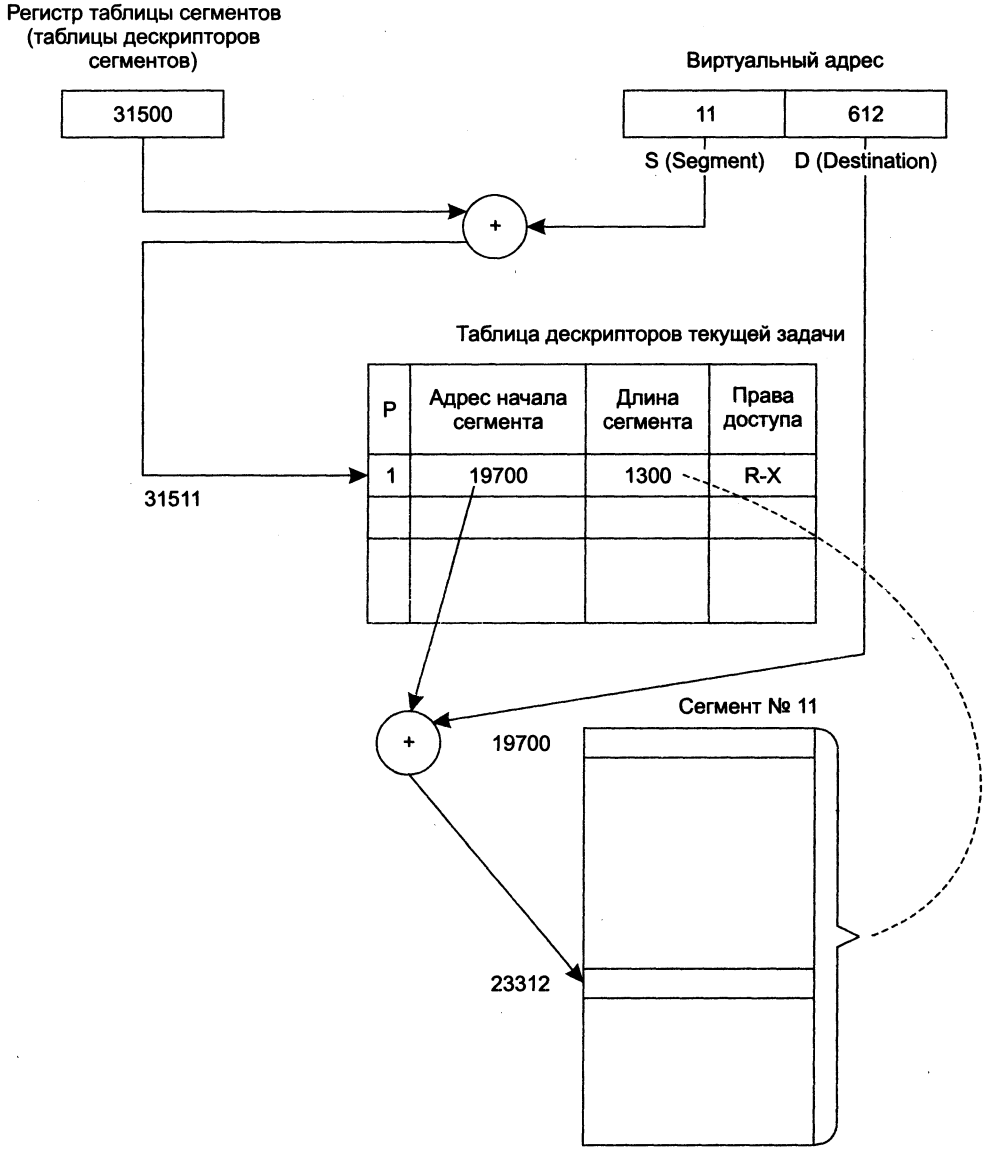


Рис. 3.4. Сегментный способ организации виртуальной памяти

Если бит присутствия в дескрипторе указывает, что сегмент находится не в оперативной, а во внешней памяти (например, на жестком диске), то названные поля

адреса и длины используются для указания адреса сегмента в координатах внешней памяти. Помимо информации о местоположении сегмента, в дескрипторе сегмента, как правило, содержатся данные о его типе (сегмент кода или сегмент данных), правах доступа к этому сегменту (можно или нельзя его модифицировать, предоставлять другой задаче), отметка об обращениях к данному сегменту (информация о том, как часто или как давно этот сегмент используется или не используется, на основании которой можно принять решение о том, чтобы предоставить место, занимаемое текущим сегментом, другому сегменту).

При передаче управления следующей задаче операционная система должна занести в соответствующий регистр адрес таблицы дескрипторов сегментов этой задачи. Сама таблица дескрипторов сегментов, в свою очередь, также представляет собой сегмент данных, который обрабатывается диспетчером памяти операционной системы.

При таком подходе появляется возможность размещать в оперативной памяти не все сегменты задачи, а только задействованные в данный момент. Благодаря этому, с одной стороны, общий объем виртуального адресного пространства задачи может превосходить объем физической памяти компьютера, на котором эта задача будет выполняться; с другой стороны, даже если потребности в памяти не превосходят имеющуюся физическую память, можно размещать в памяти больше задач, поскольку любой задаче, как правило, все ее сегменты единовременно не нужны. А увеличение *коэффициента мультипрограммирования* μ, как мы знаем, позволяет увеличить загрузку системы и более эффективно использовать ресурсы вычислительной системы. Очевидно, однако, что увеличивать количество задач можно только до определенного предела, ибо если в памяти не будет хватать места для часто используемых сегментов, то производительность системы резко упадет. Ведь сегмент, находящийся вне оперативной памяти, для участия в вычислениях должен быть перемещен в оперативную память. При этом если в памяти есть свободное пространство, то необходимо всего лишь найти нужный сегмент во внешней памяти и загрузить его в оперативную память. А если свободного места нет, придется принять решение — на место какого из присутствующих сегментов будет загружаться требуемый. Перемещение сегментов из оперативной памяти на жесткий диск и обратно часто называют *свопингом сегментов*.

Итак, если требуемого сегмента в оперативной памяти нет, то возникает прерывание, и управление передается через диспетчер памяти программе загрузки сегмента. Пока происходит поиск сегмента во внешней памяти и загрузка его в оперативную, диспетчер памяти определяет подходящее для сегмента место. Возможно, что свободного места нет, и тогда принимается решение о выгрузке какого-нибудь сегмента и выполняется его перемещение во внешнюю память. Если при этом еще остается время, то процессор передается другой готовой к выполнению задаче. После загрузки необходимого сегмента процессор вновь передается задаче, вызвавшей прерывание из-за отсутствия сегмента. Всякий раз при считывании сегмента в оперативную память в таблице дескрипторов сегментов необходимо установить адрес начала сегмента и признак присутствия сегмента.

При поиске свободного места используется одна из вышеперечисленных дисциплин работы диспетчера памяти (применяются правила «первого подходящего»

и «самого неподходящего» фрагментов). Если свободного фрагмента памяти достаточного объема нет, но, тем не менее, сумма этих свободных фрагментов превышает требования по памяти для нового сегмента, то в принципе может быть применено «уплотнение памяти», о котором мы уже говорили в подразделе «Разделы с фиксированными границами» раздела «Распределение памяти статическими и динамическими разделами».

В идеальном случае размер сегмента должен быть достаточно малым, чтобы его можно было разместить в случайно освобождающихся фрагментах оперативной памяти, но достаточно большим, чтобы содержать логически законченную часть программы с тем, чтобы минимизировать межсегментные обращения.

Для решения проблемы замещения (определения того сегмента, который должен быть либо перемещен во внешнюю память, либо просто замещен новым) используются следующие дисциплины[1]:

- правило *FIFO* (First In First Out — первый пришедший первым и выбывает);
- правило *LRU* (Least Recently Used — дольше других неиспользуемый);
- правило *LFU* (Least Frequently Used — реже других используемый);
- *случайный* (random) выбор сегмента.

Первая и последняя дисциплины являются самыми простыми в реализации, но они не учитывают, насколько часто используется тот или иной сегмент, и, следовательно, диспетчер памяти может выгрузить или расформировать тот сегмент, к которому в самом ближайшем будущем будет обращение. Безусловно, достоверной информация о том, какой из сегментов потребуется в ближайшем будущем, в общем случае быть не может, но вероятность ошибки для этих дисциплин многократно выше, чем у второй и третьей, в которых учитывается информация об использовании сегментов.

В алгоритме FIFO с каждым сегментом связывается очередность его размещения в памяти. Для замещения выбирается сегмент, первым попавший в память. Каждый вновь размещаемый в памяти сегмент добавляется в хвост этой очереди. Алгоритм учитывает только время нахождения сегмента в памяти, но не учитывает фактическое использование сегментов. Например, первые загруженные сегменты программы могут содержать переменные, требующиеся на протяжении всей ее работы. Это приводит к немедленному возвращению к только что замещенному сегменту.

Для реализации дисциплин LRU и LFU необходимо, чтобы процессор имел дополнительные аппаратные средства. Минимальные требования — достаточно, чтобы при обращении к дескриптору сегмента для получения физического адреса, с которого сегмент начинает располагаться в памяти, соответствующий *бит обращения* менял свое значение (скажем, с нулевого, которое устанавливает операционная система, в единичное). Тогда диспетчер памяти может время от времени просматривать таблицы дескрипторов исполняющихся задач и собирать для соответствующей обработки статистическую информацию об обращениях к сегмен-

[1] Их называют «дисциплинами замещения».

там. В результате можно составить список, упорядоченный либо по длительности простоя (для дисциплины LRU), либо по частоте использования (для дисциплины LFU).

Важнейшей проблемой, которая возникает при организации мультипрограммного режима, является защита памяти. Для того чтобы выполняющиеся приложения не смогли испортить саму операционную систему и другие вычислительные процессы, необходимо, чтобы доступ к таблицам сегментов с целью их модификации был обеспечен только для кода самой ОС. Для этого код операционной системы должен выполняться в некотором привилегированном режиме, из которого можно осуществлять манипуляции дескрипторами сегментов, тогда как выход за пределы сегмента в обычной прикладной программе должен вызывать прерывание по защите памяти. Каждая прикладная задача должна иметь возможность обращаться только к собственным и к общим сегментам.

При сегментном способе организации виртуальной памяти появляется несколько интересных возможностей.

Во-первых, при загрузке программы на исполнение можно размещать ее в памяти не целиком, а «по мере необходимости». Действительно, поскольку в подавляющем большинстве случаев алгоритм, по которому работает код программы, является разветвленным, а не линейным, то в зависимости от исходных данных некоторые части программы, расположенные в самостоятельных сегментах, могут быть не задействованы; значит, их можно и не загружать в оперативную память.

Во-вторых, некоторые программные модули могут быть разделяемыми. Поскольку эти программные модуля являются сегментами, относительно легко организовать доступ к таким общим сегментам. Сегмент с разделяемым кодом располагается в памяти в единственном экземпляре, а в нескольких таблицах дескрипторов сегментов исполняющихся задач будут находиться указатели на такие разделяемые сегменты.

Однако у сегментного способа распределения памяти есть и недостатки. Прежде всего (см. рис. 3.4), для доступа к искомой ячейке памяти приходится тратить много времени. Мы должны сначала найти и прочитать дескриптор сегмента, а уже потом, используя полученные данные о местонахождении нужного нам сегмента, вычислить конечный физический адрес. Для того чтобы уменьшить эти потери, используется кэширование — те дескрипторы, с которыми мы имеем дело в данный момент, могут быть размещены в сверхоперативной памяти (специальных регистрах, размещаемых в процессоре).

Несмотря на то что рассмотренный способ распределения памяти приводит к существенно меньшей фрагментации памяти, нежели способы с неразрывным распределением, фрагментация остается. Кроме того, много памяти и процессорного времени теряется на размещение и обработку дескрипторных таблиц. Ведь на каждую задачу необходимо иметь свою таблицу дескрипторов сегментов. А при определении физических адресов приходится выполнять операции сложения, что требует дополнительных затрат времени.

Поэтому следующим способом разрывного размещения задач в памяти стал способ, при котором все фрагменты задачи считаются равными (одинакового разме-

ра), причем длина фрагмента в идеале должна быть кратна степени двойки, чтобы операции сложения можно было заменить операциями конкатенации (слияния). Это — страничный способ организации виртуальной памяти. Этот способ мы детально рассмотрим ниже.

Примером использования сегментного способа организации виртуальной памяти является операционная система OS/2 первого поколения[1], которая была создана для персональных компьютеров на базе процессора i80286. В этой операционной системе в полной мере использованы аппаратные средства микропроцессора, который специально проектировался для поддержки сегментного способа распределения памяти.

OS/2 v.1 поддерживала распределение памяти, при котором выделялись сегменты программы и сегменты данных. Система позволяла работать как с именованными, так и с неименованными сегментами. Имена разделяемых сегментов данных имели ту же форму, что и имена файлов. Процессы получали доступ к именованным разделяемым сегментам, используя их имена в специальных системных вызовах. Операционная система OS/2 v.1 допускала разделение программных сегментов приложений и подсистем, а также глобальных сегментов данных подсистем. Вообще, вся концепция системы OS/2 была построена на понятии разделения памяти: процессы почти всегда разделяют сегменты с другими процессами. В этом состояло существенное отличие системы OS/2 от систем типа UNIX, которые обычно разделяют только реентерабельные программные модули между процессами.

Сегменты, которые активно не использовались, могли выгружаться на жесткий диск. Система восстанавливала их, когда в этом возникала необходимость. Так как все области памяти, используемые сегментом, должны были быть непрерывными, OS/2 перемещала в основной памяти сегменты таким образом, чтобы максимизировать объем свободной физической памяти. Такое переразмещение сегментов называется уплотнением памяти (компрессией). Программные сегменты не выгружались, поскольку они могли просто перезагружаться с исходных дисков. Области в младших адресах физической памяти, которые использовались для запуска DOS-программ и кода самой OS/2, в компрессии не участвовали. Кроме того, система или прикладная программа могла временно фиксировать сегмент в памяти с тем, чтобы гарантировать наличие буфера ввода-вывода в физической памяти до тех пор, пока операция ввода-вывода не завершится.

Если в результате компрессии памяти не удавалось создать необходимое свободное пространство, то супервизор выполнял операции фонового плана для перекачки достаточного количества сегментов из физической памяти, чтобы дать возможность завершиться исходному запросу.

Механизм перекачки сегментов использовал файловую систему для выгрузки данных из физической памяти и обратно. Ввиду того что перекачка и компрессия влияли на производительность системы в целом, пользователь мог сконфигурировать систему так, чтобы эти функции не выполнялись.

---

[1] OS/2 v.1 начала создаваться в 1984 году и поступила в продажу в 1987 году.

Было организовано в OS/2 и динамическое присоединение обслуживающих программ. Программы OS/2 используют команды удаленного вызова. Ссылки, генерируемые этими вызовами, определяются в момент загрузки самой программы или ее сегментов. Такое отсроченное определение ссылок называется *динамическим присоединением*. Загрузочный формат модуля OS/2 представляет собой расширение формата загрузочного модуля DOS. Он был расширен, чтобы поддерживать необходимое окружение для свопинга сегментов с динамическим присоединением. Динамическое присоединение уменьшает объем памяти для программ в OS/2, одновременно делая возможными перемещения подсистем и обслуживающих программ без необходимости повторного редактирования адресных ссылок к прикладным программам.

Страничный способ организации виртуальной памяти

Как уже упоминалось, при страничном способе организации виртуальной памяти все фрагменты программы, на которые она разбивается (за исключением последней ее части), получаются одинаковыми. Одинаковыми полагаются и единицы памяти, которые предоставляются для размещения фрагментов программы. Эти одинаковые части называют *страницами* и говорят, что оперативная память разбивается на физические страницы, а программа — на виртуальные страницы. Часть виртуальных страниц задачи размещается в оперативной памяти, а часть — во внешней. Обычно место во внешней памяти, в качестве которой в абсолютном большинстве случаев выступают накопители на магнитных дисках (поскольку они относятся к быстродействующим устройствам с прямым доступом), называют *файлом подкачки*, или *страничным файлом* (paging file). Иногда этот файл называют *swap-файлом*, тем самым подчеркивая, что записи этого файла — страницы — замещают друг друга в оперативной памяти. В некоторых операционных системах выгруженные страницы располагаются не в файле, а в специальном разделе дискового пространства[1].

Разбиение всей оперативной памяти на страницы одинаковой величины, причем кратной степени двойки, приводит к тому, что вместо одномерного адресного пространства памяти можно говорить о двухмерном. Первая координата адресного пространства — это номер страницы, вторая координата — номер ячейки внутри выбранной страницы (его называют индексом). Таким образом, физический адрес определяется парой (P_p, i), а виртуальный адрес — парой (P_v, i), где P_v — номер виртуальной страницы, P_p — номер физической страницы, i — индекс ячейки внутри страницы. Количество битов, отводимое под индекс, определяет размер страницы, а количество битов, отводимое под номер виртуальной страницы, — объем потенциально доступной для программы виртуальной памяти. Отображение, осуществляемое системой во время исполнения, сводится к отображению P_v в P_p и приписыванию к полученному значению битов адреса, задаваемых величиной i. При этом

[1] В UNIX-системах для этих целей выделяется специальный раздел, но кроме него могут быть использованы и файлы, выполняющие те же функции, если объема раздела недостаточно.

нет необходимости ограничивать число виртуальных страниц числом физических, то есть не поместившиеся страницы можно размещать во внешней памяти, которая в данном случае служит расширением оперативной.

Для отображения виртуального адресного пространства задачи на физическую память, как и в случае сегментного способа организации, для каждой задачи необходимо иметь *таблицу страниц* для трансляции адресных пространств. Для описания каждой страницы диспетчер памяти операционной системы заводит соответствующий дескриптор, который отличается от дескриптора сегмента прежде всего тем, что в нем нет поля длины — ведь все страницы имеют одинаковый размер. По номеру виртуальной страницы в таблице дескрипторов страниц текущей задачи находится соответствующий элемент (дескриптор). Если бит присутствия имеет единичное значение, значит данная страница размещена в оперативной, а не во внешней памяти, и мы в дескрипторе имеем номер физической страницы, отведенной под данную виртуальную. Если же бит присутствия равен нулю, то в дескрипторе мы будем иметь адрес виртуальной страницы, расположенной во внешней памяти. Таким образом и осуществляется трансляция виртуального адресного пространства на физическую память. Этот механизм трансляции иллюстрирует рис. 3.5.

Защита страничной памяти, как и в случае сегментного механизма, основана на контроле уровня доступа к каждой странице. Как правило, возможны следующие уровни доступа:

- только чтение;
- чтение и запись;
- только выполнение.

Каждая страница снабжается соответствующим кодом уровня доступа. При трансформации логического адреса в физический сравнивается значение кода разрешенного уровня доступа с фактически требуемым. При их несовпадении работа программы прерывается.

При обращении к виртуальной странице, не оказавшейся в данный момент в оперативной памяти, возникает прерывание, и управление передается диспетчеру памяти, который должен найти свободное место. Обычно предоставляется первая же свободная страница. Если свободной физической страницы нет, то диспетчер памяти по одной из вышеупомянутых дисциплин замещения (LRU, LFU, FIFO, случайный доступ) определит страницу, подлежащую расформированию или сохранению во внешней памяти. На ее месте он разместит новую виртуальную страницу, к которой было обращение из задачи, но которой не оказалось в оперативной памяти.

Напомним, что алгоритм LFU выбирает для замещения ту страницу, на которую не было ссылки на протяжении наиболее длительного периода времени. Алгоритм LRU ассоциирует с каждой страницей время ее последнего использования. Для замещения выбирается та страница, которая дольше всех не использовалась.

Для использования дисциплин LRU и LFU в процессоре должны быть соответствующие аппаратные средства. В дескрипторе страницы размещается бит обращения (на рис. 3.5 подразумевается, что этот бит расположен в последнем поле), который становится единичным при обращении к дескриптору.

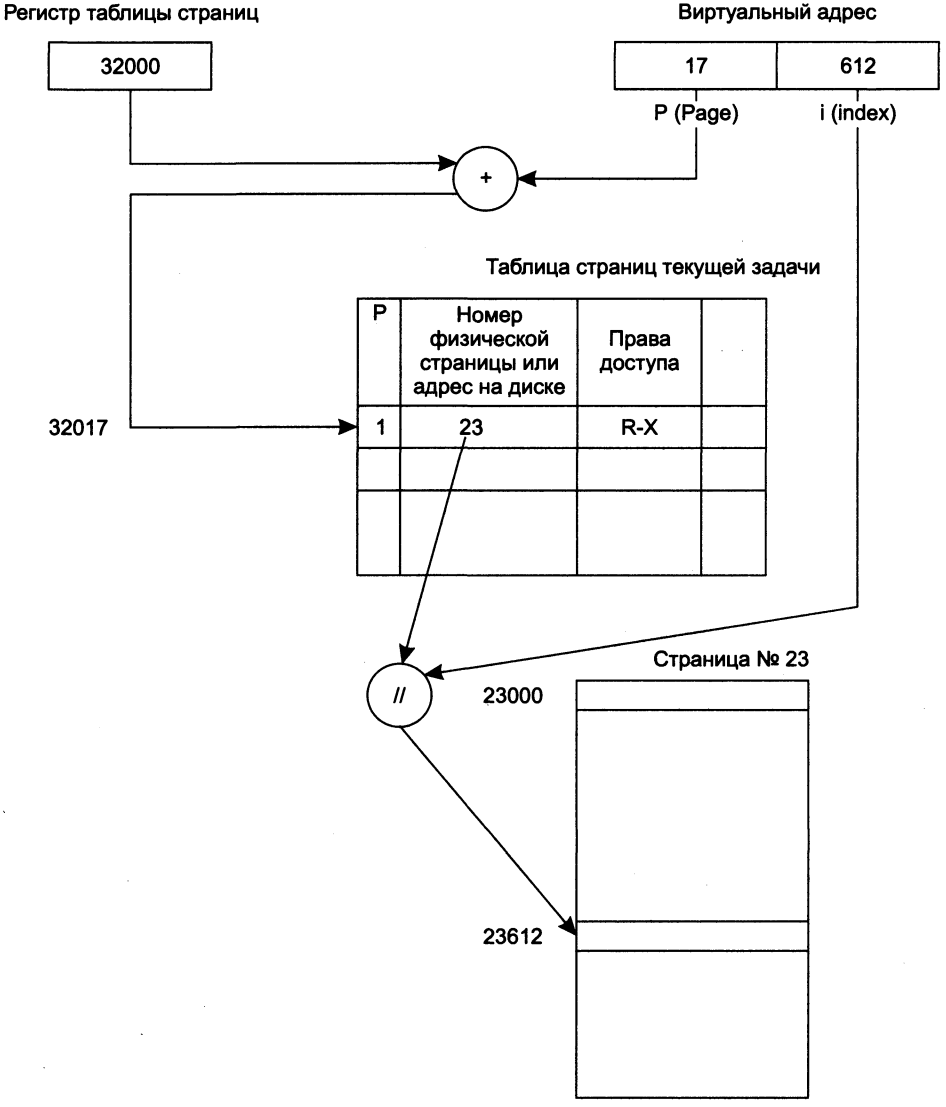


Рис. 3.5. Страничный способ организации виртуальной памяти

Если объем физической памяти небольшой и даже часто требуемые страницы не удается разместить в оперативной памяти, возникает так называемая «пробуксовка». Другими словами, *пробуксовка* — это ситуация, при которой загрузка нужной страницы вызывает перемещение во внешнюю память той страницы, с которой мы тоже активно работаем. Очевидно, что это очень плохое явление. Чтобы его не допускать, желательно увеличить объем оперативной памяти (сейчас это просто, поскольку стоимость модуля оперативной памяти многократно снизилась), уменьшить количество параллельно выполняемых задач или прибегнуть к более эффективным дисциплинам замещения.

Для абсолютного большинства современных операционных систем характерна дисциплина замещения страниц LRU как самая эффективная. Так, именно эта дисциплина используется в OS/2 и в Linux. Однако в операционных системах Windows NT/2000/XP разработчики, желая сделать их максимально независимыми от аппаратных возможностей процессора, отказались от этой дисциплины и применили правило FIFO. А для того чтобы хоть как-то компенсировать неэффективность правила FIFO, была введена «буферизация» тех страниц, которые должны быть записаны в файл подкачки на диск[1] или просто расформированы. Принцип буферизации прост. Прежде чем замещаемая страница действительно окажется во внешней памяти или просто расформированной, она помечается как кандидат на выгрузку. Если в следующий раз произойдет обращение к странице, находящейся в таком «буфере», то страница никуда не выгружается и уходит в конец списка FIFO. В противном случае страница действительно выгружается, а на ее место в «буфер» попадает следующий «кандидат». Величина такого «буфера» не может быть большой, поэтому эффективность страничной реализации памяти в Windows NT/2000/XP намного ниже, чем в других операционных системах, и явление пробуксовки начинается даже при существенно большем объеме оперативной памяти.

В ряде операционных систем с пакетным режимом работы для борьбы с пробуксовкой используется метод «рабочего множества». *Рабочее множество* — это множество «активных» страниц задачи за некоторый интервал T, то есть тех страниц, к которым было обращение за этот интервал времени. Реально количество активных страниц задачи (за интервал T) все время изменяется, и это естественно, но, тем не менее, для каждой задачи можно определить среднее количество ее активных страниц. Это количество и есть рабочее множество задачи. Наблюдения за исполнением множества различных программ показали [11, 17, 22], что даже если интервал T равен времени выполнения всей работы, то размер рабочего множества часто существенно меньше, чем общее число страниц программы. Таким образом, если операционная система может определить рабочие множества исполняющихся задач, то для предотвращения пробуксовки достаточно планировать на выполнение только такое количество задач, чтобы сумма их рабочих множеств не превышала возможности системы.

Как и в случае с сегментным способом организации виртуальной памяти, страничный механизм приводит к тому, что без специальных аппаратных средств он существенно замедляет работу вычислительной системы. Поэтому обычно используется кэширование страничных дескрипторов. Наиболее эффективным механизмом кэширования является ассоциативный кэш. Именно такой ассоциативный кэш и создан в 32-разрядных микропроцессорах i80x86. Начиная с i80386, который поддерживает страничный способ распределения памяти, в этих микропроцессорах имеется кэш на 32 страничных дескриптора. Поскольку размер страницы в этих

[1] В системе Windows NT файл с выгруженными виртуальными страницами носит название PageFile.sys. Таких файлов может быть несколько. Их совокупный размер должен быть не менее, чем объем физической памяти компьютера плюс 11 Мбайт, необходимых для самой Windows NT. В системах Windows 2000 размер файла PageFile.sys намного превышает объем установленной физической памяти и часто достигает многих сотен мегабайтов.

микропроцессорах равен 4 Кбайт, возможно быстрое обращение к памяти размером 128 Кбайт.

Итак, основным достоинством страничного способа распределения памяти является минимальная фрагментация. Поскольку на каждую задачу может приходиться по одной незаполненной странице, очевидно, что память можно использовать достаточно эффективно; этот метод организации виртуальной памяти был бы одним из самых лучших, если бы не два следующих обстоятельства.

Первое — это то, что страничная трансляция виртуальной памяти требует существенных накладных расходов. В самом деле, таблицы страниц нужно тоже размещать в памяти. Кроме того, эти таблицы нужно обрабатывать; именно с ними работает диспетчер памяти.

Второй существенный недостаток страничной адресации заключается в том, что программы разбиваются на страницы случайно, без учета логических взаимосвязей, имеющихся в коде. Это приводит к тому, что межстраничные переходы, как правило, осуществляются чаще, нежели межсегментные, и к тому, что становится трудно организовать разделение программных модулей между выполняющимися процессами.

Для того чтобы избежать второго недостатка, постаравшись сохранить достоинства страничного способа распределения памяти, был предложен еще один способ — сегментно-страничный. Правда, за счет увеличения накладных расходов на его реализацию.

Сегментно-страничный способ организации виртуальной памяти

Как и в сегментном способе распределения памяти, программа разбивается на логически законченные части — сегменты — и виртуальный адрес содержит указание на номер соответствующего сегмента. Вторая составляющая виртуального адреса — смещение относительно начала сегмента — в свою очередь может быть представлено состоящим из двух полей: виртуальной страницы и индекса. Другими словами, получается, что виртуальный адрес теперь состоит из трех компонентов: сегмента, страницы и индекса. Получение физического адреса и извлечение из памяти необходимого элемента для этого способа иллюстрирует рис. 3.6.

Из рисунка сразу видно, что этот способ организации виртуальной памяти вносит еще большую задержку доступа к памяти. Необходимо сначала вычислить адрес дескриптора сегмента и прочитать его, затем определить адрес элемента таблицы страниц этого сегмента и извлечь из памяти необходимый элемент и уже только после этого можно к номеру физической страницы приписать номер ячейки в странице (индекс). Задержка доступа к искомой ячейке получается, по крайней мере, в три раза больше, чем при простой прямой адресации. Чтобы избежать этой неприятности, вводится кэширование, причем кэш, как правило, строится по ассоциативному принципу. Другими словами, просмотры двух таблиц в памяти могут быть заменены одним обращением к ассоциативной памяти.

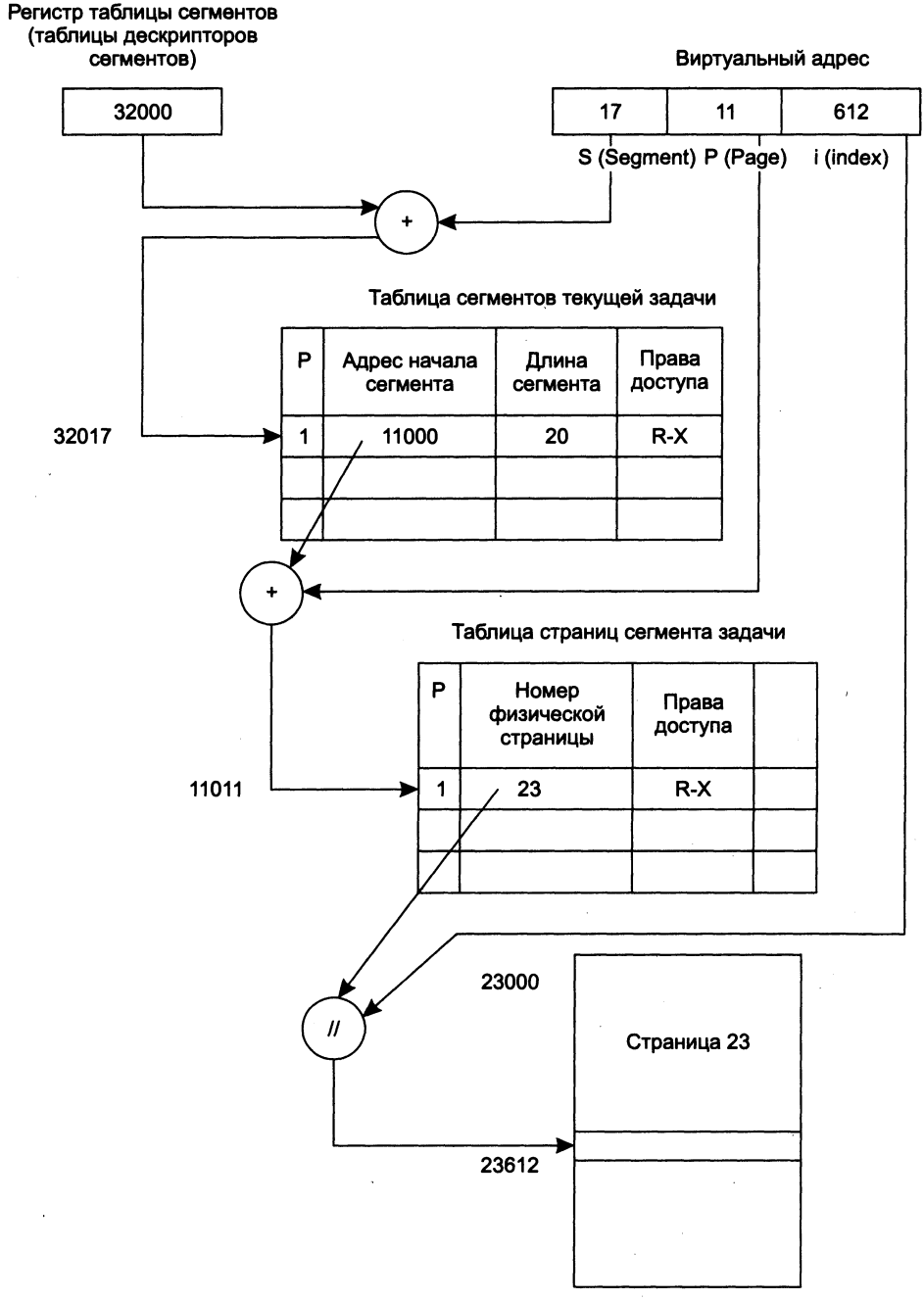


Рис. 3.6. Сегментно-страничный способ организации виртуальной памяти

Напомним, что принцип действия ассоциативного запоминающего устройства предполагает, что каждой ячейке памяти такого устройства ставится в соответ-

ствие ячейка, в которой записывается некий ключ (признак, адрес), позволяющий однозначно идентифицировать содержимое ячейки памяти. Сопутствующую ячейку с информацией, позволяющей идентифицировать основные данные, обычно называют *полем тега*. Просмотр полей тега всех ячеек ассоциативного устройства памяти осуществляется одновременно, то есть в каждой ячейке тега есть необходимая логика, позволяющая посредством побитовой конъюнкции найти данные по их признаку за одно обращение к памяти (если они там, конечно, присутствуют). Часто поле тегов называют аргументом, а поле с данными — функцией. В данном случае в качестве аргумента при доступе к ассоциативной памяти выступают номер сегмента и номер виртуальной страницы, а в качестве функции от этих аргументов получаем номер физической страницы. Остается приписать номер ячейки в странице к полученному номеру, и мы получаем адрес искомой команды или операнда.

Оценим достоинства сегментно-страничного способа. Разбиение программы на сегменты позволяет размещать сегменты в памяти целиком. Сегменты разбиты на страницы, все страницы сегмента загружаются в память. Это позволяет сократить число обращений к отсутствующим страницам, поскольку вероятность выхода за пределы сегмента меньше вероятности выхода за пределы страницы. Страницы исполняемого сегмента находятся в памяти, но при этом они могут находиться не рядом друг с другом, а «россыпью», поскольку диспетчер памяти манипулирует страницами. Наличие сегментов облегчает разделение программных модулей между параллельными процессами. Возможна и динамическая компоновка задачи. А выделение памяти страницами позволяет минимизировать фрагментацию.

Однако поскольку этот способ распределения памяти требует очень значительных затрат вычислительных ресурсов и его не так просто реализовать, используется он редко, причем в дорогих мощных вычислительных системах. Возможность реализовать сегментно-страничное распределение памяти заложена и в семейство микропроцессоров i80x86, однако вследствие слабой аппаратной поддержки, трудностей при создании систем программирования и операционной системы практически в персональных компьютерах эта возможность не используется.

Контрольные вопросы и задачи

1. Что такое «виртуальный адрес», «виртуальное адресное пространство»? Чем (в общем случае) определяется максимально возможный объем виртуального адресного пространства программы?
2. Имеются ли виртуальные адреса в программах, написанных для работы в среде DOS? Приведите примеры абсолютной двоичной программы для таких операционных систем, как MS DOS и Windows NT/2000/XP.
3. Изложите способ распределения памяти в MS DOS.
4. Что дает использование оверлеев при разработке DOS-приложений?
5. Объясните и сравните алгоритмы «первый подходящий», «самый подходящий» и «самый неподходящий», используемые при поиске и выделении фрагмента памяти.

6. Что такое «фрагментация памяти»? Какой метод распределения памяти позволяет добиться минимальной фрагментации и почему?
7. Что такое «уплотнение памяти»? Когда оно применяется?
8. Объясните сегментный способ организации виртуальной памяти. Что представляет собой (в общем случае) дескриптор сегмента?
9. Что представляет собой динамическое присоединение программ? Что оно дает?
10. Сравните сегментный и страничный способы организации виртуальной памяти. Перечислите достоинства и недостатки каждого.
11. Какие дисциплины применяются для решения задачи замещения страниц? Какие из них являются наиболее эффективными и как они реализуются?
12. Что такое «рабочее множество»? Что позволяет разрешить реализация этого понятия?
13. В каких случаях возникает «пробуксовка»? Почему системы Windows NT/2000/XP требуют для своей нормальной работы существенно большего объема оперативной памяти?

Глава 4. Особенности архитектуры микропроцессоров i80x86 для организации мультипрограммных операционных систем

В рамках данной книги мы, естественно, не будем рассматривать все многообразие современных 32-разрядных микропроцессоров, используемых в персональных компьютерах и иных вычислительных системах, а ограничимся рассмотрением только архитектурных, а не технических характеристик микропроцессоров, и под обозначением i80x86 будем понимать любые 32-разрядные микропроцессоры, имеющие основной набор команд такой же, как и в первом 32-разрядном микропроцессоре Intel 80386, и те же архитектурные решения, что и в микропроцессорах фирмы Intel. Нас не будут интересовать новые наборы команд типа MMX или SSE, не будем мы касаться и архитектурных особенностей микропроцессоров, повышающих их производительность. Мы опишем только те механизмы, которые позволяют организовать мультипрограммный и мультизадачный режимы, виртуальную память, обеспечить надежные вычисления.

Реальный и защищенный режимы работы процессора

Широко известно, что первым микропроцессором, на базе которого был создан персональный компьютер IBM PC, был Intel 8088. Этот микропроцессор отличался от первого 16-разрядного микропроцессора фирмы Intel (микропроцессора 8086), прежде всего, тем, что у него была 8-разрядная шина данных, а не 16-разрядная (как у 8086). Оба этих микропроцессора предназначались для создания вы-

числительных устройств, работающих в однозадачном режиме, то есть специальных аппаратных средств для поддержки надежных и эффективных мультипрограммных операционных систем в них не было.

Однако к тому времени, когда разработчики осознали необходимость включения специальной аппаратной поддержки мультипрограммных вычислений, уже было создано очень много программных продуктов. Поэтому для совместимости с первыми компьютерами в последующих версиях микропроцессоров была реализована возможность использовать их в двух режимах: *реальном* (real mode) — так назвали режим работы первых 16-разрядных микропроцессоров — и *защищенном* (protected mode), означающем, что параллельные вычисления могут быть защищены аппаратно-программными механизмами.

Подробно рассматривать архитектуру первых 16-разрядных микропроцессоров i8086/i8088 мы не будем, поскольку этот материал должен изучаться в других дисциплинах. Итак, мы исходим из того, что читатель знаком с архитектурой процессора i8086/i8088 и с программированием на ассемблере для этих 16-разрядных процессоров Intel. Для тех же, кто с ней незнаком, можно рекомендовать, например, такие книги, как [12, 24, 40] и многие другие. Однако мы напомним, что в этих микропроцессорах (а значит, и в остальных микропроцессорах семейства i80x86 при работе их в реальном режиме) обращение к памяти с возможным адресным пространством в 1 Мбайт осуществляется посредством механизма *сегментной адресации* (рис. 4.1). Этот механизм был использован для того, чтобы увеличить с 16 до 20 количество разрядов, участвующих в формировании адреса ячейки памяти, по которому идет обращение, и тем самым увеличить доступный объем памяти.

Рис. 4.1. Схема определения физического адреса для процессора 8086

Для конкретности будем рассматривать определение адреса команд, хотя для адресации операндов используется аналогичный механизм, только участвуют в этом случае другие сегментные регистры. Напомним, что для определения физического адреса команды содержимое регистра сегмента кода (Code Segment, CS) умножается на 16 за счет добавления справа (к младшим битам) четырех нулей, после чего к полученному значению прибавляется содержимое регистра указателя ко-

манд (Instruction Pointer, IP). Получается 20-разрядное значение[1], которое и позволяет указать любой байт из 20^{20}.

В защищенном режиме работы определение физического адреса осуществляется совершенно иначе. Прежде всего, используется сегментный механизм для организации виртуальной памяти. При этом адреса задаются 32-разрядными значениями. Кроме этого, возможна страничная трансляция адресов, также с 32-разрядными значениями. Наконец, при работе в защищенном режиме, который по умолчанию предполагает 32-разрядный код, возможно исполнение двоичных программ, созданных для работы микропроцессора в 16-разрядном режиме. Для этого введен режим виртуальной 16-разрядной машины, и 20-разрядные адреса реального режима транслируются с помощью страничного механизма в 32-разрядные значения защищенного режима. Наконец, есть еще один режим — 16-разрядный защищенный, позволяющий 32-разрядным микропроцессорам выполнять защищенный 16-разрядный код, который был характерен для микропроцессора 80286. Правда, следует отметить, что этот последний режим практически не используется, поскольку программ, созданных для него, не так уж и много.

Для изучения этих возможностей рассмотрим сначала новые архитектурные возможности микропроцессоров i80x86.

Новые системные регистры микропроцессоров i80x86

Основные регистры микропроцессора i80x86, знание которых необходимо для понимания защищенного режима работы, приведены на рис. 4.2. На этом рисунке следует обратить внимание на следующее:

- указатель команды (EIP) — это 32-разрядный регистр, младшие 16 разрядов которого представляют регистр IP;
- регистр флагов (EFLAGS) — это 32-разрядный регистр, младшие 16 разрядов которого представляют регистр FLAGS;
- регистры общего назначения EAX, EBX, ECX, EDX, а также регистры ESP, EBP, ESI, EDI 32-разрядные, однако их младшие 16 разрядов представляют собой известные регистры AX, BX, CX,DX, SP, BP, SI, DI;
- сегментные регистры CS, SS, DS, ES, FS, GS 16-разрядные, при каждом из них пунктиром изображены скрытые от программистов (недоступные никому, кроме собственно микропроцессора) 64-разрядные регистры, в которые загружаются дескрипторы соответствующих сегментов;

[1] На самом деле, поскольку происходит именно сложение и каждое из слагаемых может иметь значение в интервале от нуля до $2^{16} - 1 = 65\,535 = 64$ Кбайт, мы можем указать адрес начала сегмента, равный FFFFFFFF00H, и к нему прибавить смещение FFFFFFFFH. В этом случае мы получим переполнение разрядной сетки, но для современных 32-разрядных процессоров (и для уже забытого i80286) имеется возможность указать первые 64 Кбайт выше первого мегабайта.

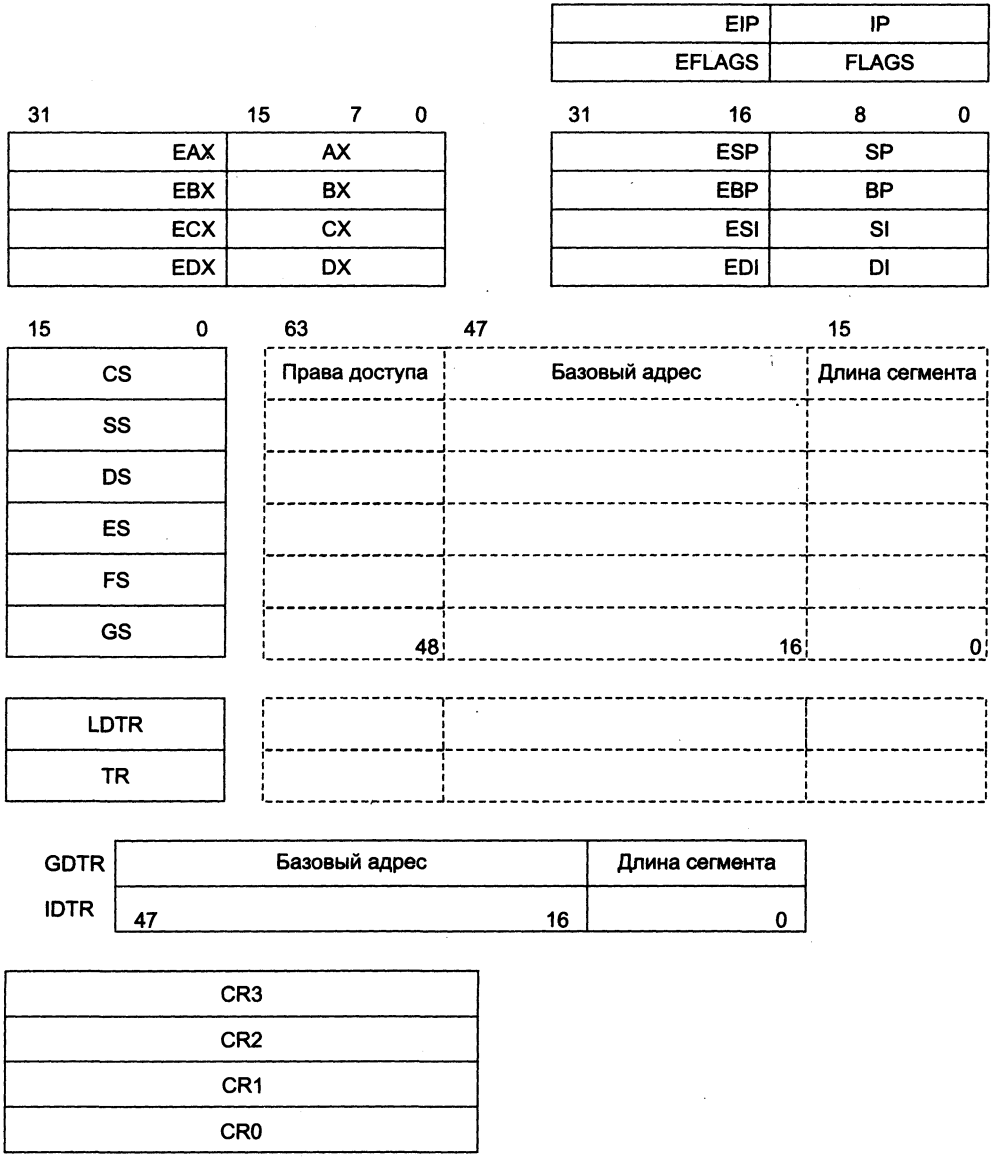


Рис. 4.2. Основные системные регистры микропроцессоров i80x86

- при 16-разрядном регистре-указателе на локальную таблицу дескрипторов (Local Descriptor Table Register, LDTR) также имеется «теневой» (скрытый от программиста) 64-разрядный регистр, в который микропроцессор заносит дескриптор, указывающий на таблицу дескрипторов сегментов задачи, описывающих ее локальное виртуальное адресное пространство;
- 16-разрядный регистр задачи (Task Register, TR) указывает на дескриптор в глобальной таблице дескрипторов, который позволяет получить доступ к дескрип-

тору *сегмента состояния задачи* (Task State Segment, TSS) — информационной структуре, которую поддерживает микропроцессор для управления задачами;

- 48-разрядный регистр GDTR (Global Descriptor Table Register) глобальной таблицы дескрипторов (Global Descriptor Table, GDT) содержит как дескрипторы общих сегментов, так и специальные системные дескрипторы, в частности, в GDT находятся дескрипторы, с помощью которых можно получить доступ к сегментам TSS;
- 48-разрядный регистр таблицы дескрипторов прерываний (IDTR) содержит информацию, необходимую для доступа к таблице прерываний (IDT);
- 32-разрядные регистры CR0–CR3 являются управляющими.

Помимо перечисленных имеются и некоторые другие регистры.

Управляющий регистр CR0 содержит целый ряд флагов, которые определяют режимы работы микропроцессора. Подробности об этих флагах можно найти, например, в [1, 8, 20]. Мы же просто ограничимся тем фактом, что самый младший бит PE (Protect Enable) этого регистра определяет режим работы процессора. При PE = 0 процессор функционирует в реальном режиме работы, а при единичном значении микропроцессор переключается в защищенный режим. Самый старший бит регистра CR0 — бит PG (PaGing) — определяет, включен (PG = 1) или нет (PG = 0) режим страничного преобразования адресов.

Регистр CR2 предназначен для размещения в нем адреса подпрограммы обработки страничного исключения, то есть в случае страничного механизма отображения памяти обращение к отсутствующей странице будет вызывать переход на соответствующую подпрограмму диспетчера памяти, и для определения этой подпрограммы потребуется регистр CR2.

Регистр CR3 содержит номер физической страницы, в которой располагается *таблица каталога таблиц страниц* текущей задачи. Очевидно, что, приписав к этому номеру нули, мы попадем на начало этой страницы.

Адресация в 32-разрядных микропроцессорах i80x86 при работе в защищенном режиме

Поддержка сегментного способа организации виртуальной памяти

Как мы уже знаем, для организации эффективной и надежной работы вычислительной системы в мультипрограммном режиме необходимо иметь соответствующие аппаратные механизмы, поддерживающие независимость адресных пространств каждой задачи и в то же время позволяющие организовать обмен данными и разделение кода. Для этого желательно выполнить следующие два требования:

- чтобы у каждого вычислительного процесса было собственное (личное, локальное) адресное пространство, никак не пересекающееся с адресными пространствами других процессов;
- чтобы существовало общее (разделяемое) адресное пространство.

Для удовлетворения этих требований в микропроцессорах i80x86 реализован сегментный способ распределения памяти. Помимо того в этих микропроцессорах может быть задействована и страничная трансляция. Поскольку для каждого сегмента нужен дескриптор, устройство управления памятью поддерживает соответствующую информационную структуру. Формат *дескриптора сегмента* приведен на рис. 4.3.

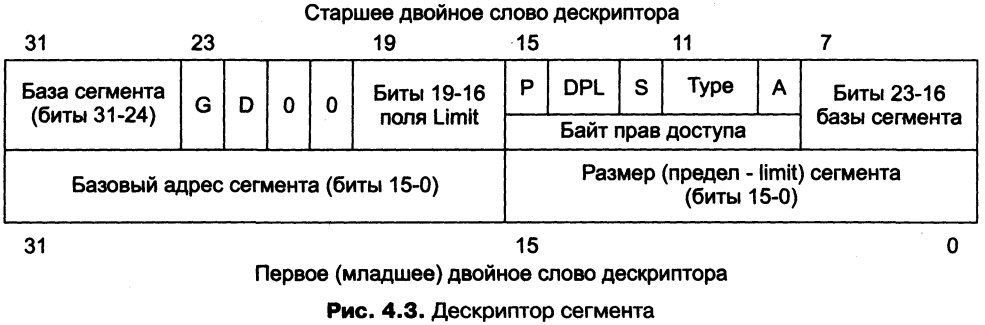


Рис. 4.3. Дескриптор сегмента

Поля дескриптора (базовый адрес, поле предела) размещены в дескрипторе не непрерывно, а в разбивку, потому что, во-первых, разработчики постарались минимизировать количество перекрестных соединений в полупроводниковой структуре микропроцессора, а во-вторых, чтобы обеспечить полную совместимость[1] микропроцессоров (предыдущий микропроцессор i80286 работал с 16-разрядным кодом и тоже поддерживал сегментный механизм реализации виртуальной памяти). Необходимо заметить, что формат дескриптора сегмента, изображенный на рис. 4.3, справедлив только для случая нахождения соответствующего сегмента в оперативной памяти. Если же бит присутствия в поле прав доступа равен нулю (сегмент отсутствует в памяти), то все биты, за исключением поля прав доступа, считаются неопределенными и могут использоваться системными программистами (для указания адреса сегмента во внешней памяти) произвольным образом.

Локальное адресное пространство задачи определяется через таблицу LDT (Local Descriptor Table). У каждой задачи может быть свое локальное адресное пространство. *Общее*, или *глобальное, адресное пространство* определяется через таблицу GDT (Global Descriptor Table). Само собой, что работу с этими таблицами (их заполнение и последующую модификацию) должна осуществлять операционная система. Доступ к таблицам LDT и GDT со стороны прикладных задач должен быть исключен.

При переключении микропроцессора в защищенный режим он начинает совершенно другим образом, чем в реальном режиме, вычислять физические адреса команд и

[1] Естественно, совместимость обеспечена только «снизу вверх», то есть программы, разработанные для предыдущих версий микропроцессора, должны выполняться на последующих без какой-либо переделки.

операндов. Прежде всего, содержимое сегментных регистров начинает интерпретироваться иначе: считается, что там содержится не адрес начала, а номер соответствующего сегмента. Для того чтобы подчеркнуть этот факт, сегментные регистры CS, SS, DS, ES, FS, GS начинают даже называть иначе — *селекторами сегментов*. При этом каждый селекторный регистр разбивается на три поля (рис. 4.4).

- Поле индекса (Index) — старшие 13 битов (3–15) определяет собственно номер сегмента (его индекс в соответствующей таблице дескрипторов).
- Поле индикатора таблицы сегментов (Table Index, TI) — бит с номером 2 определяет часть виртуального адресного пространства (общее или принадлежащее только данной задаче). Если TI = 0, то поле индекса указывает на элемент в глобальной таблице дескрипторов (GDT), то есть идет обращение к общей памяти. Если TI = 1, то идет обращение к локальной области памяти текущей задачи; это пространство описывается локальной таблицей дескрипторов (LDT).
- Поле уровня привилегий идентифицирует *запрашиваемый уровень привилегий* (Requested Privilege Level, RPL).

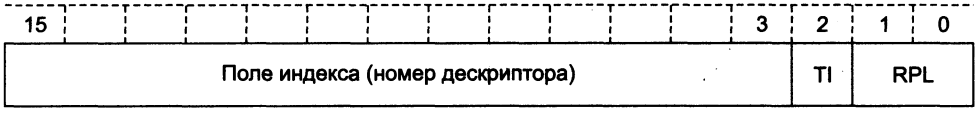


Рис. 4.4. Селектор сегмента

Операционная система в процессе своего запуска инициализирует многие регистры, и прежде всего GDTR. Этот регистр содержит начальный адрес глобальной таблицы дескрипторов (GDT) и ее размер. Как мы уже знаем, в GDT содержатся дескрипторы глобальных сегментов и системные дескрипторы.

Для манипулирования задачами операционные системы поддерживают информационную структуру, которую мы уже раньше называли как *дескриптор задачи* (см. главу 1). Микропроцессоры с архитектурой IA32 поддерживают работу с наиболее важной частью дескриптора задачи, которая меньше всего зависит от операционной системы. Эта инвариантная часть дескриптора, с которой и работает микропроцессор, названа *сегментом состояния задачи* (Task State Segment, TSS). Перечень полей TSS показан на рис. 4.5. Видно, что этот сегмент содержит в основном контекст задачи. Процессор получает доступ к этой структуре с помощью регистра задачи (Task Register, TR).

Регистр TR содержит индекс (селектор) элемента в GDT. Этот элемент представляет собой дескриптор сегмента TSS. Дескриптор заносится в теневую часть регистра (см. рис. 4.2). К рассмотрению TSS мы еще вернемся, а сейчас заметим, что в одном из полей TSS содержится указатель (селектор) на локальную таблицу дескрипторов данной задачи. При переходе процессора с одной задачи на другую содержимое поля LDTR заносится микропроцессором в одноименный регистр. Инициализировать регистр TR можно и явным образом.

Итак, регистр LDTR содержит селектор, указывающий на один из дескрипторов таблицы GDT. Этот дескриптор заносится микропроцессором в теневую часть регистра LDTR и описывает таблицу LDT для текущей задачи. Теперь, когда у нас

определены как глобальная, так и локальная таблица дескрипторов, можно рассмотреть процесс определения линейного адреса[1]. Для примера рассмотрим процесс получения адреса команды. Адреса операндов определяются по аналогии, но задействованы будут другие регистры.

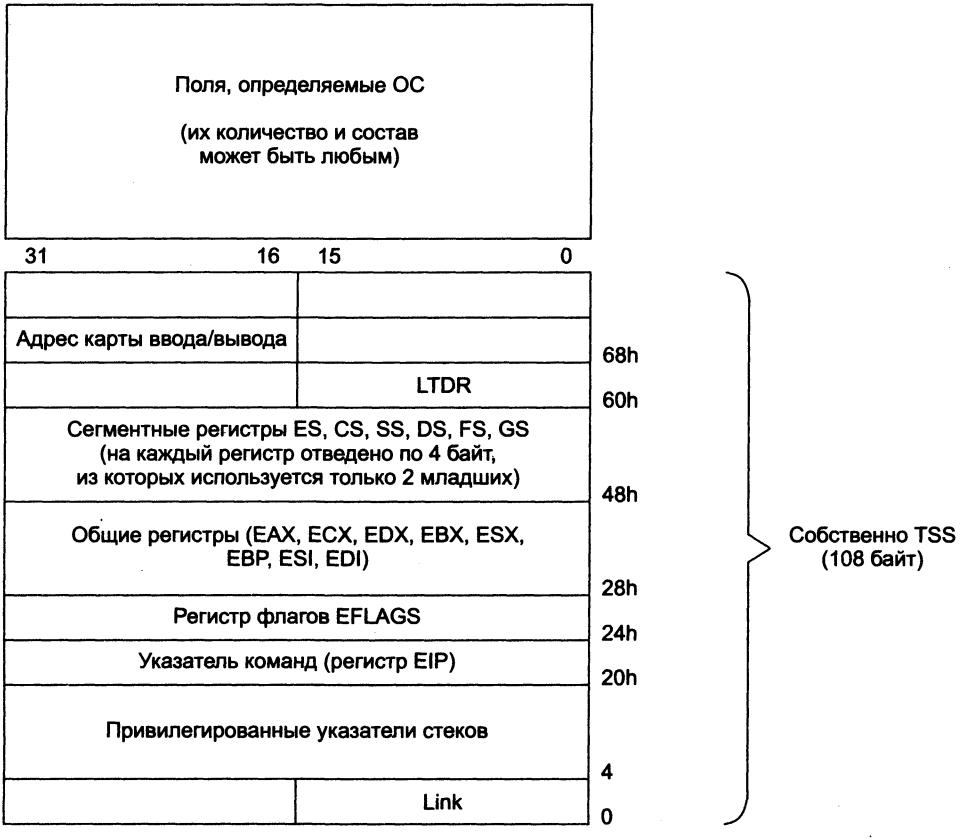


Рис. 4.5. Сегмент состояния задачи

Микропроцессор анализирует бит TI селектора кода и, в зависимости от его значения, извлекает из таблицы GDT или LDT дескриптор сегмента кода с номером (индексом), который равен полю индекса (биты 3–15 селектора на рис. 4.4). Этот дескриптор заносится в теневую (скрытую) часть регистра CS. Далее микропроцессор сравнивает значение регистра EIP (Extended Instruction Pointer — расширенный указатель команды) с полем размера сегмента, содержащегося в извлеченном дескрипторе, и если смещение относительно начала сегмента не превышает размера предела, то значение EIP прибавляется к значению поля начала сегмента, и мы получаем искомый линейный адрес команды. *Линейный адрес* — это одна из форм виртуального адреса. Исходный двоичный виртуальный адрес, вычисляе-

[1] В микропроцессорах i80x86 линейным называется адрес, полученный в результате преобразования виртуального адреса формата (S, d) в 32-разрядный адрес.

мый в соответствии с используемой схемой адресации, преобразуется в линейный. В свою очередь, линейный адрес будет либо равен физическому (если страничное преобразование отключено), либо путем страничной трансляции преобразуется в физический адрес. Если же смещение из регистра EIP превышает размер сегмента кода, то эта аварийная ситуация вызывает прерывание, и управление должно передаваться супервизору операционной системы.

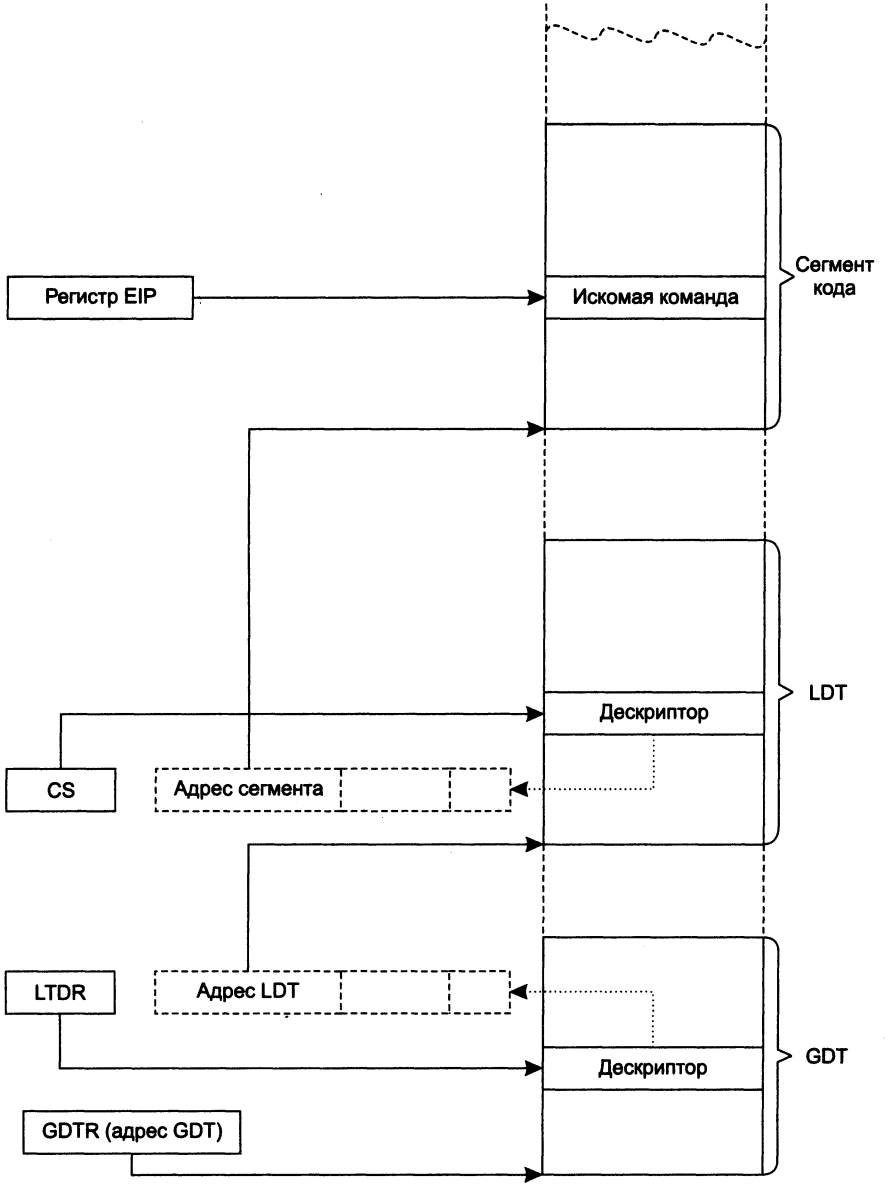


Рис. 4.6. Процесс получения линейного адреса команды

Рассмотренный нами процесс получения линейного адреса иллюстрирует рис. 4.6. Стоит отметить, что поскольку межсегментные переходы происходят нечасто, то, как правило, определение линейного адреса заключается только в сравнении значения EIP с полем предела сегмента и в прибавлении смещения к началу сегмента. Все необходимые данные уже находятся в микропроцессоре, и операция получения линейного адреса происходит очень быстро.

Итак, линейный адрес может считаться физическим адресом, если не включен режим страничной трансляции адресов. К сожалению, аппаратных средств микропроцессора для поддержки рассмотренного способа двойной трансляции виртуальных адресов в физические явно недостаточно. При наличии большого количества небольших сегментов, из которых состоят программы, это приводит к заметному замедлению в работе процессора. В самом деле, теневой регистр при каждом селекторе имеется в единственном экземпляре, и при переходе на другой сегмент требуется вновь находить и извлекать соответствующий дескриптор сегмента, а это отнимает время. Страничный же способ трансляции виртуальных адресов, как мы знаем, имеет немало достоинств. Поэтому в защищенном режиме работы, при котором всегда действует описанный выше механизм определения линейных адресов, может быть включен еще и страничный механизм.

Поддержка страничного способа организации виртуальной памяти

При создании микропроцессора i80386 разработчики столкнулись с очень серьезной проблемой в реализации страничного механизма. Дело в том, что микропроцессор имел широкую шину адреса (32 бит), и возник вопрос о разбиении всего адреса на поле страницы и поле индекса. Если большое количество битов адреса отвести под индекс, то страницы станут очень большими, что повлечет значительные потери и на фрагментацию, и на операции ввода-вывода, связанные с замещением страниц. Хотя количество страниц стало бы при этом меньше, и накладные расходы на их поддержание тоже уменьшились бы. Если же размер страницы уменьшить, то большое поле номера страницы привело бы к потенциально громадному количеству страниц, и пришлось бы либо вводить какие-то механизмы контроля за номерами страниц (с тем, чтобы они не выходили за размеры таблицы страниц), либо создавать эти таблицы максимального размера. Разработчики пошли по пути, при котором размер страницы выбран небольшим (2^{12} = 4096 = 4 Кбайт), а поле номера страницы величиной в 20 бит, в свою очередь, разбивается на два поля и осуществляется двухэтапная страничная трансляция.

Для описания каждой страницы создается соответствующий дескриптор. Длина дескриптора выбрана равной 32 бит: 20 бит линейного адреса определяют номер страницы (по существу — ее адрес, поскольку добавление к нему 12 нулей приводит к определению начального адреса страницы), а остальные биты разбиты на поля, показанные на рис. 4.7. Как видно, три бита дескриптора зарезервировано для использования системными программистами при разработке подсистемы организации виртуальной памяти. С этими битами микропроцессор сам не работает.

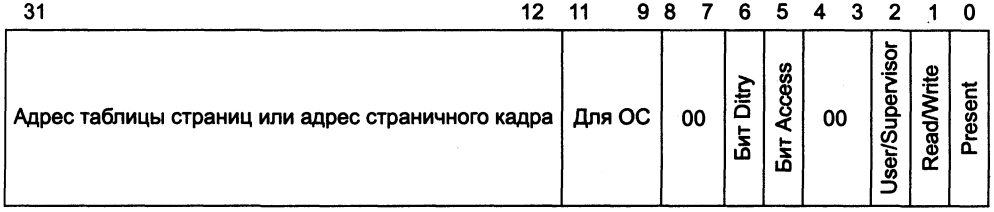


Рис. 4.7. Дескриптор страницы

Прежде всего, микропроцессор анализирует самый младший бит дескриптора — *бит присутствия*, если он равен нулю, то это означает отсутствие данной страницы в оперативной памяти, и такая ситуация влечет прерывание в работе процессора с передачей управления на соответствующую программу, которая должна будет загрузить затребованную страницу. Бит, называемый «грязным» (dirty), показывает, что данную страницу модифицировали, и при замещении этого страничного кадра следующим ее необходимо сохранить во внешней памяти. *Бит обращения* (access) свидетельствует о том, что к данной таблице или странице осуществлялся доступ. Он анализируется для определения страницы, которая будет участвовать в замещении при использовании дисциплины LRU или LFU. Наконец, первый и второй биты требуются для защиты памяти.

Старшие 10 бит линейного адреса определяют номер *таблицы страниц* (Page Table Entry, PTE), из которой посредством вторых 10 бит линейного адреса выбирается соответствующий дескриптор виртуальной страницы. И уже из этого дескриптора выбирается номер физической страницы, если данная виртуальная страница отображена на оперативную память. Эта схема определения физического адреса из линейного изображена на рис. 4.8.

Первая таблица, которую мы индексируем первыми (старшими) десятью битами линейного адреса, названа *таблицей каталога таблиц страниц* (Page Directory Entry, PDE). Ее адрес в оперативной памяти определяется старшими двадцатью битами управляющего регистра CR0.

Каждая из таблиц (PDE и PTE) состоит из 1024 элементов ($2^{10} = 1024$). В свою очередь, каждый элемент (дескриптор страницы) имеет длину 4 байт (32 бит), поэтому размер этих таблиц как раз соответствует размеру страницы.

Оценим теперь эту схему трансляции с позиций расхода памяти. Каждый дескриптор описывает страницу размером 4 Кбайт. Следовательно, одна таблица страниц, содержащая 1024 дескриптора, описывает пространство памяти в 4 Мбайт. Если задача пользуется виртуальным адресным пространством, например, в 55 Мбайт (предположим, что речь идет о некотором графическом редакторе, который обрабатывает изображение, состоящее из большого количества пикселов[1]), то для описания этой памяти необходимо иметь 14 страниц (14 × 4 Мбайт = 56 Мбайт), содержащих таблицы PTE. Кроме того, нам потребуется для этой задачи еще одна таблица PDE (тоже размером в одну страницу), в которой 14 дескрипторов будут указывать на место-

[1] Напомним, что термин «пиксел» происходит от английского Picture Element — графический элемент. Множество пикселов образуют изображение.

нахождение упомянутых таблиц PTE. Остальные дескрипторы PDE не требуются. Итого, для описания 55 Мбайт адресного пространства задачи потребуется всего 15 страниц, то есть 60 Кбайт памяти, что можно считать приемлемым.

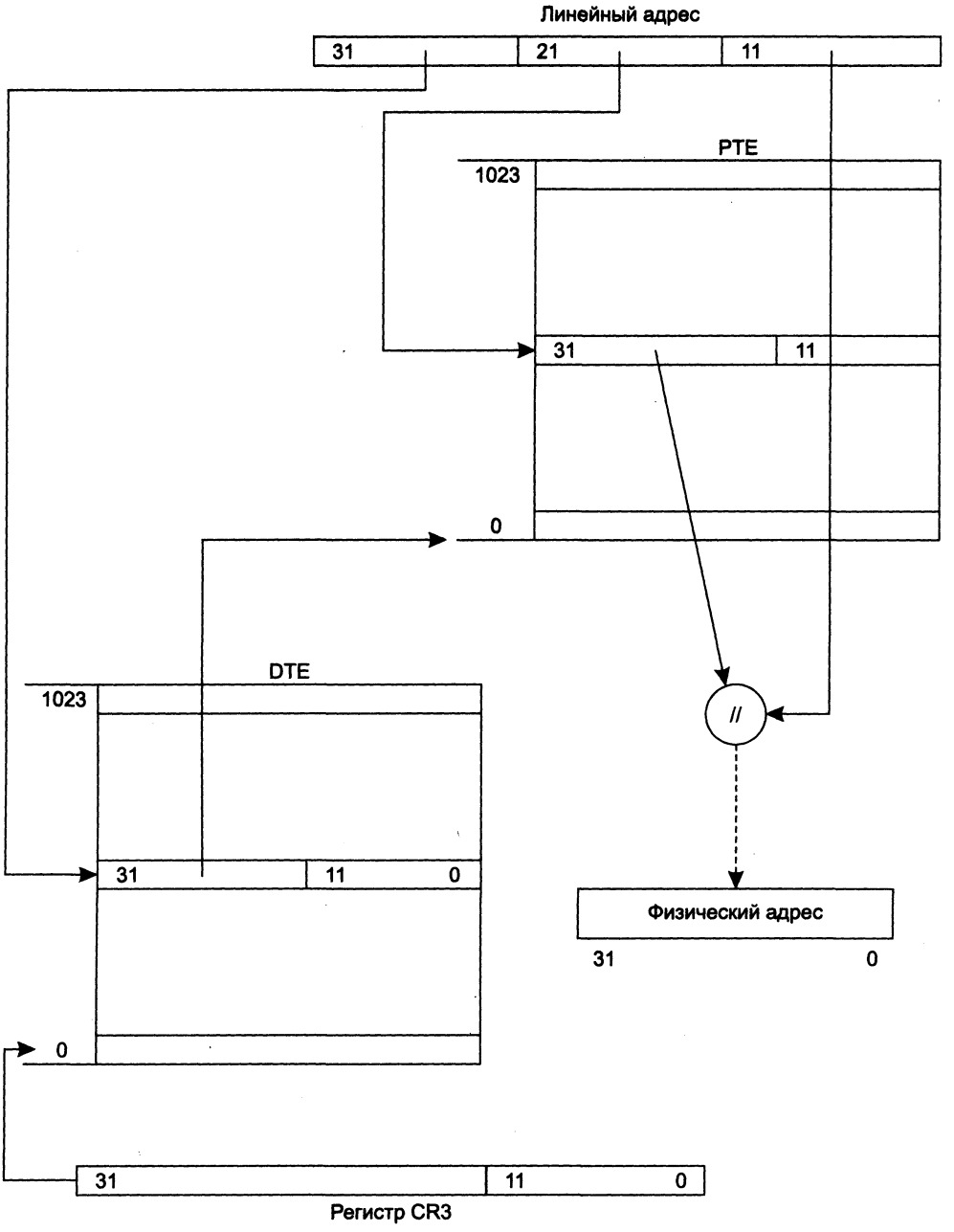


Рис. 4.8. Трансляция линейного адреса в микропроцессорах i80x86

Если бы не был использован такой двухэтапный механизм трансляции, то потери памяти на описание адресного пространства могли бы составить 4 Кбайт × 2^{10} = 4 Мбайт! Очевидно, что это уже неприемлемое решение.

Итак, микропроцессор для каждой задачи, для которой у него есть TSS, позволяет иметь таблицу PDE и некоторое количество таблиц PTE. Поскольку это дает возможность адресоваться к любому байту из 2^{32}, а шина адреса как раз и позволяет использовать физическую память с таким объемом, то можно как бы отказаться от сегментного способа адресации. Другими словами, если считать, что задача состоит из одного единственного сегмента кода и одного сегмента данных, которые, в свою очередь, разбиты на страницы, то фактически мы получаем только один страничный механизм работы с виртуальной памятью. Этот подход получил название *плоской модели памяти*. При использовании плоской модели памяти упрощается создание и операционных систем, и систем программирования, кроме того, уменьшаются расходы памяти на поддержку системных информационных структур. Поэтому в абсолютном большинстве современных 32-разрядных операционных систем, создаваемых для микропроцессоров i80x86, используется плоская модель памяти. Более того, появление новых 64-разрядных микропроцессоров во многом определено желанием получить большее адресное пространство, чем его имеют 32-разрядные процессоры, при сохранении возможности работать только с плоской моделью памяти.

Режим виртуальных машин для исполнения приложений реального режима

Разработчики рассматриваемого семейства микропроцессоров в своем стремлении обеспечить максимально возможную совместимость архитектуры пошли не только на то, чтобы за счет введения реального режима работы обеспечить возможность программам, созданным для первых 16-разрядных персональных компьютеров, без проблем выполняться на компьютерах с более поздними моделями микропроцессоров. Они обеспечили возможность выполнения 16-разрядных приложений реального режима при условии, что сам процессор функционирует в защищенном режиме работы, и операционная система, используя соответствующие аппаратные средства микропроцессора, организует мультипрограммный (мультизадачный) режим. Другими словами, микропроцессоры i80x86 поддерживают возможность создания операционных сред реального режима при работе микропроцессора в защищенном режиме. Если условно назвать 16-разрядные приложения DOS-приложениями (поскольку в абсолютном большинстве случаев это именно так), то можно сказать, что введена поддержка виртуальных DOS-машин, работающих вместе с обычными 32-разрядными приложениями защищенного режима. Это нашло отражение в названии такого режима работы микропроцессоров i80x86 (его называют режимом *виртуального процессора i8086*, иногда для краткости — *режимом V86*, или просто *виртуальным режимом*), когда в защищенном режиме работы может исполняться код DOS-приложения. Мультизадачность при выполнении нескольких программ реального режима поддерживается аппаратными средствами защищенного режима.

Переход в виртуальный режим осуществляется посредством изменения бита VM (Virtual Mode) в регистре EFLAGS. Когда процессор находится в виртуальном режиме, для адресации памяти используется схема реального режима работы (сегмент плюс смещение) с размером сегментов до 64 Кбайт, которые могут располагаться в адресном пространстве размером в 1 Мбайт, однако полученные адреса считаются не физическими, а линейными. В результате страничной трансляции осуществляется отображение виртуального адресного пространства 16-разрядного приложения на физическое адресное пространство. Это позволяет организовать параллельное выполнение нескольких задач, разработанных для реального режима, да еще совместно с обычными 32-разрядными приложениями, требующими защищенного режима работы.

Естественно, что для обработки прерываний, возникающих при выполнении 16-разрядных приложений в виртуальном режиме, процессор возвращается из этого режима в обычный защищенный режим. В противном случае невозможно было бы организовать полноценную виртуальную машину. Очевидно, что обработчики прерываний для виртуальной машины должны эмулировать работу подсистемы прерываний процессора i8086. Другими словами, прерывания отображаются в операционную систему, работающую в защищенном режиме, и уже основная операционная система моделирует работу операционной среды выполняемого приложения.

Вопрос, связанный с операциями ввода-вывода, которые недоступны для обычных приложений (см. следующий раздел), решается аналогично. При попытке выполнить недопустимые команды (ввода-вывода) возникают прерывания, и необходимые операции выполняются операционной системой, хотя задача об этом и «не подозревает». При выполнении команд `IN`, `OUT`, `INS`, `OUTS`, `CLI`, `STI` процессор, находящийся в виртуальном режиме и исполняющий код на уровне привилегий третьего (самого нижнего) кольца защиты, за счет возникающих вследствие этого прерываний переводится на выполнение высоко привилегированного кода операционной системы.

Таким образом, операционная система может полностью виртуализировать аппаратные[1] и программные ресурсы компьютера, создавая полноценную операционную среду, отличную от себя самой, ибо существуют так называемые нативные приложения, создаваемые по собственным спецификациям данной операционной системы. Очень важным моментом для организации полноценной виртуальной машины является виртуализация не только программных, но и аппаратных ресурсов. Так, например, в Windows NT эта задача выполнена явно неудачно, тогда как в OS/2 имеется полноценная виртуальная машина как для DOS-приложений, так и для приложений, работающих в среде спецификаций Win16. Правда, в последнее время это перестало быть актуальным, поскольку появилось большое количество приложений, работающих по спецификациям Win32 API.

[1] Речь идет о памяти, портах ввода-вывода, системе обработки прерываний и других устройствах.

Защита адресного пространства задач

Для создания надежных мультипрограммных операционных систем в процессорах семейства i80x86 имеется несколько механизмов защиты. Это и разделение адресных пространств задач, и введение уровней привилегий для сегментов кода и сегментов данных. Это позволяет обеспечить как защиту задач друг от друга, так и защиту самой операционной системы от прикладных задач, защиту одной части системы от других ее частей, защиту самих задач от некоторых своих собственных ошибок.

Защита адресного пространства задач осуществляется относительно легко за счет того, что каждая задача может иметь свое собственное локальное адресное пространство. Операционная система должна корректно манипулировать таблицами трансляции сегментов (дескрипторными таблицами) и таблицами трансляции страничных кадров. Сами таблицы дескрипторов как сегменты данных (а соответственно, в свою очередь, и как страничные кадры) относятся к адресному пространству операционной системы и имеют соответствующие привилегии доступа; исправлять их задачи не могут. Этими информационными структурами процессор пользуется сам на аппаратном уровне без возможности их читать и редактировать из пользовательских приложений. В плоской модели памяти возможность микропроцессора контролировать обращения к памяти только внутри текущего сегмента фактически не используется, и остается в основном только механизм отображения страничных кадров. Выход за пределы страничного кадра невозможен, поэтому фиксируется только выход за пределы своего сегмента. В этом случае приходится полагаться только на систему программирования, которая должна корректно распределять программные модули в пределах единого неструктурированного адресного пространства задачи. Поэтому создание многопоточных приложений, когда каждая задача (в данном случае — поток выполнения) может испортить адресное пространство другой задачи, — очень сложная проблема, особенно если не применять системы программирования на языках высокого уровня.

Итак, чтобы организовать взаимодействие задач, имеющих разные виртуальные адресные пространства, необходимо, как мы уже говорили, иметь общее адресное пространство. И здесь для обеспечения защиты самой операционной системы, а значит, и для повышения надежности всех вычислений используется механизм защиты сегментов с помощью уровней привилегий.

Уровни привилегий для защиты адресного пространства задач

Для того чтобы запретить пользовательским задачам модифицировать области памяти, принадлежащие самой операционной системе, необходимо иметь специальные средства. Одного разграничения адресных пространств через механизм сегментов мало, ибо можно указывать различные значения адреса начала сегмента и тем самым получать доступ к чужим сегментам. Другими словами, необходимо в явном виде отделять системные сегменты данных и кода от сегментов, принадлежащих пользовательским программам. Поэтому были введены два основных ре-

жима работы процессора: режим пользователя и режим супервизора. Большинство современных процессоров поддерживают по крайней мере два этих режима. Так, в *режиме супервизора* программа может выполнять все действия и иметь доступ по любым адресам, тогда как в *пользовательском режиме* должны быть ограничения, с тем чтобы обнаруживать и пресекать запрещенные действия, перехватывая их и передавая управление супервизору операционной системы. Часто в пользовательском режиме запрещается выполнение команд ввода-вывода и некоторых других, чтобы гарантировать выполнение этих операций только операционной системой.

В микропроцессорах i80x86 режим супервизора и режим пользователя непосредственно связаны с так называемыми *уровнями привилегий*, причем имеется не два, а четыре уровня привилегий. Для указания уровня привилегий используются два бита, поэтому код 0 обозначает самый высший уровень, а код 3 — самый низший. Самый высший уровень привилегий предназначен для операционной системы (прежде всего для ядра ОС), самый низший — для прикладных задач пользователя. Промежуточные уровни привилегий введены для большей свободы системных программистов в организации надежных вычислений при создании операционной системы и иного системного программного обеспечения. Предполагалось, что уровень с номером (кодом) 1 может быть использован, например, для системного сервиса — программ обслуживания аппаратуры, драйверов, работающих с портами ввода-вывода. Уровень привилегий с кодом 2 может быть использован для создания пользовательских интерфейсов, систем управления базами данных и прочими, то есть для реализации специальных системных функций, которые по отношению к супервизору операционной системы ведут себя как обычные приложения. Так, например, в системе OS/2 доступны три уровня привилегий: с нулевым уровнем привилегий исполняется код супервизорной части операционной системы, на втором уровне исполняются системные процедуры подсистемы ввода-вывода, на третьем уровне исполняются прикладные задачи пользователей. Однако на практике чаще всего задействуются только два уровня — нулевой и третий. Таким образом, упомянутый режим супервизора для микропроцессоров i80x86 соответствует выполнению кода с уровнем привилегий 0, обозначаемый как PL0 (Privilege Level 0 — уровень привилегий 0). Подводя итог, можно констатировать, что именно уровень привилегий задач определяет, какие команды в них можно использовать и какое подмножество сегментов и/или страниц в их адресном пространстве они могут обрабатывать.

Основными системными объектами, которыми манипулирует процессор при работе в защищенном режиме, являются дескрипторы. Именно дескрипторы сегментов содержат информацию об уровне привилегий соответствующего сегмента кода или данных. Уровень привилегий исполняющейся задачи определяется значением поля привилегий, находящегося в дескрипторе ее текущего кодового сегмента. Напомним (см. рис. 4.3), что в байте прав доступа каждого дескриптора сегмента имеется поле DPL (Descriptor Privilege Level — уровень привилегий сегмента, определяемый его дескриптором), которое и определяет уровень привилегий связанного с ним сегмента. Таким образом, поле DPL текущего сегмента кода становится полем текущего уровня привилегий (Current Privilege Level, CPL), или *уровня при-*

вилегий задачи. При обращении к какому-нибудь сегменту в соответствующем селекторе указывается (см. рис. 4.4) *запрашиваемый уровень привилегий* (Requested Privilege Level, RPL)[1].

В пределах одной задачи используются сегменты с различными уровнями привилегий, и в определенные моменты времени выполняются или обрабатываются сегменты с соответствующими им уровнями привилегий. Механизм проверки привилегий работает в ситуациях, которые можно назвать *межсегментными переходами* (обращениями). К этим ситуациям относятся доступ к сегменту данных или стековому сегменту, межсегментные передачи управления в случае прерываний (и особых ситуаций), использование команд CALL, JMP, INT, IRET, RET. В таких межсегментных обращениях участвуют два сегмента: целевой сегмент (к которому мы обращаемся) и текущий сегмент кода, из которого идет обращение.

Процессор сравнивает упомянутые значения CPL, RPL, DPL и на основе понятия *эффективного уровня привилегий* (Effective Privilege Level, EPL)[2] ограничивает возможности доступа к сегментам по следующим правилам, в зависимости от того, идет ли речь об обращении к коду или к данным.

При доступе к сегментам данных проверяется условие CPL ≤ EPL. Нарушение этого условия вызывает так называемую особую ситуацию ошибки защиты, ведущую к прерыванию. Уровень привилегий сегмента данных, к которому осуществляется обращение, должен быть таким же, как и текущий уровень, или меньше его. Обращение к сегменту с более высоким уровнем привилегий воспринимается как ошибка, так как существует опасность изменения данных с высоким уровнем привилегий программой с низким уровнем привилегий. Доступ к данным с меньшим уровнем привилегий разрешается.

Если целевой сегмент является сегментом стека, то правило проверки имеет вид:

$$CPL = DPL = RPL$$

В случае его нарушения также возникает исключение. Поскольку стек может применяться в каждом сегменте кода, и всего имеется четыре уровня привилегий кода, используется четыре стека. Сегмент стека, адресуемый регистром SS, должен иметь тот же уровень привилегий, что и текущий сегмент кода.

Правила для передачи управления, когда осуществляется межсегментный переход с одного сегмента кода на другой сегмент кода, несколько сложнее. Если для перехода с одного сегмента данных на другой сегмент данных считается допустимым обрабатывать менее привилегированные сегменты, то передача управления из более привилегированного кода на менее привилегированный код должна контролироваться дополнительно. Другими словами, код операционной системы не должен доверять коду прикладных задач. И обратно, нельзя просто так давать задачам возможность исполнять привилегированный код, хотя потребность в этом всегда имеется (ведь многие функции, в том числе и функции ввода-вывода, счи-

[1] Поле RPL определяется программистом (системой программирования). В отличие от поля DPL поле RPL легко может быть изменено.

[2] Значение эффективного уровня привилегий определяется минимальной привилегией, то есть как максимальное значение из двух уровней, RPL и DPL.

таются привилегированными и должны выполняться только самой операционной системой). Для передачи управления в сегменты кода с иными уровнями привилегий введен *механизм шлюзов*, который мы вкратце рассмотрим ниже. Более подробное рассмотрение затронутых вопросов выходит за рамки темы данной книги. Для получения более детальных сведений по этому и некоторым другим вопросам особенностей архитектуры микропроцессоров i80x86 рекомендуется обратиться к таким материалам, как, например, [1, 8].

Механизм шлюзов для передачи управления на сегменты кода с другими уровнями привилегий

Поскольку межсегментные переходы контролируются с использованием уровней привилегий, а потребность в передаче управления с одного уровня привилегий на другой уровень имеется, в микропроцессорах i80x86 реализован *механизм шлюзов*, который мы поясним с помощью рис. 4.9. Шлюзование позволяет организовать обращение к так называемым *подчиненным* сегментам кода, которые выполняют часто встречающиеся функции и должны быть доступны многим задачам, располагающимся на том же или нижележащем уровне привилегий. Часто уровни привилегий называют *кольцами защиты*, поскольку это иногда помогает объяснить принцип действия самого механизма. Часто говорят, что некоторый программный модуль «исполняется в кольце защиты с номером ...».

Помимо дескрипторов сегментов системными объектами, с которыми работает микропроцессор, являются специальные системные дескрипторы, названные *шлюзами* (gates). Главное различие между дескриптором сегмента и шлюзом вызова подчиненного сегмента кода заключается в том, что содержимое дескриптора указывает на сегмент в памяти, а шлюз обращается к дескриптору. Другими словами, если дескриптор служит механизмом отображения памяти, то шлюз служит механизмом перенаправления вычислений.

Для доступа к более привилегированному коду задача должна обратиться к нему не непосредственно (путем указания дескриптора этого кода), а через шлюз этого сегмента (рис. 4.10).

В этом дескрипторе вместо адреса сегмента указываются селектор, позволяющий найти дескриптор искомого сегмента кода, и адрес (смещение назначения), с которого будет выполняться подчиненный сегмент, то есть полный 32-разрядный адрес. Формат *дескриптора шлюза* приведен на рис. 4.11. Адресовать шлюз вызова можно с помощью команды CALL или FAR CALL (межсегментный вызов процедуры). По существу, дескрипторы шлюзов вызова не являются дескрипторами сегментов, но могут располагаться среди обычных дескрипторов (в дескрипторных таблицах) процесса. Смещение, указываемое в команде перехода на другой сегмент (FAR CALL), игнорируется, и фактически осуществляется переход на команду, адрес которой определяется через смещение из шлюза вызова. Этим гарантируется попадание только на разрешенные точки входа в подчиненные сегменты.

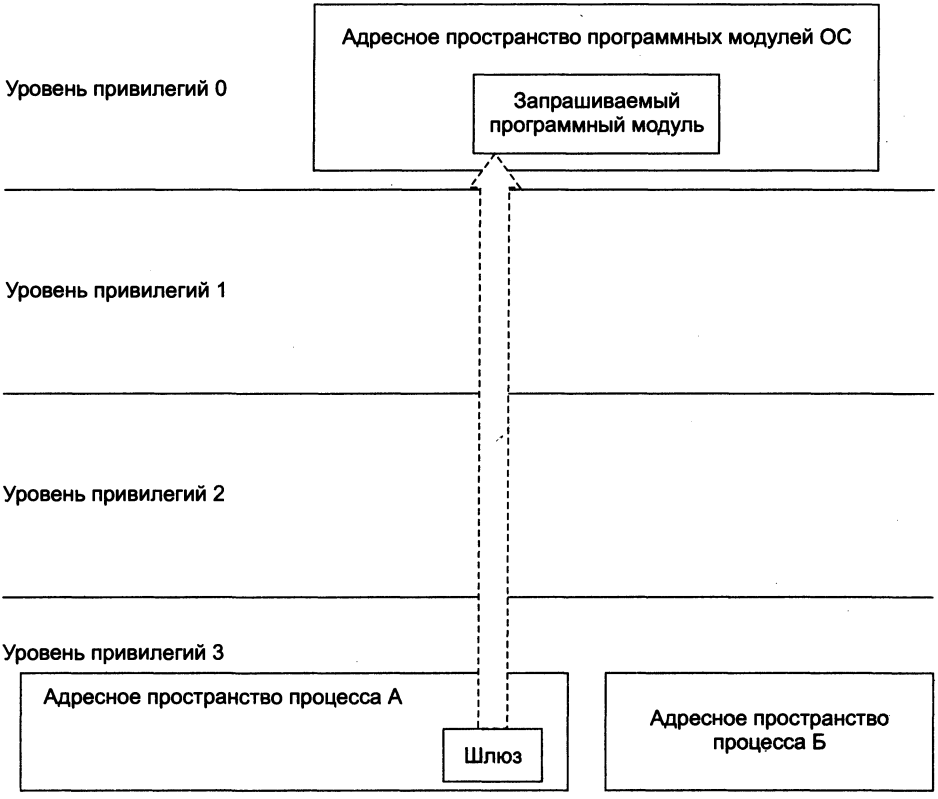


Рис. 4.9. Механизм шлюзов для перехода на другой уровень привилегий

Введены следующие *правила использования шлюзов*:

- значение DPL шлюза вызова должно быть больше или равно значению текущего уровня привилегий CPL;
- значение DPL шлюза вызова должно быть больше или равно значению поля RPL селектора шлюза;
- значение DPL шлюза вызова должно быть больше или равно значению DPL целевого сегмента кода;
- значение DPL целевого сегмента кода должно быть меньше или равно значению текущего уровня привилегий CPL.

Требование наличия и доступности шлюза вызова для перехода на более привилегированный код ограничивает менее привилегированный код заданным набором точек входа. Так как шлюзы вызова являются элементами дескрипторных таблиц (а мы говорили, что их не только можно, но и желательно там располагать), то менее привилегированная программа не может создать дополнительных (а значит, и неконтролируемых) шлюзов. Таким образом, рассмотренный механизм шлюзов дает следующие преимущества в организации среды надежных вычислений.

- Привилегированный код надежно защищен, и вызывающие его программы не могут его разрушить. Естественно, что такой системный код должен быть особенно тщательно отлаженным, не содержать ошибок, быть максимально эффективным.
- Шлюзы межсегментных переходов для вызова системных функций делают эти самые системные функции невидимыми для программных модулей, расположенных на внешних (более низких) уровнях привилегий.
- Поскольку вызывающая программа непосредственно адресует только шлюз вызова, реализуемые вызываемым модулем (сегментным кодом) функции можно изменить или переместить в адресном пространстве, не затрагивая интерфейс со шлюзом.
- Легко реализуется вызов программных модулей с более привилегированного уровня.

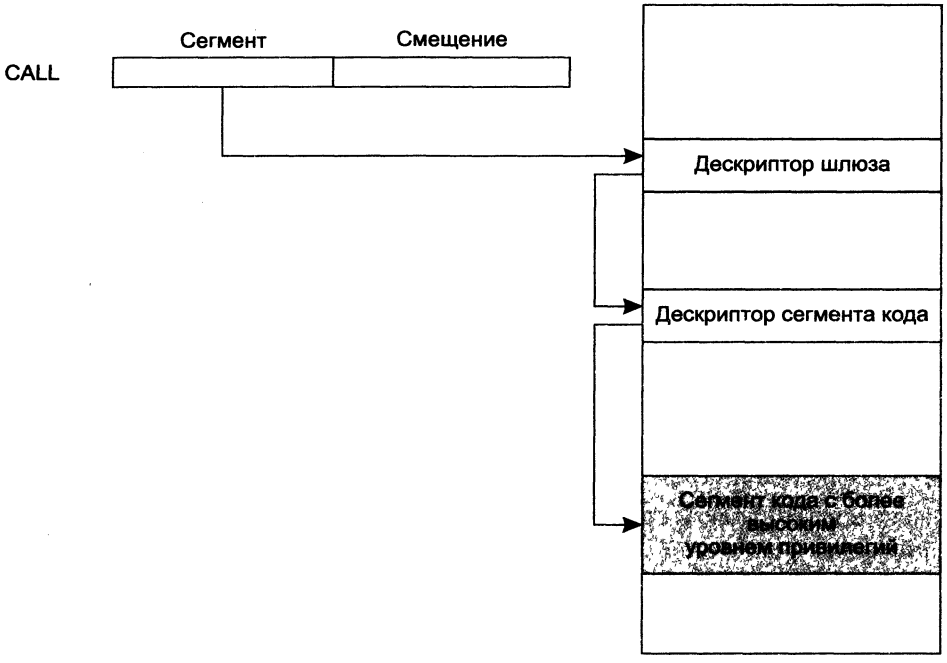


Рис. 4.10. Переход на сегмент более привилегированного кода

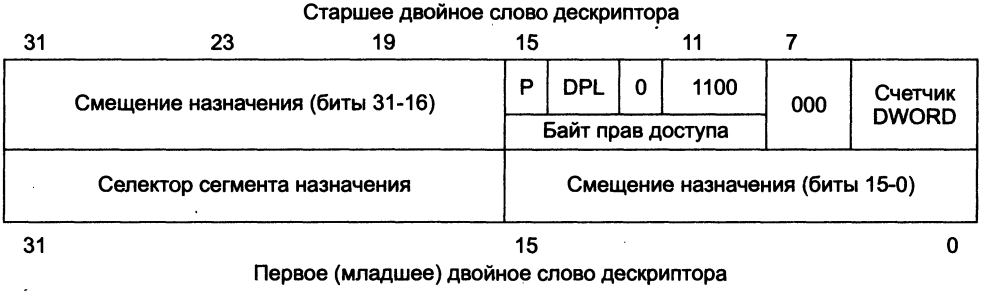


Рис. 4.11. Формат дескриптора шлюза

Изложенный вкратце аппаратный механизм защиты по привилегиям оказывается довольно сложным и жестким. Однако поскольку все практические ситуации учесть в схемах микропроцессора невозможно, то при разработке процедур операционных систем и иного привилегированного кода следует придерживаться приведенных ниже рекомендаций, заимствованных из [8].

Основной риск связан с передачей управления через шлюз вызова более привилегированной процедуре. Нельзя предоставлять вызывающей программе никаких преимуществ, вытекающих из-за временного повышения привилегий. Это особенно важно для процедур нулевого уровня привилегий (PL0-процедур).

Вызывающая программа может нарушить работу процедуры, передавая ей «плохие» параметры. Поэтому целесообразно как можно раньше проконтролировать передаваемые процедуре параметры. Шлюз вызова сам по себе не проверяет значений параметров, которые копируются в новый стек, поэтому достоверность каждого передаваемого параметра должна контролировать вызванная процедура. Ниже перечислены некоторые рекомендации по контролю передаваемых параметров.

- Следует проверять счетчики циклов и повторений на минимальные и максимальные значения.
- Необходимо проверить 8- и 16-разрядные параметры, передаваемые в 32-разрядных регистрах. Когда процедуре передается короткий параметр, его следует расширить знаковым разрядом или нулем для заполнения всего 32-разрядного регистра.
- Следует стремиться свести к минимуму время работы процессора с запрещенными прерываниями. Если процедуре требуется запрещать прерывания, необходимо, чтобы вызывающая программа не могла влиять на время нахождения процессора с запрещенными прерываниями (флаг IF = 0).
- Процедура никогда не должна воспринимать как параметр код или указатель на код.
- В операциях процессора следует явно задавать состояние флага направления DF для цепочечных команд.
- Заключительная команда RET или RET n в процедуре должна точно соответствовать полю WC (Word Counter — счетчик слов) шлюза вызова; при этом n = 4 × WC, так как счетчик задает число двойных слов, а n соответствует байтам.
- Не следует применять шлюзы вызовов для функций, которым передается переменное число параметров (см. предыдущую рекомендацию). При необходимости нужно воспользоваться счетчиком и указателем параметров.
- Функции не могут возвращать значения в стеке (см. предыдущую рекомендацию), так как после возврата стеки процедуры и вызывающей программы находятся точно в таком состоянии, в каком они были до вызова.
- В процедуре следует сохранять и восстанавливать все сегментные регистры. Без этого, если какой-либо сегментный регистр привлекался для адресации данных, недоступных вызывающей программе, процессор автоматически загрузит в него пустой селектор.

Рекомендуется контролировать все обращения к памяти. Нетрудно представить себе ситуацию, когда PL3-программа[1] передает PL0-процедуре указатель *селектор:смещение* и запрашивает считать или записать несколько байтов по этому адресу. Типичным примером может служить процедура дискового ввода-вывода, которая воспринимает как параметр системный номер файла, счетчик байтов и адрес, по которому записываются данные с диска. Хотя PL0-процедура имеет привилегии для производства такой операции, у PL3-программы разрешения на это может не быть.

Система прерываний 32-разрядных микропроцессоров i80x86

В микропроцессорах семейства i80x86 система прерываний построена таким образом, чтобы, с одной стороны, обеспечить возможность создавать эффективные и надежные мультипрограммные и мультизадачные операционные системы, которые должны функционировать в защищенном режиме, а с другой стороны, обеспечить возможность выполнять программы, разработанные для реального режима. Рассмотрим вкратце оба режима.

Работа системы прерываний в реальном режиме

В реальном режиме работы в системе прерываний используется понятие *вектора прерывания*, поскольку для указания адреса программы обработки прерывания здесь требуется не одно значение, а два (значение для сегментного регистра кода и значение для указателя команд), то есть мы имеем дело не со скалярной величиной, а с «векторной», состоящей из двух скалярных.

Итак, каждый вектор прерывания состоит из четырех байтов, или двух слов: первые два содержат новое значение для регистра IP, а следующие два — новое значение для регистра CS. *Таблица векторов прерывания* занимает 1024 байт. Таким образом, в ней может быть задано 256 векторов прерываний. В процессоре i8086 эта таблица располагается на адресах 00000H–003FFH. Расположение этой таблицы в процессорах i80286 и в более поздних определяется значением регистра *IDTR* (Interrupt Descriptor Table Register — регистр таблицы дескрипторов прерываний). При включении или сбросе процессора i80x86 этот регистр обнуляется. Однако при необходимости можно в регистре IDTR указать смещение и таким образом перейти на новую таблицу векторов прерываний.

Таблица векторов прерываний заполняется (инициализируется) при запуске системы, но, в принципе, может быть изменена или перемещена.

Каждый вектор прерывания имеет свой номер, называемый номером прерывания, который указывает его место в таблице. Этот номер, помноженный на четыре (сдвиг на два разряда влево и заполнение освободившихся битов нулями) и сложенный с содержимым регистра IDTR, дает абсолютный адрес первого байта вектора прерываний в оперативной памяти.

---

[1] Программа, имеющая уровень привилегий 3. Иначе говоря, работающая в кольце защиты с номером 3.

Подобно вызову процедуры прерывание заставляет микропроцессор сохранить в стеке информацию для последующего возврата, а затем перейти к группе команд, адрес которых определяется вектором прерывания. Таким образом, прерывание вызывает косвенный переход к своей подпрограмме обработки за счет получения ее адреса из вектора прерывания.

В IBM PC, как и в других вычислительных системах, прерывания бывают двух видов: внутренние и внешние.

Внутренние прерывания, как мы уже знаем, возникают в результате работы процессора в ситуациях, которые нуждаются в специальном обслуживании, или при выполнении специальных команд (INT, INTO). Это следующие прерывания:

- прерывание при делении на ноль (номер прерывания 0);
- прерывание по флагу TF (Trap Flag — флаг трассировки)[1] обычно используется специальными программами отладки типа DEBUG (номер прерывания 1);
- прерывания, возникающие при выполнении команд INT (Interrupt — прерывание) и INTO (Interrupt if Overflow — прерывание по переполнению), называются программными.

В качестве операнда команды INT указывается номер прерывания, которое нужно выполнить, например INT 10H. Программные прерывания как средство перехода на соответствующую процедуру были введены для того, чтобы выполнение этой процедуры осуществлялось в привилегированном режиме, а не в обычном пользовательском.

Внешние прерывания возникают по сигналу какого-нибудь внешнего устройства. Существует два специальных внешних сигнала среди входных сигналов процессора, при помощи которых можно прервать выполнение текущей программы и тем самым переключить работу центрального процессора. Это сигналы *NMI* (No Mask Interrupt — немаскируемое прерывание) и *INTR* (Interrupt Request — запрос на прерывание). Соответственно, внешние прерывания подразделяются на немаскируемые и маскируемые.

Маскируемые прерывания генерируются контроллером прерываний по заявке определенных периферийных устройств[2]. Контроллер прерываний (его обозначение i8259A) поддерживает восемь уровней (линий) приоритета; к каждому уровню «привязано» одно периферийное устройство[3]. Маскируемые прерывания часто называют аппаратными прерываниями.

---

[1] Флаг трассировки — специальный бит в регистре PSW (Program Status Word — слово состояния программы), который в случае равенства единице вызывает приостанов после каждой команды и генерирует прерывание для организации режима отладки с пошаговым выполнением программы. Чаще всего регистр PSW в микропроцессорах Intel 80x86 называют регистром флагов.

[2] Сигнал запроса на прерывание чаще всего является сигналом готовности внешнего устройства (соответствующего контроллера внешнего устройства) на выполнение следующей команды, связанной с управлением операциями ввода-вывода.

[3] В качестве внешнего периферийного устройства, занимающего одну линию запроса на прерывание, может быть использовано специальное управляющее устройство, которое позволяет разделять эту самую линию запроса между несколькими внешними устройствами.

Как известно, прерывания могут быть инициированы внешним устройством ПЭВМ или специальной командой прерывания из программы. В любом случае если прерывания разрешены, то выполняется следующая процедура.

1. В стек помещается регистр флагов PSW.
2. Флаг включения-выключения прерываний IF и флаг трассировки TF, находящиеся в регистре PSW, обнуляются для блокировки других маскируемых прерываний и исключения пошагового режима исполнения команд.
3. Значения регистров CS и IP сохраняются в стеке вслед за PSW.
4. Вычисляется адрес вектора прерывания и из вектора, соответствующего номеру прерывания, загружаются новые значения IP и CS.

Когда системная подпрограмма принимает управление, она может разрешить снова маскируемые прерывания командой STI (Set Interrupt Flag – установить флаг прерываний), которая переводит флаг IF в состояние 1, что разрешает микропроцессору вновь реагировать на прерывания, инициируемые внешними устройствами, поскольку стековая организация допускает вложение прерываний друг в друга.

Закончив работу, подпрограмма обработки прерывания должна выполнить команду IRET (Interrupt Return), которая извлекает из стека три 16-разрядных значения и загружает их в указатель команд IP, регистр сегмента команд CS и регистр PSW соответственно. Таким образом, процессор сможет продолжить работу с того места, где он был прерван.

В случае внешних прерываний процедура перехода на подпрограмму обработки прерывания дополняется следующими шагами.

1. Контроллер прерываний получает заявку от определенного периферийного устройства и, соблюдая схему приоритетов, генерирует сигнал INTR (запрос на прерывание), который является входным для микропроцессора.
2. Микропроцессор проверяет флаг IF в регистре PSW. Если он установлен в 1, то переходим к шагу 3. В противном случае работа процессора не прерывается. Часто говорят, что прерывания замаскированы, хотя правильнее говорить, что они отключены. Маскируются (запрещаются) отдельные линии запроса на прерывания посредством программирования контроллера прерываний.
3. Микропроцессор генерирует сигнал INTA (подтверждение прерывания). В ответ на этот сигнал контроллер прерываний посылает по шине данных номер прерывания. После этого выполняется описанная ранее процедура передачи управления соответствующей программе обработки прерывания.

Номер прерывания и его приоритет устанавливаются на этапе инициализации системы. После запуска ОС пользователь, как мы уже отмечали, может изменить таблицу векторов прерываний, поскольку она ему доступна.

Работа системы прерываний в защищенном режиме

В защищенном режиме работы система прерываний микропроцессора i80x86 работает совершенно иначе. Прежде всего, вместо таблицы векторов, о которой мы

говорили выше, она имеет дело с *таблицей дескрипторов прерываний* (Interrupt Descriptor Table, IDT). Дело здесь не столько в названии таблицы, сколько в том, что таблица IDT представляет собой не таблицу с адресами обработчиков прерываний, а таблицу со специальными системными структурами данных (дескрипторами), доступ к которой со стороны пользовательских (прикладных) программ невозможен. Только сам микропроцессор (его система прерываний) и код операционной системы могут получить доступ к этой таблице, представляющей собой специальный сегмент, адрес и длина которого содержатся в регистре IDTR (см. рис. 4.2). Этот регистр аналогичен регистру GDTR в том отношении, что он инициализируется один раз при загрузке системы. Интересно заметить, что в реальном режиме работы регистр IDTR также указывает на адрес таблицы прерываний, но при этом, как и в процессоре i8086, каждый элемент таблицы прерываний (вектор) занимает всего 4 байт и содержит 32-разрядный адрес в формате *селектор:смещение* (CS:IP). Начальное значение этого регистра равно нулю, но в него можно занести и другое значение. В этом случае таблица векторов прерываний будет находиться в другом месте оперативной памяти. Естественно, что перед тем, как занести в регистр IDTR новое значение, необходимо подготовить саму таблицу векторов. В защищенном режиме работы загрузку регистра IDTR может произвести только код с максимальным уровнем привилегий.

Каждый элемент в таблице дескрипторов прерываний, о которой мы говорим уже в защищенном режиме, представляет собой 8-байтовую структуру, более похожую на дескриптор шлюза, нежели на дескриптор сегмента.

Как мы уже знаем, в зависимости от причины прерывания процессор автоматически индексирует таблицу прерываний и выбирает соответствующий элемент, с помощью которого и осуществляется перенаправление в исполнении кода, то есть передача управления на обработчик прерывания. Однако таблица IDT содержит только дескрипторы шлюзов, а не дескрипторы сегментов кода, поэтому фактически получается что-то типа косвенной адресации, но с рассмотренным ранее механизмом защиты с помощью уровней привилегий. Благодаря этому пользователи уже не могут сами изменить обработку прерываний, которая предопределяется системным программным обеспечением.

Дескриптор прерываний может относиться к одному из трех типов:

- *коммутатор прерывания* (interrupt gate);
- *коммутатор перехвата* (trap gate);
- *коммутатор задачи* (task gate).

При обнаружении запроса на прерывание и при условии, что прерывания разрешены, процессор действует в зависимости от типа дескриптора (коммутатора), соответствующего номеру прерывания. Первые два типа дескрипторов прерываний вызывают переход на соответствующие сегменты кода, принадлежащие виртуальному адресному пространству текущего вычислительного процесса. Поэтому про них говорят, что обработка прерываний по этим дескрипторам осуществляется *под контролем (в контексте) текущей задачи*. Последний тип дескриптора (коммутатор задачи) вызывает *полное переключение процессора на новую задачу* со сменой всего контекста в соответствии с сегментом состояния задачи (TSS). Рассмотрим оба варианта.

Обработка прерываний в контексте текущей задачи

Обработку прерывания в контексте текущей задачи поясняет рис. 4.12.

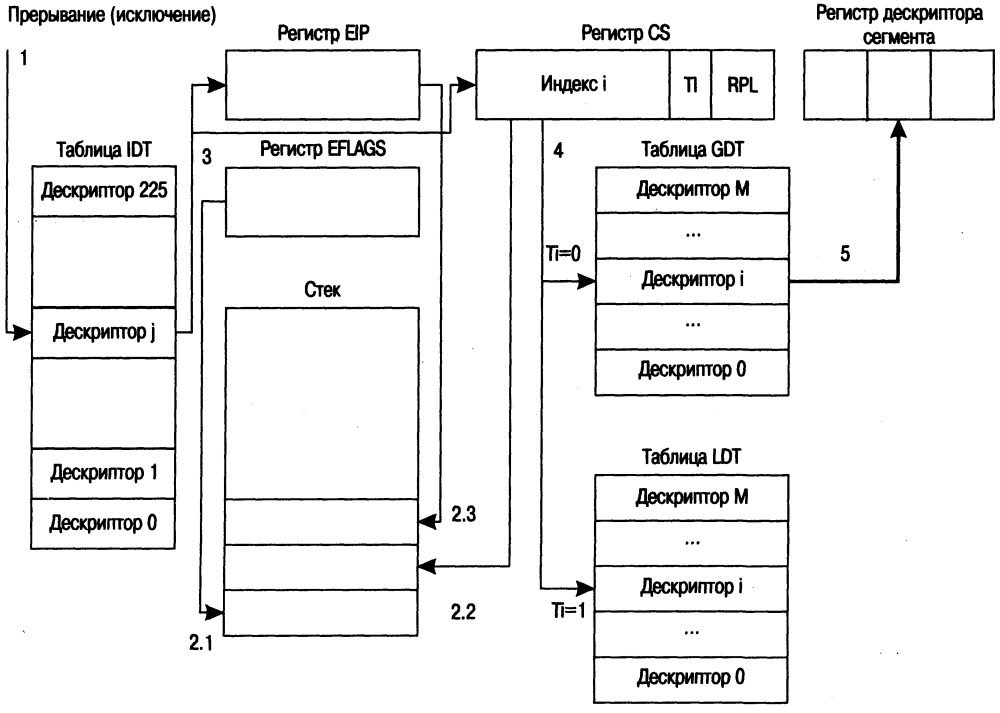


Рис. 4.12. Схема передачи управления при прерывании в контексте текущей задачи

При возникновении прерывания процессор по номеру прерывания индексирует таблицу IDT, то есть адрес соответствующего коммутатора определяется путем сложения содержимого поля адреса в регистре IDTR и номера прерывания, умноженного на 8 (справа к номеру прерывания добавляется три нуля). Полученный дескриптор анализируется, и если его тип соответствует коммутатору перехвата или коммутатору прерывания, то выполняются следующие действия.

1. В стек на уровне привилегий текущего сегмента кода помещаются:
   - значения SS и SP, если уровень привилегий в коммутаторе выше уровня привилегий ранее исполнявшегося кода;
   - регистр флагов EFLAGS;
   - регистры CS и IP.
2. Если рассматриваемому прерыванию соответствовал коммутатор прерывания, то запрещаются прерывания (устанавливается флаг IF = 0 в регистре EFLAGS). В случае коммутатора перехвата флаг прерываний не сбрасывается, и обработка новых прерываний на период обработки текущего прерывания тем самым не запрещается.

3. Поле селектора из дескриптора прерывания используется для индексирования таблицы дескрипторов задачи. Дескриптор сегмента заносится в теневой регистр, а смещение относительно начала нового сегмента кода определяется полем смещения из дескриптора прерывания.

Таким образом, в случае обработки прерываний, когда дескриптором прерывания является коммутатор перехвата или коммутатор прерывания, мы остаемся в том же виртуальном адресном пространстве, и полной смены контекста текущей задачи не происходит. Просто мы переключаемся на исполнение другого (как правило, более привилегированного) кода, доступного исполняемой задаче. Этот код создается системными программистами, и прикладные программисты его просто используют. В то же время механизмы защиты микропроцессора позволяют обеспечить недоступность этого кода для его исправления (со стороны приложений, его вызывающих) и недоступность самой таблицы дескрипторов прерываний. Удобнее всего код обработчиков прерываний располагать в общем адресном пространстве, то есть селекторы, указывающие на такой код, должны располагаться в глобальной таблице дескрипторов.

Обработка прерываний с переключением на новую задачу

Совершенно иначе осуществляется обработка прерываний в случае, если дескриптором прерываний является коммутатор задачи. Формат коммутатора задачи отличается от формата коммутаторов перехвата и прерывания прежде всего тем, что в нем вместо селектора сегмента кода, на который передается управление, указывается селектор сегмента состояния задачи (рис. 4.13). В результате осуществляется процедура перехода на новую задачу с полной сменой контекста, ибо сегмент состояния задачи полностью определяет новое виртуальное пространство и адрес начала программы, а текущее состояние прерываемой задачи аппаратно (по микропрограмме микропроцессора) сохраняется в ее собственном сегменте TSS.

При этом происходит полное переключение на новую задачу с вложением, то есть выполняются следующие действия.

1. Сохраняются все рабочие регистры процессора в текущем сегменте TSS, базовый адрес этого сегмента берется в регистре TR (см. раздел «Адресация в 32-разрядных микропроцессорах i80x86 при работе в защищенном режиме»).
2. Текущая задача отмечается как занятая.
3. По селектору из коммутатора задачи выбирается новый сегмент TSS (поле селектора помещается в регистр TR) и загружается состояние новой задачи. Напомним, что загружаются значения регистров LDTR, EFLAGS, восьми регистров общего назначения, регистра EIP и шести сегментных регистров.
4. Устанавливается бит NT (Next Task).
5. В поле обратной связи TSS помещается селектор прерванной задачи.
6. С помощью значений CS:IP, взятых из нового сегмента TSS, обнаруживается и выполняется первая команда обработчика прерывания.

Таким образом, коммутатор задачи дает указание процессору произвести переключение задачи, и обработка прерывания осуществляется под контролем отдельной внешней задачи. В каждом сегменте TSS имеется селектор локальной таблицы деск-

рипторов (LDT), поэтому при переключении задачи процессор загружает в регистр LDTR новое значение. Это позволяет обратиться к сегментам кода, которые абсолютно не пересекаются с сегментами кода любых других задач, поскольку именно локальные таблицы дескрипторов обеспечивают эффективную изоляцию виртуальных адресных пространств. Новая задача начинает свое выполнение на уровне привилегий, определяемом полем RPL нового содержимого регистра CS, которое загружается из сегмента TSS. Достоинством этого коммутатора является то, что он позволяет сохранить все регистры процессора с помощью механизма переключения задач, тогда как коммутаторы перехвата и прерываний сохраняют только содержимое регистров IFLAGS, CS и IP, а сохранение других регистров возлагается на программиста, разрабатывающего соответствующую программу обработки прерывания.

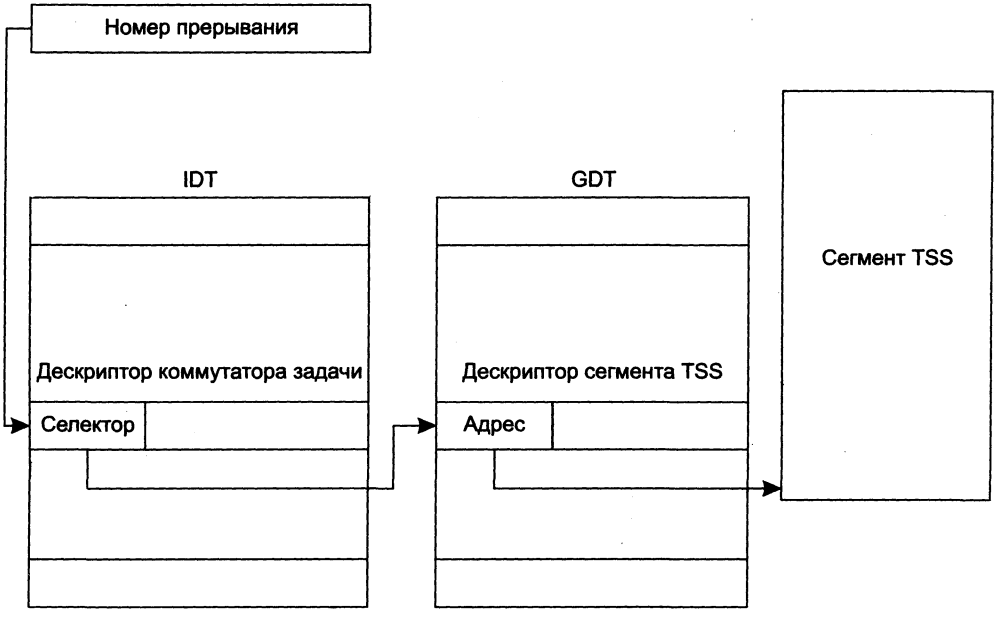


Рис. 4.13. Схема передачи управления при прерывании с переключением на новую задачу

Справедливости ради следует признать, что несмотря на возможности коммутатора задачи, разработчики современных операционных систем достаточно редко его используют, поскольку переключение на другую задачу требует существенно больших затрат времени, а полное сохранение всех рабочих регистров часто не требуется. В основном обработку прерываний осуществляют в контексте текущей задачи, так как это приводит к меньшим накладным расходам и повышает быстродействие системы.

Контрольные вопросы и задачи

1. Как в реальном режиме работы микропроцессоров i80x86 осуществляется преобразование виртуального адреса в физический?

2. Какие механизмы виртуальной памяти используются в защищенном режиме работы микропроцессоров i80x86?
3. Для чего в микропроцессоры i80x86 введен регистр-указатель задачи TR? Какой он разрядности?
4. Как в микропроцессорах i80x86 реализована поддержка сегментного способа организации виртуальной памяти?
5. Что понимается под термином «линейный адрес»? Как осуществляется преобразование линейного адреса в физический? Может ли линейный адрес быть равным физическому?
6. Что дало введение двухэтапной страничной трансляции в механизме страничного способа реализации виртуальной памяти? Как разработчики микропроцессора i80386 решили проблему замедления доступа к памяти, которое при двухэтапном преобразовании адресов очень существенно?
7. Что означает термин «плоская модель памяти»? В чем заключаются достоинства (и недостатки, если они есть) этой модели?
8. Что дало введение виртуального режима? Как в этом режиме осуществляется вычисление физического адреса?
9. Что имеется в микропроцессорах i80x86 для обеспечения защиты адресного пространства задач?
10. Что такое «уровень привилегий»? Сколько уровней привилегий в микропроцессорах i80x86? Для каких целей введено такое количество уровней привилегий?
11. Что такое текущий уровень привилегий? Как узнать, чему он равен? Что такое эффективный уровень привилегий?
12. Объясните правила работы с уровнями привилегий для различных типов сегментов.
13. Поясните работу механизма шлюзов. Для чего он предназначен, как осуществляется передача управления на сегменты кода с другими уровнями привилегий?
14. Опишите работу системы прерываний микропроцессоров i80x86 в реальном режиме.
15. В чем заключаются принципиальные отличия работы системы прерываний микропроцессоров i80x86 в защищенном режиме по сравнению с реальным режимом?
16. Как осуществляется переход на программу обработки прерываний, если дескриптор прерываний является коммутатором прерываний?
17. Как осуществляется переход на программу обработки прерываний, если дескриптор прерываний является коммутатором перехвата?
18. Как осуществляется переход на программу обработки прерываний, если дескриптор прерываний является коммутатором задачи?

Глава 5. Управление вводом-выводом в операционных системах

Побудительной причиной, в конечном итоге приведшей разработчиков к созданию системного программного обеспечения, в том числе операционных систем, стала необходимость предоставить программам средства обмена данными с внешними устройствами, которые бы не требовали непосредственного включения в каждую программу двоичного кода, управляющего устройствами ввода-вывода. Напомним, что программирование ввода-вывода является наиболее сложным и трудоемким, требующим очень высокой квалификации. Поэтому код, реализующий операции ввода-вывода, сначала стали оформлять в виде системных библиотечных процедур, а потом и вовсе вывели из систем программирования, включив в операционную систему. Это позволило не писать такой код в каждой программе, а только обращаться к нему — системы программирования стали генерировать обращения к системному коду ввода-вывода. Таким образом, управление вводом-выводом — это одна из основных функций любой операционной системы.

С одной стороны, организация ввода-вывода в различных операционных системах имеет много общего. С другой стороны, реализация ввода-вывода в ОС так сильно отличается от системы к системе, что очень нелегко выделить и описать именно основные принципы реализации этих функций. Проблема усугубляется еще и тем, что в большинстве ныне используемых систем эти моменты вообще, как правило, подробно не описаны (исключением являются только системы Linux и FreeBSD, для которых имеются комментированные исходные тексты), а детально описываются только функции API, реализующие ввод-вывод. Другими словами, для тех же систем Windows от компании Microsoft мы воспринимаем подсистему ввода-вывода как «черный ящик». Известно, как можно и нужно использовать эту подсистему, но детали ее внутреннего устройства остаются неизвестными. Поэтому в данной главе мы рассмотрим только основные идеи и концепции. Наконец, поскольку такой важный ресурс, как внешняя память, в основном реализуется на устройствах ввода-вывода с прямым доступом, а к ним, прежде всего, относятся накопители на магнитных дисках, мы также рассмотрим логическую структуру дис-

ка, начальную стадию процесса загрузки операционной системы, кэширование операций ввода-вывода, оптимизацию дисковых операций.

Основные концепции организации ввода-вывода в операционных системах

Как известно, ввод-вывод считается одной из самых сложных областей проектирования операционных систем, в которой сложно применить общий подход и в которой изобилуют частные методы. В действительности, источником сложности является огромное число устройств ввода-вывода разнообразной природы, которые должна поддерживать операционная система. При этом перед создателями операционной системы встает очень непростая задача — не только обеспечить эффективное управление устройствами ввода-вывода, но и создать удобный и эффективный виртуальный интерфейс устройств ввода-вывода, позволяющий прикладным программистам просто считывать или сохранять данные, не обращая внимание на специфику устройств и проблемы распределения устройств между выполняющимися задачами. Система ввода-вывода, способная объединить в одной модели широкий спектр устройств, должна быть универсальной. Она должна учитывать потребности существующих устройств, от простой мыши до клавиатур, принтеров, графических дисплеев, дисковых накопителей, компакт-дисков и даже сетей. С другой стороны, необходимо обеспечить доступ к устройствам ввода-вывода для множества параллельно выполняющихся задач, причем так, чтобы они как можно меньше мешали друг другу.

Поэтому самым главным является следующий принцип: *любые операции по управлению вводом-выводом объявляются привилегированными и могут выполняться только кодом самой операционной системы.* Для обеспечения этого принципа в большинстве процессоров даже вводятся *режимы пользователя* и *супервизора.* Последний еще называют *привилегированным режимом*, или *режимом ядра.* Как правило, в режиме *супервизора* выполнение команд ввода-вывода разрешено, а в пользовательском режиме — запрещено. Обращение к командам ввода-вывода в пользовательском режиме вызывает *исключение*[1], и управление через механизм прерываний передается коду операционной системы. Хотя возможны и более сложные схемы, в которых в ряде случаев пользовательским программам может быть разрешено непосредственное выполнение команд ввода-вывода.

Еще раз подчеркнем, что мы, прежде всего, говорим о мультипрограммных операционных системах, для которых существует проблема разделения ресурсов, и одним из основных видов ресурсов являются устройства ввода-вывода и соответствующее программное обеспечение, с помощью которого осуществляется обмен данными между внешними устройствами и оперативной памятью. Помимо разделяемых устройств ввода-вывода (эти устройства допускают разделение посредством механизма доступа) существуют неразделяемые устройства. Примерами

[1] Исключение — это определенный вид внутреннего прерывания. Этим термином, во-первых, обозначают некоторое множество синхронных прерываний, а во-вторых, подчеркивают, что ситуация, вызвавшая запрос на прерывание, является исключительной, то есть отличается от обычной.

разделяемого устройства могут служить накопитель на магнитных дисках, устройство чтения компакт-дисков. Это устройства с прямым доступом. Примеры неразделяемых устройств — принтер, накопитель на магнитных лентах. Это устройства с последовательным доступом. Операционные системы должны управлять и теми, и другими, предоставляя возможность параллельно выполняющимся задачам их использовать.

Можно назвать три основные причины, по которым нельзя разрешать каждой отдельной пользовательской программе обращаться к внешним устройствам непосредственно.

- *Необходимость разрешать возможные конфликты в доступе к устройствам ввода-вывода.* Например, пусть две параллельно выполняющиеся программы пытаются вывести на печать результаты своей работы. Если не предусмотреть внешнего управления устройством печати, то в результате мы можем получить абсолютно нечитаемый текст, так как каждая программа будет время от времени выводить свои данные, перемежающиеся с данными от другой программы. Либо можно взять ситуацию, когда для одной программы необходимо прочитать данные с одного сектора магнитного диска, а для другой записать результаты в другой сектор того же накопителя. Если операции ввода-вывода не будут отслеживаться каким-то третьим (внешним) процессом-арбитром, то после позиционирования магнитной головки для первой задачи может тут же прийти команда позиционирования головки для второй задачи, и обе операции ввода-вывода не смогут выполниться корректно.
- *Желание увеличить эффективность использования ресурсов ввода-вывода.* Например, у накопителя на магнитных дисках время подвода головки чтения/записи к необходимой дорожке и время обращения к определенному сектору могут значительно (до тысячи раз) превышать время пересылки данных. В результате, если задачи по очереди обращаются к цилиндрам, далеко отстоящим друг от друга, то полезная работа, выполняемая накопителем, может быть существенно снижена.
- *Необходимость избавить программы ввода-вывода от ошибок.* Ошибки в программах ввода-вывода могут привести к краху всех вычислительных процессов, ибо часть операций ввода-вывода требуются самой операционной системе. В ряде операционных систем системный ввод-вывод имеет существенно более высокие привилегии, чем ввод-вывод задач пользователя. Поэтому системный код, управляющий операциями ввода-вывода, очень тщательно отлаживается и оптимизируется для повышения надежности вычислений и эффективности использования оборудования.

Итак, управление вводом-выводом осуществляется компонентом операционной системы, который часто называют *супервизором ввода-вывода.* Перечислим основные задачи, возлагаемые на супервизор.

1. Модуль супервизора операционной системы, иногда называемый *супервизором задач*, получает запросы от прикладных задач на выполнение тех или иных операций, в том числе на ввод-вывод. Эти запросы проверяются на корректность и, если они соответствуют спецификациям и не содержат ошибок, то обрабатыва-

ются дальше. В противном случае пользователю (задаче) выдается соответствующее диагностическое сообщение о недействительности (некорректности) запроса.

2. Супервизор ввода-вывода получает запросы на ввод-вывод от супервизора задач или от программных модулей самой операционной системы.
3. Супервизор ввода-вывода вызывает соответствующие распределители каналов и контроллеров, планирует ввод-вывод (определяет очередность предоставления устройств ввода-вывода задачам, затребовавшим эти устройства). Запрос на ввод-вывод либо тут же выполняется, либо ставится в очередь на выполнение.
4. Супервизор ввода-вывода инициирует операции ввода-вывода (передает управление соответствующим драйверам) и в случае управления вводом-выводом с использованием прерываний предоставляет процессор диспетчеру задач с тем, чтобы передать его первой задаче, стоящей в очереди на выполнение.
5. При получении сигналов прерываний от устройств ввода-вывода супервизор идентифицирует эти сигналы (см. раздел «Прерывания» в главе 1) и передает управление соответствующим программам обработки прерываний.
6. Супервизор ввода-вывода осуществляет передачу сообщений об ошибках, если таковые происходят в процессе управления операциями ввода-вывода.
7. Супервизор ввода-вывода посылает сообщения о завершении операции ввода-вывода запросившей эту операцию задаче и снимает ее с состояния ожидания ввода-вывода, если задача ожидала завершения операции.

В случае, если устройство ввода-вывода является *инициативным*[1], управление со стороны супервизора ввода-вывода будет заключаться в активизации соответствующего вычислительного процесса (перевод его в состояние готовности к выполнению).

Таким образом, прикладные программы (а в общем случае — все обрабатывающие программы) не могут непосредственно связываться с устройствами ввода-вывода независимо от того, в каком режиме используются эти устройства (монопольно или совместно), но, установив соответствующие значения параметров в *запросе на ввод-вывод*, определяющие требуемую операцию и количество потребляемых ресурсов, обращаются к супервизору задач. Последний передает управление супервизору ввода-вывода, который и запускает необходимые логические и физические операции.

---

[1] Инициативным называют такое устройство ввода-вывода, по сигналу прерывания от которого запускается соответствующая ему программа (обычно это не стандартное устройство ввода-вывода, а набор датчиков). Такая программа, с одной стороны, не является драйвером, поэтому ей не нужно управлять операциями обмена данными, но, с другой стороны, запуск такой программы осуществляется именно по событиям, связанным с генерацией устройством ввода-вывода соответствующего сигнала. Разница между драйверами, работающими по прерываниям, и инициативными программами заключается в их статусе. Драйвер является компонентом операционной системы и часто выполняется не как вычислительный процесс, а как системный объект, а инициативная программа является обычным вычислительным процессом, только его запуск осуществляется по инициативе внешнего устройства.

Упомянутый выше запрос на ввод-вывод должен удовлетворять требованиям API той операционной системы, в среде которой выполняется приложение. Параметры, которые указываются в запросах на ввод-вывод, передаются не только в вызывающих последовательностях, создаваемых по спецификациям API, но и как данные, хранящиеся в соответствующих системных таблицах. Все параметры, которые будут стоять в вызывающей последовательности, предоставляются компилятором и отражают требования программиста, а также постоянные сведения об операционной системе и архитектуре компьютера в целом. Переменные сведения о вычислительной системе (ее конфигурация, состав оборудования, состав и особенности системного программного обеспечения) содержатся в специальных системных таблицах. Процессору, каналам прямого доступа в память и контроллерам необходимо передавать конкретную двоичную информацию, с помощью которой и осуществляется управление оборудованием. Эта конкретная двоичная информация в виде кодов и данных часто готовится с помощью препроцессоров, но часть ее хранится в системных таблицах.

Режимы управления вводом-выводом

Как известно, имеется два основных режима ввода-вывода: *режим обмена с опросом готовности* устройства ввода-вывода и *режим обмена с прерываниями* (рис. 5.1).

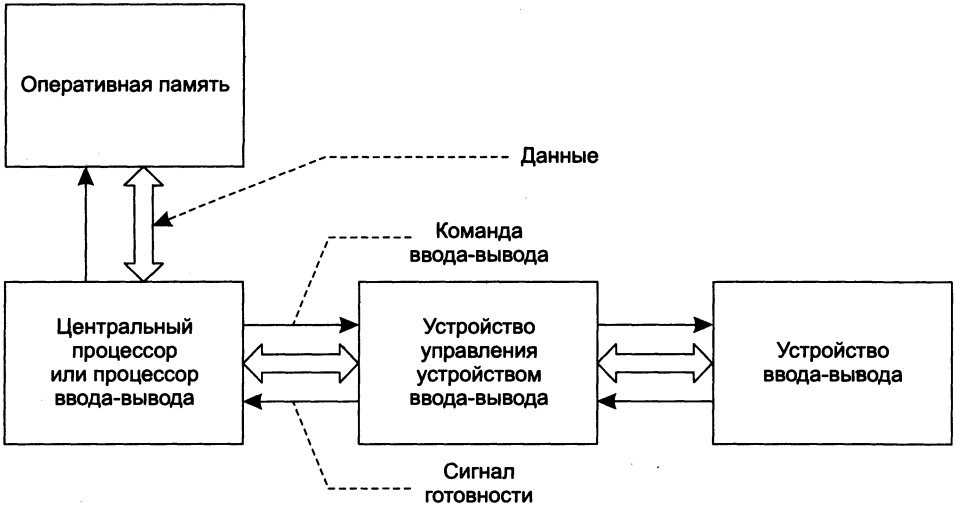


Рис. 5.1. Управление вводом-выводом

Пусть для простоты рассмотрения этих вопросов управление вводом-выводом осуществляет центральный процессор. В этом случае часто говорят о работе *программного канала* обмена данными между внешними устройством и оперативной памятью (в отличие от *канала прямого доступа к памяти*, при котором управление вводом-выводом осуществляет специальное дополнительное оборудование). Итак, пусть центральный процессор посылает команду устройству управления,

требующую, чтобы устройство ввода-вывода выполнило некоторое действие. Например, если мы управляем дисководом, то это может быть команда на включение двигателя или команда, связанная с позиционированием магнитных головок. Устройство управления исполняет команду, транслируя сигналы, понятные ему и центральному устройству, в сигналы, понятные устройству ввода-вывода. После выполнения команды устройство ввода-вывода (или его устройство управления) выдает *сигнал готовности*, который сообщает процессору о том, что можно выдать новую команду для продолжения обмена данными. Однако поскольку быстродействие устройства ввода-вывода намного меньше быстродействия центрального процессора (порой на несколько порядков), то сигнал готовности приходится очень долго ожидать, постоянно опрашивая соответствующую линию интерфейса на наличие или отсутствие нужного сигнала. Посылать новую команду, не дождавшись сигнала готовности, сообщающего об исполнении предыдущей команды, бессмысленно. В режиме опроса готовности драйвер, управляющий процессом обмена данными с внешним устройством, как раз и выполняет в цикле команду «проверить наличие сигнала готовности». До тех пор пока сигнал готовности не появится, драйвер ничего другого не делает. При этом, естественно, нерационально используется время центрального процессора. Гораздо выгоднее, выдав команду ввода-вывода, на время забыть об устройстве ввода-вывода и перейти на выполнение другой программы. А появление сигнала готовности трактовать как запрос на прерывание от устройства ввода-вывода. Именно эти сигналы готовности и являются *сигналами запроса на прерывание* (см. раздел «Прерывания» в главе 1).

Режим обмена с прерываниями по своей сути является режимом асинхронного управления. Для того чтобы не потерять связь с устройством (после выдачи процессором очередной команды по управлению обменом данными и переключения его на выполнение других программ), может быть запущен отсчет времени, в течение которого устройство обязательно должно выполнить команду и выдать-таки сигнал запроса на прерывание. Максимальный интервал времени, в течение которого устройство ввода-вывода или его контроллер должны выдать сигнал запроса на прерывание, часто называют *установкой тайм-аута*. Если это время истекло после выдачи устройству очередной команды, а устройство так и не ответило, то делается вывод о том, что связь с устройством потеряна и управлять им больше нет возможности. Пользователь и/или задача получают соответствующее диагностическое сообщение.

Драйверы, работающие в режиме прерываний, представляют собой сложный комплекс программных модулей и могут иметь несколько секций: *секцию запуска*, одну или несколько *секций продолжения* и *секцию завершения*.

Секция запуска инициирует операцию ввода-вывода. Эта секция запускается для включения устройства ввода-вывода или просто для инициализации очередной операции ввода-вывода.

Секция продолжения (их может быть несколько, если алгоритм управления обменом данными сложный, и требуется несколько прерываний для выполнения одной логической операции) осуществляет основную работу по передаче данных. Секция продолжения, собственно говоря, и является основным обработчиком пре-

рывания. Поскольку используемый интерфейс может потребовать для управления вводом-выводом несколько последовательностей управляющих команд, а сигнал прерывания у устройства, как правило, только один, то после выполнения очередной секции прерывания супервизор прерываний при следующем сигнале готовности должен передать управление другой секции. Это делается путем изменения адреса обработки прерывания после выполнения очередной секции, а если имеется только одна секция продолжения, она сама передает управление в ту или иную часть кода подпрограммы обработки прерывания.

Секция завершения обычно выключает устройство ввода-вывода или просто завершает операцию.

Управление операциями ввода-вывода в режиме прерываний требует значительных усилий со стороны системных программистов — такие программы создавать сложнее. Примером тому может служить существующая ситуация с драйверами печати. Так, в операционных системах Windows (и Windows 9x, и Windows NT/2000) печать через параллельный порт осуществляется не в режиме с прерываниями, как это сделано в других ОС, а в режиме опроса готовности, что приводит к 100-процентной загрузке центрального процессора на все время печати. При этом, естественно, выполняются и другие задачи, запущенные на исполнение, но исключительно за счет того, что упомянутые операционные системы поддерживают вытесняющую мультизадачность, время от времени прерывая процесс управления печатью и передавая центральный процессор остальным задачам.

Закрепление устройств, общие устройства ввода-вывода

Как известно, многие устройства и, прежде всего, устройства с последовательным доступом не допускают совместного использования. Такие устройства могут стать *закрепленными* за процессом, то есть их можно предоставить некоторому вычислительному процессу на все время жизни этого процесса. Однако это приводит к тому, что вычислительные процессы часто не могут выполняться параллельно — они ожидают освобождения устройств ввода-вывода. Чтобы организовать совместное использование многими параллельно выполняющимися задачами тех устройств ввода-вывода, которые не могут быть разделяемыми, вводится понятие *виртуальных устройств*. Принцип виртуализации позволяет повысить эффективность вычислительной системы.

Вообще говоря, понятие виртуального устройства шире, нежели понятие *спулинга* (spooling — Simultaneous Peripheral Operation On-Line, то есть имитация работы с устройством в режиме непосредственного подключения к нему). Основное назначение спулинга — создать видимость разделения устройства ввода-вывода, которое фактически является устройством с последовательным доступом и должно использоваться только монопольно и быть закрепленным за процессом. Например, мы уже говорили, что в случае, когда несколько приложений должны выводить на печать результаты своей работы, если разрешить каждому такому приложению печатать строку по первому же требованию, то это приведет к потоку строк,

не представляющих никакой ценности. Однако если каждому вычислительному процессу предоставлять не реальный, а виртуальный принтер, и поток выводимых символов (или управляющих кодов для их печати) сначала направлять в специальный файл на диске (так называемый *спул-файл* — spool-file) и только потом, по окончании виртуальной печати, в соответствии с принятой дисциплиной обслуживания и приоритетами приложений выводить содержимое спул-файла на принтер, то все результаты работы можно будет легко читать. Системные процессы, которые управляют спул-файлом, называются *спулером чтения* (spool-reader) или *спулером записи* (spool-writer).

Достаточно рационально организована работа с виртуальными устройствами в системах Windows 9x/NT/2000/XP компании Microsoft. В качестве примера можно кратко рассмотреть подсистему печати. Microsoft различает термины «принтер» и «устройство печати». Принтер — это некоторая виртуализация, объект операционной системы, а устройство печати — это физическое устройство, которое может быть подключено к компьютеру. Принтер может быть *локальным* или *сетевым*.

При установке локального принтера в операционной системе создается новый объект, связанный с реальным устройством печати через тот или иной интерфейс. Интерфейс может быть и сетевым, то есть передача управляющих кодов в устройство печати может осуществляться через локальную вычислительную сеть, однако принтер все равно будет считаться локальным.

Локальность принтера означает, что его спул-файл будет находиться на том же компьютере, что и принтер. Если же некоторый локальный принтер предоставить в сети в общий доступ с теми или иными разрешениями, то для других компьютеров и их пользователей он может стать *сетевым*. Компьютер, на котором имеется локальный принтер, предоставленный в общий доступ, называется *принт-сервером*.

Для получения управляющих кодов принтера устанавливается программное обеспечение (компания Microsoft называет его высокоуровневым драйвером, хотя правильнее было бы называть его иначе: например, препроцессором). Эти управляющие коды посылаются на устройство печати по соответствующему интерфейсу через назначенные принтеру порты и управляют работой устройства печати. При получении операционной системой от приложения запроса на печать она выделяет для этого процесса виртуальный принтер. Можно сказать, что операционная система закрепляет за процессом виртуальный принтер, но никак не устройство печати. Обработанные драйвером принтера данные, посланные на него из приложения, как правило (по умолчанию), направляются в спул-файл, откуда они затем передаются на печать по мере освобождения устройства печати и в соответствии с приоритетом локального принтера. При установке сетевого принтера операционная система устанавливает для этого объекта высокоуровневый драйвер и связывает полученный объект со спулером того компьютера, на котором установлен соответствующий локальный принтер.

Локальных принтеров, связанных с конкретным устройством печати, на компьютере может быть несколько. Каждому локальному принтеру можно назначить тот или иной приоритет, который будет учитываться при формировании очереди пе-

чати в процессе работы спулера. В результате каждый процесс может послать на печать свои данные и не связывать реальное выполнение некоторого задания на печать с занятостью или освобождением самого устройства печати. Приоритетность в печати определяется приоритетом того локального или сетевого принтера, к которому обратилось приложение.

Основные системные таблицы ввода-вывода

Для управления всеми операциями ввода-вывода и отслеживания состояния всех ресурсов, занятых в обмене данными, операционная система должна иметь соответствующие информационные структуры. Эти информационные структуры, прежде всего, призваны отображать следующую информацию:

- состав устройств ввода-вывода и способы их подключения;
- аппаратные ресурсы, закрепленные за имеющимися в системе устройствами ввода-вывода;
- логические (символьные) имена устройств ввода-вывода, используя которые вычислительные процессы могут запрашивать те или иные операции ввода-вывода;
- адреса размещения драйверов устройств ввода-вывода и области памяти для хранения текущих значений переменных, определяющих работу с этими устройствами;
- области памяти для хранения информации о текущем состоянии устройства ввода-вывода и параметрах, определяющих режимы работы устройства;
- данные о текущем процессе, который работает с данным устройством;
- адреса тех областей памяти, которые содержат данные, собственно и участвующие в операциях ввода-вывода (получаемые при операциях ввода данных и выводимые на устройство при операциях вывода данных).

Эти информационные структуры часто называют таблицами ввода-вывода, хотя они, в принципе, могут быть организованы и в виде списков.

Каждая операционная система ведет свои таблицы ввода-вывода, их состав (и количество, и назначение каждой таблицы) может сильно отличаться. В некоторых операционных системах вместо таблиц создаются списки, хотя использование статических структур данных для организации ввода-вывода, как правило, приводит к более высокому быстродействию. Здесь очень трудно вычленить общие составляющие, тем более что для современных операционных систем подробной документации на эту тему крайне мало, разве что воспользоваться материалами ныне устаревших ОС. Тем не менее попытаемся это сделать, опираясь на идеи семейства простых, но эффективных операционных систем реального времени, разработанных фирмой Hewlett Packard для своих мини-ЭВМ.

Исходя из принципа управления вводом-выводом исключительно через супервизор операционной системы и учитывая, что драйверы устройств ввода-вывода ис-

пользуют механизм прерываний для установления обратной связи центральной части с внешними устройствами, можно сделать вывод о необходимости создания по крайней мере трех системных таблиц.

Первая таблица (или список) содержит информацию обо всех устройствах ввода-вывода, подключенных к вычислительной системе. Назовем ее условно *таблицей оборудования* (equipment table), а каждый элемент этой таблицы пусть называется *UCB* (Unit Control Block — блок управления устройством ввода-вывода). Каждый элемент UCB таблицы оборудования, как правило, содержит следующую информацию об устройстве:

- тип устройства, его конкретная модель, символическое имя и характеристики устройства;
- способ подключения устройства (через какой интерфейс, к какому разъему, какие порты и линия запроса прерывания используются и т. д.);
- номер и адрес канала (и подканала), если такие используются для управления устройством;
- информация о драйвере, который должен управлять этим устройством, адреса секции запуска и секции продолжения драйвера;
- информация о том, используется или нет буферизация при обмене данными с устройством, «имя» (или просто адрес) буфера, если такой выделяется из системной области памяти;
- установка тайм-аута и ячейки для счетчика тайм-аута;
- состояние устройства;
- поле указателя для связи задач, ожидающих устройство;
- возможно, множество других сведений.

Поясним перечисленное. Поскольку во многих операционных системах драйверы могут обладать свойством *реентерабельности* (напомним, это означает, что один и тот же экземпляр драйвера может обеспечить параллельное обслуживание сразу нескольких однотипных устройств), то в элементе UCB должна храниться либо непосредственно сама информация о текущем состоянии устройства и сами переменные для реентерабельной обработки, либо указание на место, где такая информация может быть найдена. Наконец, важнейшим компонентом элемента таблицы оборудования является указатель на дескриптор той задачи, которая в настоящий момент использует данное устройство. Если устройство свободно, то поле указателя будет иметь нулевое значение. Если же устройство уже занято и рассматриваемый указатель не нулевой, то новые запросы к устройству фиксируются посредством образования списка из дескрипторов задач, ожидающих данное устройство.

Вторая таблица предназначена для реализации еще одного принципа виртуализации устройств ввода-вывода — принципа независимости от устройства. Желательно, чтобы программисту не приходилось учитывать конкретные параметры (и/или возможности) того или иного устройства ввода-вывода, которое установлено (или не установлено) в компьютер. Для него должны быть важными только самые общие возможности, характерные для данного класса устройств ввода-вывода. Например,

принтер должен уметь выводить (печатать) символы или графические изображения. А накопитель на магнитных дисках — считывать или записывать порцию данных по указанному адресу, то есть в координатах C-H-S (Cylinder-Head-Sector — номера цилиндра, головки и сектора) или по порядковому номеру блока данных. Хотя чаще всего программист и не использует прямую адресацию при работе с магнитными дисками, а работает на уровне файловой системы (см. главу 6). Однако в таком случае уже разработчики системы управления файлами не должны зависеть от того, каких типа и модели накопитель используется в данном компьютере и кто является его производителем (например, HDD IBM IC35L 120AVV207-0, WD1200JB или еще какой-нибудь). Важным должен быть только сам факт существования накопителя, имеющего некоторое количество цилиндров, головок чтения-записи и секторов на дорожке магнитного диска. Упомянутые значения количества цилиндров, головок и секторов должны быть взяты из элемента таблицы оборудования. При этом для программиста также не должно иметь значения, каким образом то или иное устройство подключено к вычислительной системе. Поэтому в запросе на ввод-вывод программист указывает именно *логическое имя устройства*. Действительное устройство, которое сопоставляется виртуальному (логическому), выбирается супервизором с помощью описываемой таблицы.

Итак, способ подключения устройства, его конкретная модель и соответствующий ей драйвер содержатся в уже рассмотренной таблице оборудования. Но для того чтобы связать некоторое виртуальное устройство, использованное программистом, с системной таблицей, отображающей информацию о том, какое конкретно устройство и каким образом подключено к компьютеру, требуется вторая системная таблица. Назовем ее условно *таблицей виртуальных логических устройств* (Device Reference Table, DRT). Назначение этой второй таблицы — установление связи между виртуальными (логическими) устройствами и реальными устройствами, описанными посредством первой таблицы (таблицы оборудования). Другими словами, вторая таблица позволяет супервизору перенаправить запрос на ввод-вывод из приложения в те программные модули и структуры данных, которые (или адреса которых) хранятся в соответствующем элементе первой таблицы. Во многих многопользовательских системах таких таблиц несколько: одна общая и по одной на каждого пользователя, что позволяет строить необходимые связи между логическими устройствами (символьными именами устройств) и реальными физическими устройствами, которые имеются в системе.

Наконец, третья таблица — *таблица прерываний* — необходима для организации обратной связи между центральной частью и устройствами ввода-вывода. Эта таблица указывает для каждого сигнала запроса на прерывание тот элемент UCB, который сопоставлен данному устройству. Каждое устройство либо имеет свою линию запроса на прерывание, либо разделяет линию запроса на прерывание с другими устройствами, но при этом имеется механизм второго уровня адресации устройств ввода-вывода. Таким образом, таблица прерываний отображает связи между сигналами запроса на прерывания и самими устройствами ввода-вывода. Как и системная таблица ввода-вывода, таблица прерываний в явном виде может и не присутствовать. Другими словами, можно сразу из основной таблицы прерываний

компьютера передать управление на программу обработки (драйвер), связанную с элементом UCB. Важно наличие связи между сигналами прерываний и таблицей оборудования.

В ряде сложных операционных систем, а к ним следует отнести все современные 32-разрядные системы для персональных компьютеров, имеется гораздо больше системных таблиц или списков, используемых для организации управления операциями ввода-вывода. Например, одной из возможных и часто реализуемых информационных структур, сопровождающих практически каждый запрос на ввод-вывод, является *блок управления данными* (Data Control Block, DCB). Назначение DCB — подключение препроцессоров к процессу подготовки данных на ввод-вывод, то есть учет конкретных технических характеристик и используемых преобразований. Это необходимо для того, чтобы имеющееся устройство получало не какие-то непонятные ему коды или форматы данных, не соответствующие режиму его работы, а коды и форматы, созданные специально под данное устройство. Теперь такие препроцессоры часто называют высокоуровневыми драйверами, или просто драйверами, хотя изначально под термином «драйвер» подразумевалась программа управления операциями ввода-вывода.

Взаимосвязи между описанными таблицами изображены на рис. 5.2.

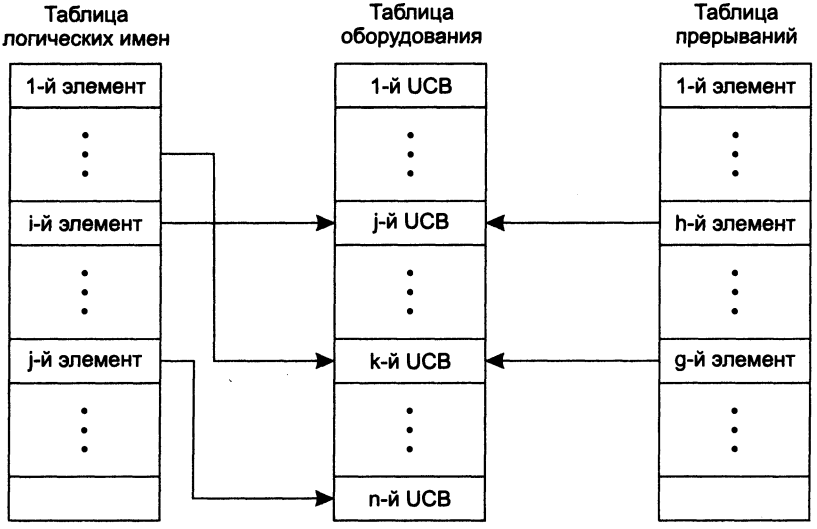


Рис. 5.2. Взаимосвязи системных таблиц ввода-вывода

Нам осталось рассмотреть процесс управления вводом-выводом еще раз, теперь с учетом изложенных принципов (рис. 5.3).

Запрос на операцию ввода-вывода от выполняющейся программы поступает на супервизор задач (шаг 1). Этот запрос представляет собой обращение к операционной системе и указывает на конкретную функцию API. Вызов сопровождается некоторыми параметрами, уточняющими требуемую операцию. Модуль операционной системы, принимающий от задач запросы на те или иные действия, часто

называют *супервизором задач*. Не следует путать его с диспетчером задач. Супервизор задач проверяет системный вызов на соответствие принятым спецификациям и в случае ошибки возвращает задаче соответствующее сообщение (шаг 1-1). Если же запрос корректен, то он перенаправляется в *супервизор ввода-вывода* (шаг 2). Последний по логическому (виртуальному) имени с помощью таблицы DRT находит соответствующий элемент UCB в таблице оборудования. Если устройство уже занято, то описатель задачи, запрос которой обрабатывается супервизором ввода-вывода, помещается в список задач, ожидающих это устройство. Если же устройство свободно, то супервизор ввода-вывода определяет из UCB тип устройства и при необходимости запускает препроцессор, позволяющий получить последовательность управляющих кодов и данных, которую сможет правильно понять и отработать устройство (шаг 3). Когда «программа» управления операцией ввода-вывода будет готова, супервизор ввода-вывода передает управление соответствующему драйверу на секцию запуска (шаг 4). Драйвер инициализирует операцию управления, обнуляет счетчик тайм-аута и возвращает управление супервизору (диспетчеру задач) с тем, чтобы он поставил на процессор готовую к исполнению задачу (шаг 5). Система работает своим чередом, но когда устройство ввода-вывода отработает посланную ему команду, оно выставляет сигнал запроса на прерывание, по которому через таблицу прерываний управление передается на секцию продолжения (шаг 6). Получив новую команду, устройство вновь начинает ее обрабатывать, а управление процессором опять передается диспетчеру задач, и процессор продолжает выполнять полезную работу. Таким образом, получается параллельная обработка задач, на фоне которой процессор осуществляет управление операциями ввода-вывода.

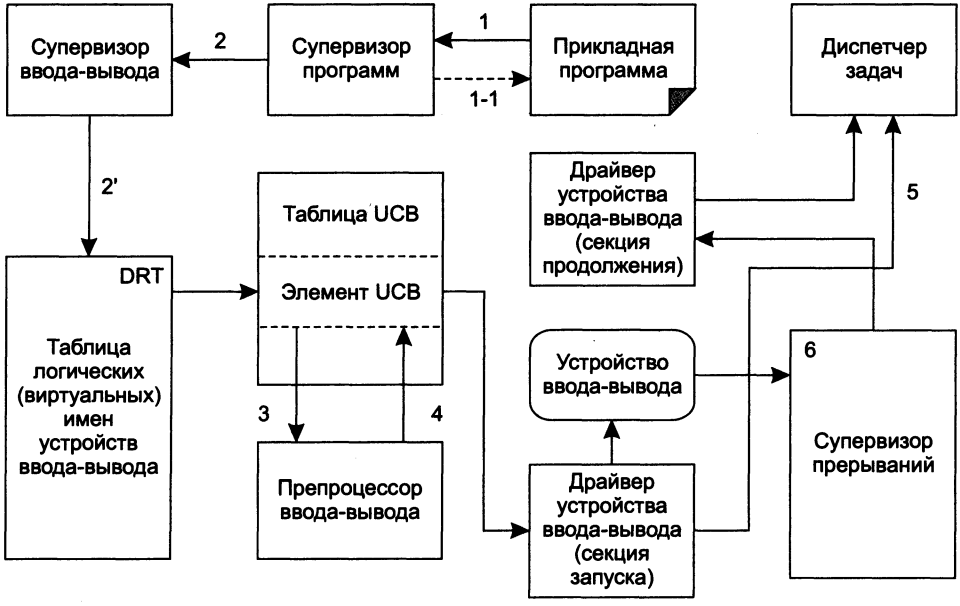


Рис. 5.3. Процесс управления вводом-выводом

Очевидно, что если имеются специальные аппаратные средства для управления вводом-выводом (речь идет о каналах прямого доступа к памяти), которые позволяют освободить центральный процессор от этой работы, то в функции центрального процессора будут по-прежнему входить все только что рассмотренные шаги, за исключением последнего — непосредственного управления операциями ввода-вывода. В случае использования каналов прямого доступа в память последние исполняют соответствующие канальные программы и освобождают центральный процессор от непосредственного управления обменом данными между памятью и внешними устройствами.

Синхронный и асинхронный ввод-вывод

Задача, выдавшая запрос на операцию ввода-вывода, переводится супервизором в состояние ожидания завершения заказанной операции. Когда супервизор получает от секции завершения сообщение о том, что операция завершилась, он переводит задачу в состояние готовности к выполнению, и она продолжает выполняться. Эта ситуация соответствует *синхронному вводу-выводу*. Синхронный ввод-вывод является стандартным для большинства операционных систем. Чтобы увеличить скорость выполнения приложений, было предложено при необходимости использовать *асинхронный ввод-вывод*.

Простейшим вариантом *асинхронного вывода* является так называемый *буферизованный* вывод данных на внешнее устройство, при котором данные из приложения передаются не непосредственно на устройство ввода-вывода, а в специальный системный буфер — область памяти, отведенную для временного размещения передаваемых данных. В этом случае логически операция вывода для приложения считается выполненной сразу же, и задача может не ожидать окончания действительного процесса передачи данных на устройство. Реальным выводом данных из системного буфера занимается супервизор ввода-вывода. Естественно, что выделение буфера из системной области памяти берет на себя специальный системный процесс по указанию супервизора ввода-вывода. Итак, для рассмотренного случая вывод будет асинхронным, если, во-первых, в запросе на ввод-вывод указано на необходимость буферизации данных, а во-вторых, устройство ввода-вывода допускает такие асинхронные операции, и это отмечено в UCB.

Можно организовать и *асинхронный ввод данных*. Однако для этого необходимо не только выделять область памяти для временного хранения считываемых с устройства данных и связывать выделенный буфер с задачей, заказавшей операцию, но и сам запрос на операцию ввода-вывода разбивать на две части (на два запроса). В первом запросе указывается операция на считывание данных, подобно тому как это делается при синхронном вводе-выводе, однако тип (код) запроса используется другой, и в запросе указывается еще по крайней мере один дополнительный параметр — имя (код) системного объекта, которое получает задача в ответ на запрос и которое идентифицирует выделенный буфер. Получив имя буфера (будем так условно называть этот системный объект, хотя в различных операционных системах используются и другие термины, например «класс»), задача продолжает

свою работу. Здесь очень важно подчеркнуть, что в результате запроса на асинхронный ввод данных задача не переводится супервизором ввода-вывода в состояние ожидания завершения операции ввода-вывода, а остается в состоянии выполнения или в состоянии готовности к выполнению. Через некоторое время, выполнив необходимый код, который был определен программистом, задача выдает второй запрос на завершение операции ввода-вывода. В этом втором запросе к тому же устройству, который, естественно, имеет другой код (или имя запроса), задача указывает имя системного объекта (буфера для асинхронного ввода данных) и в случае успешного завершения операции считывания данных тут же получает их из системного буфера. Если же данные еще не успели до конца переписаться с внешнего устройства в системный буфер, супервизор ввода-вывода переводит задачу в состояние ожидания завершения операции ввода-вывода, и далее все напоминает обычный синхронный ввод данных.

Асинхронный ввод-вывод характерен для большинства мультипрограммных операционных систем, особенно если операционная система поддерживает мультизадачность с помощью механизма потоков выполнения. Однако если асинхронный ввод-вывод в явном виде отсутствует, его можно реализовать самому, организовав для вывода данных отдельный поток выполнения.

Аппаратуру ввода-вывода можно рассматривать как совокупность аппаратных процессоров, которые способны работать параллельно друг другу, а также параллельно центральному процессору (процессорам). На таких «процессорах» выполняются так называемые *внешние процессы*. Например, для печатающего устройства (внешнее устройство вывода данных) внешний процесс может представлять собой совокупность операций, обеспечивающих перевод печатающей головки, продвижение бумаги на одну позицию, смену цвета чернил или печать каких-то символов. Внешние процессы, используя аппаратуру ввода-вывода, взаимодействуют как между собой, так и с обычными «программными» процессами, выполняющимися на центральном процессоре. Важным при этом является обстоятельство, что скорости выполнения внешних процессов будут существенно (порой на порядок или больше) отличаться от скорости выполнения обычных (*внутренних*) процессов. Для своей нормальной работы внешние и внутренние процессы обязательно должны синхронизироваться. Для сглаживания эффекта значительного несоответствия скоростей между внутренними и внешними процессами используют упомянутую выше буферизацию. Таким образом, можно говорить о системе параллельных взаимодействующих процессов (см. главу 7).

Буферы (буфер) являются критическим ресурсом в отношении внутренних (программных) и внешних процессов, которые при параллельном своем развитии информационно взаимодействуют. Через буферы данные либо посылаются от некоторого процесса к адресуемому внешнему (операция вывода данных на внешнее устройство), либо от внешнего процесса передаются некоторому программному процессу (операция считывания данных). Введение буферизации как средства информационного взаимодействия выдвигает проблему управления этими системными буферами, которая решается средствами супервизорной части операционной системы. При этом на супервизор возлагаются задачи не только по выделению и освобождению буферов в системной области памяти, но и по синхронизации

процессов в соответствии с состоянием операций заполнения или освобождения буферов, а также по их ожиданию, если свободных буферов в наличии нет, а запрос на ввод-вывод требует буферизации. Обычно супервизор ввода-вывода для решения перечисленных задач использует стандартные средства синхронизации, принятые в данной операционной системе. Поэтому если операционная система имеет развитые средства для решения проблем параллельного выполнения взаимодействующих приложений и задач, то, как правило, она реализует и асинхронный ввод-вывод.

Организация внешней памяти на магнитных дисках

Для организации внешней памяти желательно использовать относительно недорогие, но достаточно быстродействующие и емкие устройства с прямым доступом к данным. К таким устройствам, прежде всего, относятся накопители на жестких магнитных дисках (НЖМД). Нынче чаще всего такие накопители называют «винчестерами», но мы не будем употреблять это название.

Детальное изучение этих устройств выходит за рамки темы настоящей книги, в основном их изучают в рамках дисциплины «Устройства ввода-вывода». Однако поскольку большинство компьютеров имеет накопители на жестких магнитных дисках и фактически ни одна современная операционная система для повсеместно распространенных персональных компьютеров не обходится без дисковой подсистемы, мы ознакомимся с логической организацией хранения и доступа к данным в этих устройствах, причем применительно к персональным компьютерам.

Действительно, дисковая подсистема для большинства компьютеров является одной из важнейших. Именно на магнитных дисках чаще всего располагается загружаемая в компьютер операционная система, которая и обеспечивает нам удобный интерфейс для работы. Благодаря использованию систем управления файлами, данные на магнитных дисках располагаются в виде именованных наборов данных, называемых файлами. Таким образом, помимо файлов самой операционной системы, на дисках располагаются многочисленные прикладные программы и разнообразные файлы пользователей. Наконец, благодаря тому, что по сравнению с другими устройствами внешней памяти дисковые механизмы обладают большими быстродействием и вместительностью, а также средствами непосредственной (прямой) адресации блоков данных, дисковую подсистему часто используют для организации механизмов виртуальной памяти, что существенно расширяет возможности компьютера.

Основные понятия

Из оперативной памяти в НЖМД и обратно информация передается байтами, а вот записывается на диск и считывается с него она уже последовательно (побитно). Из-за того что запись и считывание бита данных не являются абсолютно надежными операциями, информация перед записью кодируется с достаточно большой

избыточностью. Для этой цели применяют коды Рида–Соломона. Избыточное кодирование информации позволяет не только обнаруживать ошибки, но и автоматически исправлять их. Следовательно, перед тем как данные, считанные с поверхности магнитного диска, будут переданы в оперативную память, их нужно предварительно обработать (перекодировать). На эту операцию необходимо время, поэтому в ходе обработки данных быстро вращающийся диск успевает повернуться на некоторый угол, и мы можем констатировать, что на магнитном диске данные располагаются не сплошь, а порциями (блоками). Говорят, что НЖМД относится к блочным устройствам. Нельзя прочитать (или записать) байт или несколько байтов. Можно прочитать сразу только блок данных и уже потом извлекать из него нужные байты, использовать их в своих вычислениях и изменять. Записать потом данные обратно тоже можно только сразу блоком.

За счет того что при вращении диска магнитная головка, зафиксированная на некоторое время в определенном положении, образует окружность (*дорожку* — track), блоки данных на таких окружностях называют *секторами* (sectors). С некоторых пор размер сектора стал стандартным и в абсолютном большинстве случаев он равен 512 байт хранимых данных. Все сектора пронумерованы, и помимо данных пользователя на магнитных дисках размещается и служебная информация, с помощью которой можно находить искомый сектор. Служебная информация (сервоинформация), как правило, располагается в межсекторных промежутках.

Группы дорожек (треков) одного радиуса, расположенные на поверхностях магнитных дисков, образуют так называемые *цилиндры* (cylinders). Современные жесткие диски могут иметь по нескольку десятков тысяч цилиндров. Выбор конкретной дорожки в цилиндре осуществляется указанием порядкового номера той *головки* (head) *чтения/записи данных*, которая и образует эту дорожку. Таким образом, адрес конкретного блока данных указывается с помощью уже упоминавшихся трех координат C-H-S — номеров цилиндра, головки и сектора. Устройство управления НЖМД обеспечивает позиционирование блока головок на нужный цилиндр, выбирает заданную поверхность и находит требуемый сектор. Этот способ адресации нынче считается устаревшим и почти не используется. Второй способ адресации блоков данных основывается на том, что все блоки (секторы) пронумерованы.

Логическая структура магнитного диска

Для того чтобы можно было загрузить с магнитного диска операционную систему, а уже с ее помощью организовать работу с файлами, были приняты специальные системные соглашения о структуре диска. Хранение данных на магнитном диске можно организовать различными способами. Можно поделить все дисковое пространство на несколько частей — *разделов* (partitions), а можно его и не делить. Деление НЖМД на разделы позволяет организовать на одном физическом устройстве несколько логических; в этом случае говорят о *логических дисках*. Следует, однако, заметить, что не во всех операционных системах используется понятие логического диска. Так, UNIX-системы не имеют логических дисков.

Разделение всего дискового пространства на разделы полезно по нескольким соображениям. Во-первых, это структурирует хранение данных. Например, выделе-

ние отдельного раздела под операционную систему и программное обеспечение и другого раздела под данные пользователей позволяет отделить последние от системных файлов и не только повысить надежность системы, но и сделать более удобным ее обслуживание. Во-вторых, на каждом разделе может быть организована своя файловая система, что иногда бывает необходимо. Например, при установке операционной системы Linux нужно иметь не менее двух разделов[1], поскольку файл подкачки (страничный файл) должен располагаться в отдельном разделе. Наконец, в ряде случаев на компьютере может потребоваться установка более одной операционной системы.

Для того чтобы системное программное обеспечение получило информацию о том, как организовано хранение данных на каждом конкретном накопителе, нужно разместить в одном из секторов соответствующие данные. Даже если НЖМД используется как единственный *логический диск*, все равно нужно указать, что имеется всего один диск, и его размер. Структура данных, несущая информацию о логической организации диска, вместе с небольшой программой, с помощью которой можно ее проанализировать, а также найти и загрузить в оперативную память программу загрузки операционной системы, получила название *главной загрузочной записи* (Master Boot Record, MBR). MBR располагается в самом первом секторе НЖМД, то есть в секторе с координатами 0-0-1. Программа, расположенная в MBR, носит название *внесистемного загрузчика* (Non-System Bootstrap, NSB).

Вследствие того что сектор состоит только из 512 байт и помимо программы в нем должна располагаться информация об организации диска, внесистемный загрузчик очень прост, а структура данных, называемая *таблицей разделов* (Partition Table, PT), занимает всего 64 байт. Таблица разделов располагается в MBR по смещению 0x1BE и содержит четыре элемента. Структура записи элемента таблицы разделов приведена в табл. 5.1. Каждый элемент этой таблицы описывает один раздел, причем двумя способами: через координаты C-H-S начального и конечного секторов, а также через номер первого сектора в спецификации LBA[2] (Logical Block Addressing) и общее число секторов в разделе. Важно отметить, что каждый раздел начинается с первого сектора на заданных цилиндре и поверхности и имеет размер не менее одного цилиндра. Поскольку координаты MBR равны 0-0-1, то первый сектор первого раздела в большинстве случаев получается равным 0-1-1 (в координатах LBA это будет сектор 64).

Первым байтом в элементе таблицы разделов идет флаг *активности раздела* Boot Indicator (значение 0 — не активен, 128 ($80_{(h)}$) — активен). Он позволяет определить, является ли данный раздел системным загрузочным. В результате процесс загрузки операционной системы осуществляется путем загрузки первого сектора

---

[1] Практика показывает, что Linux и другие UNIX-подобные системы лучше всего устанавливать, разбив НЖМД на 6 разделов. Раздел подкачки (swap partition) служит для размещения файла подкачки. К основному (корневому) разделу, обозначаемому символом /, монтируются разделы /usr, /home, /var и /boot. Такое разбиение диска на разделы считается наиболее технологичным.

[2] Способ указания блока данных, согласно которому все секторы диска считаются пронумерованными по следующему правилу: LBA = c × H + h) × S + s – 1. Здесь H — это максимальное число рабочих поверхностей в цилиндре; S — количество секторов на одной дорожке; c, h и s — «координаты» искомого сектора.

с такого активного раздела и передачи управления на расположенную в нем программу, которая и продолжает загрузку. Активным может быть только один раздел, и это обычно проверяется программой NSB, расположенной в MBR.

Таблица 5.1. Формат элемента таблицы разделов

Название записи элемента таблицы разделов	Длина, байт
Флаг активности раздела	1
Номер головки начала раздела	1
Номера сектора и цилиндра загрузочного сектора раздела	2
Кодовый идентификатор операционной системы	1
Номер головки конца раздела	1
Номера сектора и цилиндра последнего сектора раздела	2
Младшее и старшее двухбайтовые слова относительного номера начального сектора	4
Младшее и старшее двухбайтовые слова размера раздела в секторах	4

За флагом активности раздела следует байт номера головки, с которой начинается раздел. За ним следуют два байта, означающие соответственно номер сектора и номер цилиндра загрузочного сектора, где располагается первый сектор загрузчика операционной системы. Затем следует кодовый идентификатор System ID (длиной в один байт), указывающий на принадлежность данного раздела к той или иной операционной системе и на установку в этом разделе соответствующей файловой системы. Поскольку крайне сложно найти информацию по этим кодовым идентификаторам, которыми помечаются разделы дисков, в табл. 5.2 приведены не полтора десятка наиболее часто встречающихся, а все известные сигнатуры (кодовые идентификаторы).

Таблица 5.2. Кодовые идентификаторы разделов диска

Код	Описание	Код	Описание
000h	Раздел не использован	085h	Linux Extended, XOSL
001h	FAT12	086h	FAT16 volume set
002h	Xenix root	087h	NTFS volume set
003h	Xenix /usr	08Ah	AiR-Boot
004h	FAT16 (<32Mb)	08Bh	FAT32 volume set
005h	Extended	08Ch	FAT32 LBA volume set
006h	FAT16	08Dh	Free FDISK FAT12
007h	NTFS, HPFS	08Eh	Linux LVM
008h	AIX Boot	090h	Free FDISK FAT16 (<32Mb)
009h	AIX Data	091h	Free FDISK Extended
00Ah	OS/2 Boot Manager	092h	Free FDISK FAT16
00Bh	FAT32	093h	Amoeba native

Код	Описание	Код	Описание
00Ch	FAT32 LBA	094h	Amoeba BBT
00Eh	FAT16 LBA	095h	MIT EXOPC
00Fh	Extended LBA	097h	Free FDISK FAT32
010h	Opus	098h	Free FDISK FAT32 LBA
011h	Hidden FAT12	099h	DCE376
012h	Compaq Setup	09Ah	Free FDISK FAT16 LBA
013h	B-TRON	09Bh	Free FDISK Extended LBA
014h	Hidden FAT16 (<32Mb)	09Fh	BSDI
016h	Hidden FAT16	0A0h	Laptop hibernation
017h	Hidden NTFS, HPFS	0A1h	NEC hibernation
018h	AST Windows Swap	0A5h	Free BSD, BSD/386
019h	Photon	0A6h	Open BSD
01Bh	Hidden FAT32	0A7h	NextStep
01Ch	Hidden FAT32 LBA	0A8h	Apple UFS
01Eh	Hidden FAT16 LBA	0A9h	Net BSD
020h	OFS1	0Aah	Olivetti service
022h	Oxygen	0Abh	Apple Booter
024h	NEC DOS	0Aeh	ShagOS native
035h	OS/2 JFS	0Afh	ShagOS swap
035h	Theos 3.x	0B0h	BootStar Dummy
039h	Theos 4.x spanned, Plan9	0B7h	BSDI old native
03Ah	Theos 4.x 4G	0B8h	BSDI old swap
03Bh	Theos 4.x Extended	0BBh	OS Selector
03Ch	Partition Magic	0Beh	Solaris 8 boot
040h	Venix 80286	0C0h	CTOS, REAL/32 smal
041h	Minix, PPC Boot	0C1h	DR-DOS FAT12
042h	LinuxSwp/DR-DOS, SFS, Win2K DDM	0C6h	DR-DOS FAT16, FAT16 set corrupt
043h	LinuxNat/DR-DOS	0C2h	Hidden Linux swap
045h	Eumel/Ergos 45h, Boot-US	0C3h	Hidden Linux native
046h	Eumel/Ergos 46h	0C4h	DR-DOS FAT16 (<32Mb)
047h	Eumel/Ergos 47h	0C7h	Syrinx boot, NTFS set corrupt
048h	Eumel/Ergos 48h	0CBh	DR-DOS FAT32
04Dh	QNX 4.x first	0CCh	DR-DOS FAT32 LBA
04Eh	QNX 4.x second	0CDh	CTOS memdump
04Fh	QNX 4.x third, Oberon	0Ceh	DR-DOS FAT16 LBA
050h	OnTrack DM R/O, Lynx RTOS	0D0h	REAL/32 big

продолжение ⇩

Таблица 5.2 (*продолжение*)

Код	Описание	Код	Описание
051h	DM6 Aux1, DM R/W	0D1h	Multiuser DOS FAT12
052h	CP/M, Microport System V	0D4h	Multiuser DOS FAT16 (<32Mb)
053h	OnTrack DM6 Aux3	0D5h	Multiuser DOS Extended
054h	OnTrack DM6 DD0	0D6h	Multiuser DOS FAT16
055h	EZ-Drive	0D8h	CP/M-86
056h	GoldenBow Vfeature	0DBh	Concurrent DOS, CTOS
057h	Drive Pro	0DDh	Hidden CTOS memdump
05Ch	Priam Edisk	0DFh	DG/UX
061h	Speed Stor	0E0h	ST AVFS
063h	Unix	0E1h	Speed Stor FAT32
064h	NetWare 2.x, PC-ARMOUR	0E3h	Speed Stor R/O
065h	NetWare 3.x	0E4h	Speed Stor FAT16
067h	Novell 67h	0Ebh	BeOS
068h	Novell 68h	0Eeh	EFI header
068h	Novell 69h	0Efh	EFI file system
070h	DiskSecure Multi-Boot	0F0h	Linux/PA-RISC boot
074h	ScramDisk	0F1h	Storage Dimensions
075h	PC/AX	0F2h	DOS Secondary
078h	XOSL	0F4h	Speed Stor large, Prologue singl
07Eh	F.I.X	0F5h	Prologue multi
080h	MINIX 1.1-1.4a	0FBh	VMware native
081h	MINIX 1.4b+, ADM	0FCh	Vmware swap
082h	Linux swap, Solaris	0FDh	Linux RAID
083h	Linux native	0Feh	Speed Stor (>1024), LanStep
084h	Hibernation, OS/2 C: Hidden	0FFh	Xenix BBT

Можно сказать, что таблица разделов — одна из наиболее важных структур данных на жестком диске. Если эта таблица повреждена, то не только не будет загружаться ни одна из установленных на компьютере операционных систем, но станут недоступными данные, расположенные в НЖМД, особенно если жесткий диск был разбит на несколько разделов.

Последние два байта MBR имеют значение $55AA_{(h)}$, то есть чередующиеся значения 0 и 1. Эта сигнатура выбрана для того, чтобы проверить работоспособность всех линий передачи данных. Значение $55AA_{(h)}$, присвоенное последним двум байтам, имеется во всех загрузочных секторах.

Разделы диска могут быть двух типов: *первичные* (primary) и *расширенные* (extended). Максимальное число первичных разделов равно четырем. Если первичных разделов несколько, то только один из них может быть активным. Именно загрузчику, расположенному в активном разделе, передается управление при вклю-

чении компьютера с помощью внесистемного загрузчика. Для DOS-систем и иных операционных систем, использующих спецификации DOS, остальные первичные разделы в этом случае считаются невидимыми (hidden). Так ведут себя и операционные системы Windows 9x.

Согласно принятым спецификациям на одном жестком диске может быть только один расширенный раздел, который, в свою очередь, может быть разделен на большое количество подразделов — *логических дисков* (logical disks). В этом смысле термин «первичный» можно признать не совсем удачным переводом слова «primary» — лучше было бы перевести «простейший», или «примитивный». В этом случае становится понятным и логичным термин «расширенный». Расширенный раздел содержит вторичную запись MBR (Secondary MBR, SMBR), в состав которой вместо таблицы разделов входит аналогичная ей *таблица логических дисков* (Logical Disks Table, LDT). Таблица LDT описывает размещение и характеристики раздела, содержащего единственный логический диск, а также может специфицировать следующую запись SMBR. Следовательно, если в расширенном разделе создано K логических дисков, то он содержит K экземпляров SMBR, связанных в список. Каждый элемент этого списка описывает соответствующий логический диск и ссылается (кроме последнего) на следующий элемент списка.

Как мы уже сказали, загрузчик NSB служит для поиска с помощью таблицы разделов активного раздела, копирования в оперативную память компьютера системного загрузчика (System Bootstrap, SB) из выбранного раздела и передачи на него управления, что позволяет осуществить загрузку ОС.

Вслед за сектором MBR размещаются собственно сами разделы (рис. 5.4). В процессе начальной загрузки сектора MBR, содержащего таблицу разделов, работают программные модули BIOS. Начальная загрузка считается выполненной корректно только в том случае, если таблица разделов содержит допустимую информацию.

Рассмотрим еще раз процесс загрузки операционной системы. Процедура начальной загрузки (bootstrap loader) вызывается как программное прерывание (BIOS INT 19h). Эта процедура определяет первое готовое устройство из списка разрешенных и доступных (гибкий или жесткий диск, а в современных компьютерах это могут быть еще и компакт-диск, привод ZIP-drive компании Iomega, сетевой адаптер или еще какое-нибудь устройство) и пытается загрузить с него в оперативную память короткую главную программу-загрузчик. Для накопителей на жестких магнитных дисках — это уже известный нам главный, или внесистемный, загрузчик (NSB) из MBR, и ему передается управление. Главный загрузчик определяет на диске активный раздел, загружает его собственный системный загрузчик и передает управление ему. И наконец, этот загрузчик находит и загружает необходимые файлы операционной системы и передает ей управление. Далее операционная система выполняет инициализацию подведомственных ей программных и аппаратных средств. Она добавляет новые сервисы, вызываемые, как правило, тоже через механизм программных прерываний, и расширяет (или заменяет) некоторые сервисы BIOS. Необходимо отметить, что в современных мультипрограммных операционных системах большинство сервисов BIOS, изначально расположенных в ПЗУ, как правило, заменяются собственными драйверами ОС,

поскольку они должны работать в режиме прерываний, а не в режиме сканирования готовности.

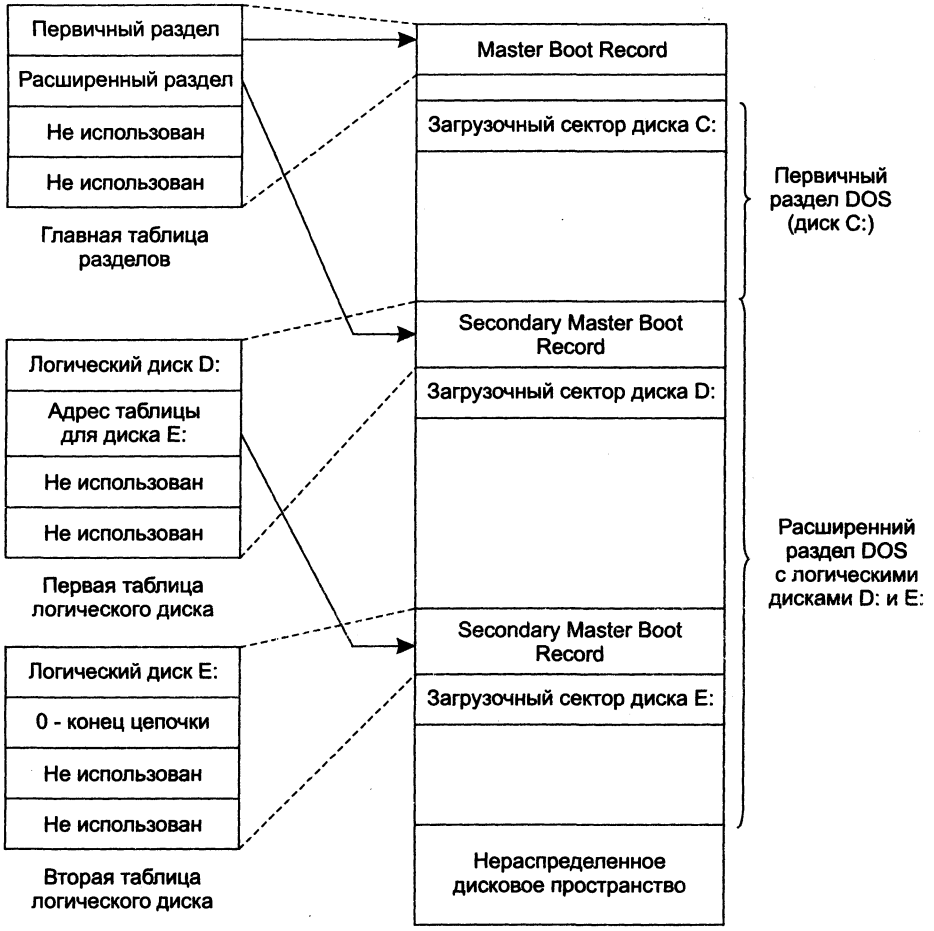


Рис. 5.4. Разбиение диска на разделы

Согласно рассмотренному процессу, каждый раз при запуске компьютера будет загружаться одна и та же операционная система. Это не всегда нас может устраивать. Так называемые *менеджеры загрузки* (boot managers) предназначены для того, чтобы пользователь мог выбрать среди нескольких установленных на компьютере операционных систем желаемую и передать управление на загрузчик выбранной ОС. Имеется большое количество таких менеджеров. Одним из наиболее мощных менеджеров загрузки является OS Selector от фирмы Acronis. Эта программа имеет следующие основные особенности:

- поддержка большого количества операционных систем, включая различные версии DOS (MS DOS, DR-DOS и др.), Windows (9x/ME, NT/2000/XP), OS/2, Linux, FreeBSD, SCO Unix, BeOS и др.;

- возможность установки на любой раздел FAT16/FAT32, в том числе и на отдельный раздел, недоступный другим операционным системам;
- возможность с помощью меню загрузки, предоставляемого менеджером, осуществить загрузку с дискеты;
- автоматическая идентификация операционных систем как на первичных разделах, так и на логических дисках расширенного раздела всех НЖМД, доступных через BIOS компьютера;
- поддержка нескольких операционных систем на одном разделе FAT16/FAT32, при этом предотвращаются конфликты по системным и конфигурационным файлам для систем, установленных на одном разделе;
- возможность дополнительной настройки конфигураций операционных систем и легкого их добавления и удаления;
- встроенная защита от загрузочных вирусов;
- легкое восстановление в случае повреждения MBR;
- поддержка больших жестких дисков во всех режимах современных подсистем BIOS;
- возможность установки паролей отдельно на меню загрузки и на выбранные конфигурации.

Формирование таблицы разделов осуществляется с помощью специальных утилит. Обычно их называют FDisk (от слов «Form Disk» — формирование диска). Хотя есть и иные программы, которые могут делать с разделами намного больше, чем простейшие утилиты FDisk от Microsoft. Надо признать, что в последнее время появилось большое количество утилит, которые предоставляют возможность более наглядно представить разбиение диска на разделы, поскольку в них используется графический интерфейс. Эти программы успешно и корректно работают с наиболее распространенными типами разделов (разделы под FAT, FAT32, NTFS). Однако созданы они в основном для работы в среде Win32API, что часто ограничивает возможность их применения. Одной из самых известных и мощных программ для работы с разделами жесткого диска является Partition Magic фирмы Power Quest.

Еще одной мощной утилитой такого рода является Администратор дисков, входящий в состав уже упоминавшегося менеджера загрузки OS Selector от Acronis. Эта утилита позволяет:

- создавать разделы любых типов и форматировать их под файловые системы FAT16, FAT32, NTFS, Ext2FS (Linux), Linux ReiserFS, Linux Swap, при этом можно выбирать точное или произвольное расположение раздела и указывать его параметры;
- получать подробную информацию о разделах и о самих жестких дисках;
- удалять любые разделы;
- преобразовывать разделы из FAT16 в FAT32 и обратно;
- копировать и перемещать разделы с FAT16, FAT32, NTFS, Linux Ext2FS, Linux ReiserFS и Linux Swap;

- изменять размеры разделов с вышеперечисленными файловыми системами;
- выбирать размер кластера вручную во время любой операции создания, копирования, перемещения или изменения размера раздела;
- посекторно редактировать содержимое жестких дисков и разделов с помощью встроенного многооконного редактора дисков.

В популярных операционных системах от Microsoft тоже имеются средства для просмотра и изменения структуры разделов жесткого диска. Так, в Windows NT 4.0 для управления дисками имеется программа Администратор дисков (Disk Manager), а в Windows 2000 и Windows XP — консоль управления с оснасткой под названием Управление дисками (Disk Management). Эти средства имеют графический интерфейс и позволяют создавать новые разделы, удалять разделы, переопределять букву (имя) логического диска и создавать наборы дисков, выступающие как один логический том.

Утилиты формирования дисков, входящие в состав MS DOS и Windows 95/98, а также утилита, встроенная в программу установки Windows NT, первым элементом таблицы разделов всегда делают первичный раздел. Вторым элементом становится расширенный раздел, в котором, в свою очередь, организуется один или несколько логических дисков. При этом создаваемые логические диски помимо известного буквенного именования (диски C:, D:, E: и т. д.) получают еще и так называемые номера разделов. Диск C: получает в этом случае порядковый номер 1, диск D: — 2, диск E: — 3, и т. д. Именно номера разделов используются в файле boot.ini, который указывает системному загрузчику Windows NT/2000/XP, где находятся файлы выбранной операционной системы.

Следует заметить, что в операционных системах типа Linux логические диски и разделы нумеруются и обозначаются иным способом. Жесткий диск с IDE-интерфейсом, подключенный к первому контроллеру как главный (master), имеет имя hda. Если это второй диск на том же шлейфе, то его именуют hdb[1]. Соответственно, имя hdc будет соответствовать диску, подключенному ко второму порту контроллера и имеющему адрес 0, то есть главному. И так далее. Если раздел диска указан посредством таблицы из MBR, то он имеет номер элемента таблицы разделов. Если же речь идет о логических дисках, созданных в пределах расширенного раздела, то их номера уже начинаются с 5. Тем самым указывается, что раздел описан в следующей (вторичной) записи MBR, то есть в SMBR.

Так, для рассматриваемого нами примера (см. рис. 5.4), раздел с номером 1 в Linux тоже будет иметь номер 1. Если мы имеем единственный накопитель, подключенный к первому порту контроллера, то этот раздел обозначается как hda1. А вот логический диск, по умолчанию именуемый в Windows диском D: и имеющий номер раздела 2, в Linux будет обозначаться как hda5. Логический диск E:, имеющий в Windows номер раздела 3, станет в Linux диском с номером раздела 6 и будет обозначаться hda6. Чтобы понять причину такой нумерации, рассмотрим рис. 5.4

[1] Главным является тот накопитель, который имеет адресацию 0 на IDE-интерфейсе, тогда как диск с адресом 1 обозначается как вспомогательный (slave). Адресация выставляется на одной из линий IDE-шлейфа (26 линия).

более внимательно. Вслед за сектором с MBR размещаются собственно сами разделы. Поскольку на рисунке это в явном виде не показано, напомним, что любой раздел начинается с первого сектора. В таблице разделов имеется 4 элемента, но только два из них задействованы. Первый элемент описывает раздел с номером 1 и ему соответствует логический диск C:. Второй элемент указывает на запись SMBR, в которой первый элемент в таблице логических дисков описывает логический диск D:. И этот элемент является уже пятым элементом, если учесть четыре элемента в MBR. А далее нумерация разделов в Linux отходит от этой идеи. Диск E: получает порядковый номер 6, а не 9, как следовало бы ожидать, если подсчитывать все имеющиеся элементы в таблицах разделов. И это логично, поскольку в каждой таблице дисков логический диск описывает только один элемент — первый. Таким образом, если бы расширенный раздел был разбит не на два, а на три логических диска, то последний подраздел (в системе Windows он именовался бы диском F:) получил бы номер 7.

Системный загрузчик Windows NT/2000/XP

Операционные системы класса Windows NT имеют возможность загружать не одну операционную систему, а несколько, то есть системный загрузчик Windows NT/2000/XP является менеджером загрузки. Для указания установленных операционных систем и выбора одной из них используется файл boot.ini. Этот файл является текстовым. Он обрабатывается программой ntldr, которая, собственно, и является системным загрузчиком и на которую передается управление из внесистемного загрузчика.

Файл boot.ini состоит из двух секций. Пример такого файла приведен в листинге 5.1.

Листинг 5.1. Файл boot.ini

```
[boot loader]
timeout=10
default=multi(0)disk(0)rdisk(0)partition(2)\WINNT
[operating systems]
multi(0)disk(0)rdisk(0)partition(2)\WINNT="IT.MTC.EDU Microsoft Windows 2000 Server RUS"
/fastdetect
multi(0)disk(0)rdisk(1)partition(2)\WIN2KP="Staff.MTC.EDU Microsoft Windows 2000
Professional RUS" /fastdetect
multi(0)disk(0)rdisk(0)partition(4)\WIN2K_S="SQL server on M$ Windows 2000 Server RUS" /
fastdetect
multi(0)disk(0)rdisk(2)partition(2)\WIN2K.PRO="Microsoft Windows 2000 Professional RUS"
/fastdetect
C:\="Microsoft Windows 98"
C:\CMDCONS\BOOTSECT.DAT="Recovery Console Microsoft Windows 2000" /cmdcons
```

В первой секции этого файла, названной [boot loader], строка timeout задает время в секундах, по истечении которого будет загружаться операционная система, указанная в строке default этой секции. Как мы видим, для выбора одной из операционных систем пользователю дается 10 с. Если бы значение timeout равнялось нулю или во второй секции была бы прописана только одна операционная система, то у пользователя не было бы выбора. В этом случае будет загружаться система,

указанная в строке default. Если же значение timeout равняется –1, то загрузка не будет происходить до тех пор, пока пользователь явно не выберет в меню одну из операционных систем и не нажмет клавишу Enter.

Инструкция default указывает, где (на каком накопителе и в каком разделе этого накопителя) располагается операционная система, загружаемая по умолчанию. В большинстве случаев мы можем увидеть там примерно такую строку:

```
default=multi(0)disk(0)rdisk(0)partition(2)\WINNT
```

Слово multi в этой строке означает, что при работе программы ntldr должны использоваться драйверы из BIOS компьютера (используется прерывание int13h). Номер в скобках должен быть равен 0.

Слово disk на персональных компьютерах с подключением накопителей на магнитных дисках через IDE-интерфейс фактически не несет никакой информации, однако оно должно быть записано, а в скобках должен стоять ноль. В случае SCSI-дисков это слово задает идентификатор SCSI ID диска.

Слово rdisk определяет порядковый номер накопителя. Всего при использовании IDE-интерфейса может быть до 4 накопителей на жестких дисках; они нумеруются от 0 до 3.

Наконец, слово partition определяет номер раздела, на который установлена операционная система. После указания раздела записывается имя каталога, в котором расположены файлы этой операционной системы.

Во второй секции, обозначенной как [operating systems], построчно перечисляются пути к установленным операционным системам с текстовыми полями, заключенными в кавычки. Именно тот текст мы и видим при работе загрузчика ntldr, когда он выводит меню с операционными системами. Если на компьютере установлены помимо систем Windows NT/2000/XP еще какие-нибудь операционные системы (например, DOS, Windows 9x, Linux и т. д.), то их можно будет также загрузить. Для этого в секции необходимо указать полный путь к файлу, в котором должен содержаться соответствующий системный загрузчик (загрузочный сектор). Этот файл обязательно должен располагаться на том же диске C:, иначе программа ntldr не сможет его найти. Следует отметить, что для MS DOS и Windows 9x можно не указывать имя файла с загрузочным сектором, а указать только сам корневой каталог диска C:. Но это возможно только в том случае, если имя файла, содержащего системный загрузчик, будет стандартным — bootsect.dos.

Кэширование операций ввода-вывода при работе с накопителями на магнитных дисках

Как известно, накопители на магнитных дисках обладают крайне низким быстродействием по сравнению с процессорами и оперативной памятью. Разница составляет несколько порядков. Например, современные процессоры за один такт работы, а они работают уже с частотами в несколько гигагерц, могут выполнять по две операции, и, таким образом, время выполнения операции (с позиции внешнего на-

блюдателя, который не видит конвейеризации при выполнении машинных команд, позволяющей увеличить производительность в несколько раз) может составлять менее 0,5 нс (!). В то же время переход магнитной головки с дорожки на дорожку занимает несколько миллисекунд; подобная же задержка требуется и на поиск нужного сектора данных. Как известно, в современных приводах средняя длительность на чтение случайным образом выбранного сектора данных составляет около 20 мс, что существенно медленнее, чем выборка команды или операнда из оперативной памяти и уж тем более из кэш-памяти. Правда, после этого данные читаются большим пакетом (сектор, как мы уже говорили, имеет размер 512 байт, а при операциях с диском часто читаются или записываются сразу несколько секторов). Таким образом, средняя скорость работы процессора с оперативной памятью на 2–3 порядка выше, чем средняя скорость передачи данных из внешней памяти на магнитных дисках в оперативную память.

Для того чтобы сгладить такое сильное несоответствие в производительности основных подсистем, используется буферизация и/или *кэширование* данных в дисковом кэше (disk cache). Простейшим вариантом ускорения дисковых операций чтения данных можно считать использование двойной буферизации. Ее суть заключается в том, что пока в один буфер заносятся данные с магнитного диска, из второго буфера ранее считанные данные могут быть прочитаны и переданы в запросившую их задачу. Аналогично и при записи данных. Буферизация используется во всех операционных системах, но помимо буферизации применяется и кэширование. Кэширование исключительно полезно в том случае, когда программа неоднократно читает с диска одни и те же данные. После того как они один раз будут помещены в кэш, обращений к диску больше не потребуется, и скорость работы программы значительно возрастет.

Упрощая, можно сказать, что под дисковым кэшем можно понимать некий пул буферов, которыми мы управляем с помощью соответствующего системного процесса. Если считывается какое-то множество секторов, содержащих записи того или иного файла, то эти данные, пройдя через кэш, там остаются (до тех пор, пока другие секторы не заменят эти буферы). Если впоследствии потребуется повторное чтение, то данные могут быть извлечены непосредственно из оперативной памяти без фактического обращения к диску. Ускорить можно и операции записи: данные помещаются в кэш, и для запросившей эту операцию задачи получается, что фактически они уже записаны. Задача может продолжить свое выполнение, а системные внешние процессы через некоторое время запишут данные на диск. Это называется *отложенной записью* (lazy write[1]). Если режим отложенной записи отключен, только одна задача может записывать на диск свои данные. Остальные приложения должны ждать своей очереди. Это ожидание подвергает информацию риску не меньшему (если не большему), чем сама отложенная запись, которая к тому же и более эффективна по скорости работы с диском.

Интервал времени, после которого данные будут фактически записываться, с одной стороны, желательно выбрать большим, поскольку это позволило бы не читать (если потребуется) эти данные заново, так как они уже и так фактически на-

[1] Дословно — «ленивая» запись.

ходятся в кэше. И после их модификации эти данные опять же помещаются в быстродействующий кэш. С другой стороны, для большей надежности, желательно поскорее отправить данные во внешнюю память, поскольку она энергонезависима, и в случае какой-нибудь аварии (например, нарушения питания) данные в оперативной памяти пропадут, в то время как на магнитном диске они с большой вероятностью останутся в безопасности.

Поскольку количество буферов, составляющих кэш, ограничено, может возникнуть ситуация, когда считываемые или записываемые данные потребуют замены данных в этих буферах. При этом возможны различные дисциплины выделения буферов под вновь затребованную операцию кэширования.

Кэширование дисковых операций может быть существенно улучшено за счет *упреждающего чтения* (read ahead), которое основано на чтении с диска гораздо большего количества информации, чем на самом деле запрошено приложением или операционной системой. Когда некоторой программе требуется считать с диска только один сектор, программа кэширования читает несколько дополнительных блоков данных. При этом, как известно, операции последовательного чтения нескольких секторов фактически несущественно замедляют операцию чтения затребованного сектора с данными. Поэтому, если программа вновь обратится к диску, вероятность того, что нужные ей данные уже находятся в кэше, будет достаточно высока. Поскольку передача данных из одной области памяти в другую происходит во много раз быстрее, чем чтение их с диска, кэширование существенно сокращает время выполнения операций с файлами.

Итак, путь информации от диска к прикладной программе пролегает как через буфер, так и через дисковый кэш. Когда приложение запрашивает с диска данные, программа кэширования перехватывает этот запрос и читает вместе с необходимыми секторами еще и несколько дополнительных. Затем она помещает в буфер требующуюся задаче информацию и ставит об этом в известность операционную систему. Операционная система сообщает задаче, что ее запрос выполнен, и данные с диска находятся в буфере. При следующем обращении приложения к диску программа кэширования прежде всего проверяет, не находятся ли уже в памяти затребованные данные. Если это так, то она копирует их в буфер, если же их в кэше нет, то запрос на чтение диска передается операционной системе. Когда задача изменяет данные в буфере, они копируются в кэш.

Важно заметить, что простое увеличение объема памяти, отводимого под кэширование файлов, может и не привести к росту быстродействия системы. Другими словами, наблюдается далеко не прямо пропорциональная зависимость ускорения операций с файлами от размера кэша. Кривая этой зависимости достаточно скоро перестает расти, а затем и вовсе эффективность кэширования начинает снижаться. Объяснение этому заключается в том, что поиск нужного фрагмента данных в буферах кэша осуществляется путем их полного перебора. Поэтому с ростом числа буферов кэша затраты на их перебор становятся значительными. И поскольку невозможно обеспечить 100-процентного кэш-попадания искомых данных, то естественно наступает момент, когда среднее время доступа к данным перестает снижаться с увеличением кэша. Очевидно, что оптимальный размер дискового кэша

зависит от очень многих факторов, в том числе и от частоты повторных обращений к недавно прочитанным данным, и от среднего объема обрабатываемых файлов, и от разницы в быстродействии центральной части компьютера и дисковой подсистемы.

В ряде операционных систем имеется возможность указать в явном виде параметры кэширования, в то время как в других за эти параметры отвечает сама операционная система.

Так, в системах семейства Windows 9x мы можем указать и объем памяти, отводимый для кэширования, и объем порции (chunk[1]) данных, из которых набирается кэш, и предельное количество имен файлов, и параметры кэширования каталогов. В файле SYSTEM.INI, расположенном в основном каталоге такой операционной системы (обычно это каталог Windows), в секции [vcache] есть возможность прописать, например, следующие значения:

```
[vcache]
MinFileCache=4096
MaxFileCache=32768
ChunkSize=512
```

Здесь указано, что минимально под кэширование данных зарезервировано 4 Мбайт оперативной памяти, максимальный объем кэша может достигать 32 Мбайт, а размер данных, которыми манипулирует менеджер кэша, равен одному сектору. Следует заметить, что поскольку в современных компьютерах нередко устанавливается большой объем оперативной памяти, порой существенно превосходящий 256 Мбайт, то для обеспечения корректной работы подсистемы кэширования обязательно нужно указывать в явном виде значение MaxFileCache. Оно ни в коем случае не должно превышать величину 262 144 Кбайт. Это ограничение следует из-за особенностей программной реализации подсистемы кэширования[2] — при превышении этого значения происходят нарушения в работе подсистемы памяти и вычислительные процессы могут быть разрушены.

Во всех операционных системах от Microsoft принята стратегия активного кэширования файлов, при которой для кэширования отводится вся свободная память. Поэтому без явного ограничения объема памяти, отводимой под кэширование файлов, мы можем столкнуться с ситуацией, когда рост дискового кэша приводит к значительному росту числа страниц памяти, «сброшенных» в файл подкачки. Последнее может привести к заметному замедлению работы системы, несмотря на то что кэширование имеет целью именно ускорение в работе дисковой подсистемы.

В операционных системах Windows NT 4.0, Windows 2000 и Windows XP также имеется возможность управлять некоторыми параметрами кэширования. Правда, сделать это можно только путем редактирования реестра.

Например, если в разделе [HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management] реестра найти параметр IOPageLockLimit и присвоить ему значение 163777216, то это будет означать, что 16 384 Кбайт будут отведе-

[1] Дословно — «кусочек».

[2] Драйвер VCACHE разрабатывался в то время, когда объем памяти в 256 Мбайт казался недостижимым.

ны в физической памяти для хранения буферов дискового кэша. Эта память не может быть выгружена в файл подкачки. Дело в том, что, к большому сожалению, разработчики из Microsoft приняли решение, согласно которому кэшируемые файлы отображаются на виртуальное адресное пространство, а не на физическую память компьютера, как это сделано в других операционных системах. Это означает, что некоторые страничные кадры этого виртуального адресного пространства могут быть отображены не на реальную оперативную память компьютера, а размещены во внешней памяти (попасть в страничный файл подкачки). Очевидно, что это может сильно замедлять работу рассматриваемой подсистемы. Поэтому блокирование некоторого числа страниц файлового кэша от перемещения их во внешнюю память должно приводить к повышению эффективности кэширования. В качестве рекомендации можно заметить, что упомянутое значение в 16 Мбайт можно выделять для компьютеров с объемом памяти более 128 Мбайт.

В других операционных системах можно указывать больше параметров, определяющих работу подсистемы кэширования (см., например, раздел «Файловая система HPFS» в главе 6).

Помимо описанных действий, связанных с кэшированием файлов, операционная система может оптимизировать перемещение головок чтения/записи данных, связанное с выполнением запросов от параллельно выполняющихся задач. Время, необходимое на получение данных с магнитного диска, складывается из времени перемещения магнитной головки на требуемый цилиндр и времени поиска заданного сектора; а временем считывания найденного сектора и временем передачи этих данных в оперативную память мы можем пренебречь. Таким образом, основные затраты времени уходят на поиск данных. В мультипрограммных операционных системах при выполнении многих задач запросы на чтение и запись данных могут идти таким потоком, что при их обслуживании образуется очередь. Если выполнять эти запросы в порядке поступления их в очередь, то вследствие случайного характера обращений к тому или иному сектору магнитного диска потери времени на поиск данных могут значительно возрасти. Напрашивается очевидное решение: поскольку переупорядочивание запросов с целью минимизации затрат времени на поиск данных можно выполнить очень быстро (практически этим временем можно пренебречь, учитывая разницу в быстродействии центральной части компьютера и устройств ввода-вывода), то необходимо найти метод, позволяющий выполнить такое переупорядочивание оптимальным образом. Изучение этой проблемы позволило найти наиболее эффективные дисциплины планирования.

Перечислим известные дисциплины, в соответствии с которыми можно перестраивать очередь запросов на операции чтения/записи данных [11].

- *SSTF* (Shortest Seek Time First — запрос с наименьшим временем позиционирования выполняется первым). В соответствии с этой дисциплиной при позиционировании магнитных головок следующим выбирается запрос, для которого необходимо минимальное перемещение с цилиндра на цилиндр, даже если этот запрос не был первым в очереди на ввод-вывод. Однако для этой дисциплины характерна сильная дискриминация некоторых запросов, а ведь они могут идти от высокоприоритетных задач. Обращения к диску проявляют тен-

денцию концентрироваться, в результате чего запросы на обращение к самым внешним и самым внутренним дорожкам могут обслуживаться существенно дольше, и нет никакой гарантии обслуживания. Достоинством такой дисциплины является максимально возможная пропускная способность дисковой подсистемы.

- *Scan* (сканирование). При сканировании головки поочередно перемещаются то в одном, то в другом «привилегированном» направлении, обслуживая «по пути» подходящие запросы. Если при перемещении головок чтения/записи более нет попутных запросов, то движение начинается в обратном направлении.
- *Next-Step Scan* (отложенное сканирование). Отличается от предыдущей дисциплины тем, что на каждом проходе обслуживаются только те запросы, которые уже существовали на момент начала прохода. Новые запросы, появляющиеся в процессе перемещения головок чтения/записи, формируют новую очередь запросов, причем таким образом, чтобы их можно было оптимально обслужить на обратном ходу.
- *C-Scan* (циклическое сканирование). По этой дисциплине головки перемещаются циклически с самой наружной дорожки к внутренним, по пути обслуживая имеющиеся запросы, после чего вновь переносятся к наружным цилиндрам. Эту дисциплину иногда реализуют таким образом, чтобы запросы, поступающие во время текущего прямого хода головок, обслуживались не попутно, а при следующем проходе, что позволяет исключить дискриминацию запросов к самым крайним цилиндрам. Эта дисциплина характеризуется очень малой дисперсией времени ожидания обслуживания [11]. Ее часто называют «элеваторной».

Контрольные вопросы и задачи

Вопросы для проверки

1. Почему создание подсистемы ввода-вывода считается одной из самых сложных областей проектирования операционных систем?
2. Почему операции ввода-вывода в операционных системах объявляются привилегированными?
3. Перечислите основные задачи, возлагаемые на супервизор ввода-вывода?
4. В каких случаях устройство ввода-вывода называется инициативным?
5. Какие режимы управления вводом-выводом вы знаете? Опишите каждый из них.
6. Что означает термин «spooling» и что означает термин «swapping»?
7. Чем обеспечивается независимость пользовательских программ от устройств ввода-вывода, подключенных к компьютеру?
8. Что такое синхронный и асинхронный ввод-вывод?
9. Опишите структуру магнитного диска (разбиение дисков на разделы). Сколько (и каких) разделов может быть на магнитном диске?

10. Как в общем случае осуществляется загрузка операционной системы после включения компьютера? Что такое начальный, системный и внесистемный загрузчики? Где они располагаются?
11. Расскажите о кэшировании операций ввода-вывода при работе с накопителями на магнитных дисках.

Задания

Используя специально выделенный для этих целей компьютер, изучите структуру диска и освойте работу с программой Disk Editor.

1. Включите компьютер. Во время выполнения программы самотестирования войдите в BIOS и установите возможность загрузки с дискеты.
2. Загрузите операционную систему, расположенную на магнитном диске.
3. Загрузитесь с системной дискеты (на ней должна быть система MS DOS с необходимыми утилитами).
4. Запустите программу Disk Editor из комплекта утилит Питера Нортона. С помощью встроенной справочной системы изучите основные возможности этой утилиты.
5. Посмотрите структуру диска. Сохраните MBR и загрузочный сектор на своей дискете.
6. Найдите таблицы размещения файлов для указанного раздела магнитного диска. Сохраните их на дискете. Выйдите из программы Disk Editor.
7. Запустите программу FDisk. Изучите структуру диска, посмотрите, сколько и каких разделов на нем расположено?
8. Удалите все логические диски с помощью программы FDisk.
9. Перезапустите компьютер и убедитесь, что операционная система, расположенная раньше на магнитном диске, больше не функционирует.
10. Используя программу Disk Editor, восстановите операционную систему, а также файлы, расположенные на магнитном диске и созданные ранее с помощью программы Disk Editor.

Глава 6. Файловые системы

Система управления файлами является основной в абсолютном большинстве современных операционных систем. Например, операционные системы UNIX никак не могут функционировать без файловой системы, ибо понятие файла для них является одним из самых фундаментальных. Все современные операционные системы используют файлы и соответствующее программное обеспечение для работы с ними. Дело в том что, во-первых, через файловую систему связываются по данным многие системные обрабатывающие программы. Во-вторых, с помощью этой системы решаются проблемы централизованного распределения дискового пространства и управления данными. Наконец, пользователи получают более простые способы доступа к своим данным, которые они размещают на устройствах внешней памяти.

Существует большое количество файловых систем, созданных для разных устройств внешней памяти и разных операционных систем. В них используются, соответственно, разные принципы размещения данных на носителе. В данной главе мы ограничимся рассмотрением наиболее распространенных файловых систем, с которыми мы сталкиваемся при работе на персональных компьютерах. Это системы FAT, FAT32 и NTFS. Знание основных принципов их построения необходимо не только специалисту в области вычислительной техники, но и обычному пользователю. Особенно актуальными становятся знания возможностей файловой системы NTFS, которая сегодня получает все большее распространение.

Функции файловой системы и иерархия данных

Напомним, что под *файлом* обычно понимают именованный набор данных, организованных в виде совокупности записей одинаковой структуры. Для управления этими данными создаются соответствующие *файловые системы*. Файловая система предоставляет возможность иметь дело с логическим уровнем структуры данных и операций, выполняемых над данными в процессе их обработки. Именно файловая система определяет способ организации данных на диске или на каком-

нибудь ином носителе. Специальное системное программное обеспечение, реализующее работу с файлами по принятым спецификациям файловой системы, часто называют *системой управления файлами*. Именно системы управления файлами отвечают за создание, уничтожение, организацию, чтение, запись, модификацию и перемещение файловой информации, а также за управление доступом к файлам и за управление ресурсами, которые используются файлами. Назначение системы управления файлами — предоставление более удобного доступа к данным, организованным как файлы, то есть вместо низкоуровневого доступа к данным с указанием конкретных физических адресов нужной нам записи используется логический доступ с указанием имени файла и записи в нем.

Благодаря системам управления файлами пользователям предоставляются следующие возможности:

- создание, удаление, переименование (и другие операции) именованных наборов данных (файлов) из своих программ или посредством специальных управляющих программ, реализующих функции интерфейса пользователя с его данными и активно использующих систему управления файлами;
- работа с недисковыми периферийными устройствами как с файлами;
- обмен данными между файлами, между устройствами, между файлом и устройством (и наоборот);
- работа с файлами путем обращений к программным модулям системы управления файлами (часть API ориентирована именно на работу с файлами);
- защита файлов от несанкционированного доступа.

Как правило, все современные операционные системы имеют соответствующие системы управления файлами. А некоторые операционные системы имеют возможность работать с несколькими файловыми системами (либо с одной из нескольких, либо сразу с несколькими одновременно). В этом случае говорят о *монтируемых файловых системах* (монтируемую систему управления файлами можно установить как дополнительную), и в этом смысле они самостоятельны.

Очевидно, что система управления файлами, будучи компонентом операционной системы, не является независимой от нее, поскольку активно использует соответствующие вызовы API. С другой стороны, системы управления файлами сами дополняют API новыми вызовами. Можно сказать, что основное назначение файловой системы и соответствующей ей системы управления файлами — предоставление удобного доступа к данным, организованным в виде файлов, то есть вместо низкоуровневого доступа к данным с указанием конкретных физических адресов нужной нам записи используется логический доступ с указанием имени файла и записи в нем.

Следует заметить, что любая система управления файлами не существует сама по себе — она разрабатывается для работы в конкретной операционной системе. В качестве примера можно сказать, что всем известная файловая система FAT (File Allocation Table — таблица размещения файлов) имеет множество реализаций как система управления файлами. Так, система, получившая это название и разработанная для первых персональных компьютеров, называлась просто FAT (нынче ее

называют FAT12[1]). Хотя ее разрабатывали для работы с дискетами, некоторое время она использовалась при работе с жесткими дисками. Потом ее доработали для работы с жесткими дисками большего объема, и новая реализация получила название FAT16. Это название файловой системы мы употребляем и по отношению к подсистеме управления файлами самой системы MS DOS, однако реализацию системы управления файлами для OS/2, которая использует основные принципы системы FAT, называют super-FAT; основное отличие — возможность поддерживать для каждого файла расширенные атрибуты. Есть версия системы управления файлами с принципами FAT и для Windows 95/98, есть реализация для Windows NT и т. д. Другими словами, для работы с файлами, организованными в соответствии с некоторой файловой системой, для каждой операционной системы должна быть разработана соответствующая система управления файлами. И эта система управления файлами будет работать только в той операционной системе, для которой создана, но при этом обеспечит доступ к файлам, созданным с помощью системы управления файлами другой операционной системы, но работающей по тем же основным принципам файловой системы.

В качестве примера снова можно привести всем известную файловую систему FAT, поддерживаемую абсолютным большинством операционных систем, работающих на современных персональных компьютерах. В MS DOS, OS/2, Windows 95/98/ME, Windows NT/2000/XP, Linux, FreeBSD и других можно работать с файлами, организованными по принципам FAT. Однако программные модули соответствующих систем управления файлами не взаимозаменяемы. Кроме того, все эти системы управления файлами имеют свои индивидуальные особенности и ограничения. Иногда только из контекста ясно, о чем идет речь — о принципах работы файловой системы или о ее конкретной реализации. Другими словами, для работы с файлами, организованными в соответствии с некоторой файловой системой, для каждой операционной системы должна быть разработана соответствующая система управления файлами; и эта система управления файлами будет работать только в той операционной системе, для которой она и создана. Таким образом, *файловая система* — это множество именованных наборов данных, организованное по принятым спецификациям, которые определяют способы получения адресной информации, необходимой для доступа к этим файлам.

Таким образом, термин *файловая система* определяет, прежде всего, принципы доступа к данным, организованным в файлы. Тот же термин используют и по отношению к конкретным файлам, расположенным на том или ином носителе данных. А термин *система управления файлами* следует употреблять по отношению к конкретной реализации файловой системы, то есть это — комплекс программных модулей, обеспечивающих работу с файлами в конкретной операционной системе.

Информация, с которой работает человек, обычно структурирована. Это, прежде всего, позволяет более эффективно организовать хранение данных, облегчает их поиск, предоставляет дополнительные возможности в именовании. Аналогично,

[1] Число 12 в имени этой файловой системы означает, что для указания адреса данных, составляющих файл, используется 12 двоичных разрядов.

и при работе с файлами желательно ввести механизмы структурирования. Проще всего организовать иерархические отношения. Для этого достаточно ввести понятие *каталога* (directory). Каталог содержит информацию о данных, организованных в виде файлов. Другими словами, в каталоге должны содержаться *дескрипторы файлов*. Если файлы организованы на блочном устройстве, то именно с помощью каталога система управления файлами будет находить адреса тех блоков, в которых размещены искомые данные. Причем очевидно, что каталогом может быть не только специальная системная информационная структура, которую часто называют *корневым каталогом*, но и сам файл. Такой *файл-каталог* должен иметь специальное системное значение; система управления файлами должна его выделять на фоне обычных файлов. Файл-каталог часто называют *подкаталогом* (subdirectory). Если файл-каталог содержит информацию о других файлах, то поскольку среди них также могут быть файлы-каталоги, мы получаем возможность строить почти ничем не ограниченную иерархию.

Более того, введение таких файловых объектов, как файлы-каталоги, позволяет не только структурировать файловую систему, но и решить проблему ограниченного количества элементов в корневом каталоге. Ограничений на количество элементов в файле-каталоге нет, поэтому можно создавать каталоги чрезвычайно большого размера.

Файловая система FAT

Файловая система FAT (File Allocation Table — таблица размещения файлов) получила свое название благодаря простой таблице, в которой указываются:

- непосредственно адресуемые участки логического диска, отведенные для размещения в них файлов или их фрагментов;
- свободные области дискового пространства;
- дефектные области диска (эти области содержат дефектные участки и не гарантируют чтение и запись данных без ошибок).

В файловой системе FAT дисковое пространство любого логического диска делится на две области (рис. 6.1): *системную область* и *область данных*.

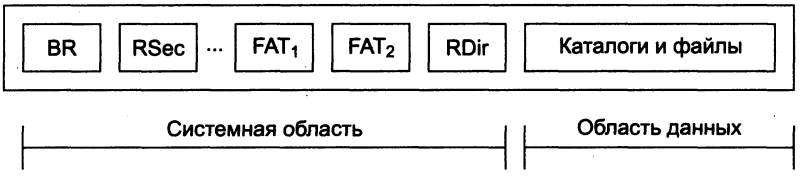


Рис. 6.1. Структура логического диска в FAT

Системная область логического диска создается и инициализируется при форматировании, а в последующем обновляется при работе с файловой структурой. Область данных логического диска содержит обычные файлы и файлы-каталоги; эти объекты образуют иерархию, подчиненную корневому каталогу. Элемент катало-

га описывает файловый объект, который может быть либо обычным файлом, либо файлом-каталогом. Область данных, в отличие от системной области, доступна через пользовательский интерфейс операционной системы. Системная область состоит из следующих компонентов (расположенных в логическом адресном пространстве друг за другом):

- загрузочной записи (Boot Record, BR);
- зарезервированных секторов (Reserved Sectors, ResSec);
- таблицы размещения файлов (File Allocation Table, FAT);
- корневого каталога (Root Directory, RDir).

Таблица размещения файлов

Таблица размещения файлов является очень важной информационной структурой. Можно сказать, что она представляет собой адресную карту области данных, в которой описывается и состояние каждого участка области данных, и принадлежность его к тому или иному файловому объекту.

Всю область данных разбивают на так называемые *кластеры*. Кластер представляет собой один или несколько смежных секторов в логическом дисковом адресном пространстве (точнее — только в области данных). Кластер — это минимальная адресуемая единица дисковой памяти, выделяемая файлу (или некорневому каталогу). Кластеры введены для того, чтобы уменьшить количество адресуемых единиц в области данных логического диска.

Каждый файл занимает целое число кластеров. Последний кластер при этом может быть задействован не полностью, что при большом размере кластера может приводить к заметной потере дискового пространства. На дискетах кластер занимает один или два сектора, а на жестких дисках его размер зависит от объема раздела (табл. 6.1). В таблице FAT кластеры, принадлежащие одному файлу (или файлу-каталогу), связываются в цепочки. Для указания номера кластера в файловой системе FAT16 используется 16-разрядное слово, следовательно, можно иметь до $2^{16} = 65\,536$ кластеров (с номерами от 0 до 65 535).

Таблица 6.1. Соотношения между размером раздела и размером кластеров в FAT16

Емкость раздела, Мбайт	Количество секторов в кластере	Размер кластеров, Кбайт
16–127	4	2
128–255	8	4
256–511	16	8
512–1023	32	16
1024–2047	64	32

Заметим, что в Windows NT/2000/XP разделы файловой системы FAT могут иметь размер до 4097 Мбайт. В этом случае кластер будет объединять уже 128 секторов.

Номер кластера всегда относится к области данных диска (пространству, зарезервированному для файлов и подкаталогов). Номера кластеров соответствуют эле-

ментам таблицы размещения файлов. Первый допустимый номер кластера всегда начинается с 2.

Логическое разбиение области данных на кластеры как совокупности секторов взамен использования одиночных секторов имеет следующий смысл:

- прежде всего, уменьшается размер самой таблицы FAT;
- уменьшается возможная фрагментация файлов;
- ускоряется доступ к файлу, так как в несколько раз сокращается длина цепочек фрагментов дискового пространства, выделенных для него.

Однако слишком большой размер кластера ведет к неэффективному использованию области данных, особенно в случае большого количества маленьких файлов. Как мы только что заметили, в среднем на каждый файл теряется около половины кластера. Из табл. 6.1 следует, что при размере кластера в 32 сектора (объем раздела при этом — от 512 до 1023 Мбайт), то есть 16 Кбайт, средняя величина потерь на файл равняется 8 Кбайт, и при нескольких тысячах файлов[1] потери могут составлять более 100 Мбайт. Поэтому в современных файловых системах размеры кластеров ограничиваются (обычно от 512 байт до 4 Кбайт), либо предоставляется возможность выбирать размер кластера.

Достаточно наглядно идею файловой системы, использующей таблицу размещения файлов, иллюстрирует рис. 6.2.

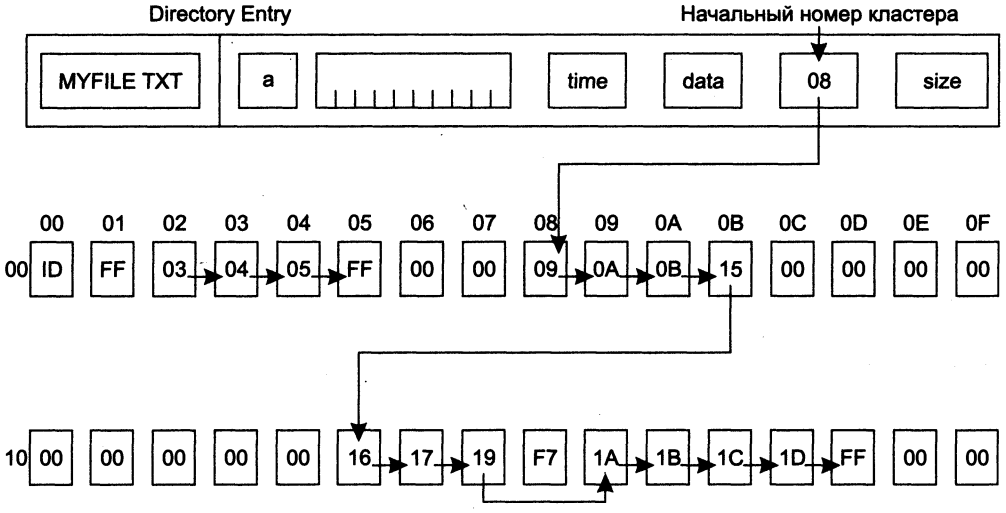


Рис. 6.2. Иллюстрация основной концепции FAT

Из рисунка видно, что файл MYFILE.TXT размещается, начиная с восьмого кластера. Всего файл MYFILE.TXT занимает 12 кластеров. Цепочка (chain) кластеров для нашего примера может быть записана следующим образом: 8, 9, 0A, 0B, 15, 16, 17, 19,

[1] Например, число 10 000–15 000 файлов (или даже более, особенно когда файлы небольшого размера) на логическом диске с объемом в 1000 Мбайт встречается достаточно часто.

1A, 1B, 1C, 1D. Кластер с номером 18 помечен специальным кодом F7 как плохой (bad), он не может быть использован для размещения данных. При форматировании обычно проверяется поверхность магнитного диска, и те сектора, при контрольном чтении с которых происходили ошибки, помечаются в FAT как плохие. Кластер 1D помечен кодом FF как конечный (последний в цепочке) кластер, принадлежащий данному файлу. Свободные (незанятые) кластеры помечаются кодом 00; при выделении нового кластера для записи файла берется первый свободный кластер. Возможные значения, которые могут приписываться элементам таблицы FAT, приведены в табл. 6.2.

Таблица 6.2. Значения элементов FAT

Значение	Описание
0000h	Свободный кластер
fff0h–fff6h	Зарезервированный кластер
fff7h	Плохой кластер
fff8h–ffffh	Последний кластер в цепочке
0002h–ffefh	Номер следующего кластера в цепочке

Поскольку файлы на диске изменяются (удаляются, перемещаются, увеличиваются или уменьшаются), то упомянутое правило выделения первого свободного кластера для новой порции данных приводит к *фрагментации* файлов, то есть данные одного файла могут располагаться не в смежных кластерах, а порой в очень удаленных друг от друга, образуя сложные цепочки. Естественно, что это приводит к существенному замедлению работы с файлами.

В связи с тем, что таблица FAT используется при доступе к диску очень интенсивно, она обычно загружается в оперативную память (в буферы ввода-вывода или в кэш) и остается там настолько долго, насколько это возможно. Если таблица большая, а файловый кэш, напротив, относительно небольшой, в памяти размещаются только фрагменты этой таблицы, к которым обращались в последнее время.

В связи с чрезвычайной важностью таблицы FAT она обычно хранится в двух идентичных экземплярах, второй из которых непосредственно следует за первым. Обновляются копии FAT одновременно, используется же только первый экземпляр. Если он по каким-либо причинам окажется разрушенным, то произойдет обращение ко второму экземпляру. Так, например, утилита проверки и восстановления файловой структуры ScanDisk из ОС Windows 9x при обнаружении несоответствия первичной и резервной копии FAT предлагает восстановить главную таблицу, используя данные из копии.

Корневой каталог отличается от обычного файла-каталога тем, что он помимо размещения в фиксированном месте логического диска имеет еще и фиксированное число элементов. Для каждого файла и каталога в файловой системе хранится информация в соответствии со структурой, представленной в табл. 6.3.

Для работы с данными на магнитных дисках в системах DOS, которые имеют файловую систему FAT, удобно использовать широко известную утилиту Disk Editor из

комплекта утилит Питера Нортона. У нее много достоинств. Прежде всего, она компактна, легко размещается на системной дискете с MS DOS, снабжена встроенной системой подсказок и необходимой справочной информацией. Используя ее, можно сохранять, модифицировать и восстанавливать загрузочную запись, восстанавливать таблицу FAT в случае ее повреждения, а также выполнять много других операций. Основными недостатками этой программы на сегодняшний день являются ограничения на размеры диска и разделов и отсутствие поддержки работы с такими распространенными файловыми системами, как FAT32 и NTFS. Вместо нее теперь часто используют утилиту Partition Magic, однако наилучшей альтернативой этой программе на сегодняшний день можно считать утилиту Администратор дисков от Acronis.

Таблица 6.3. Структура элемента каталога

Размер поля данных, байт	Содержание поля
11	Имя файла или каталога
1	Атрибуты файла
1	Резервное поле
3	Время создания
2	Дата создания
2	Дата последнего доступа
2	Зарезервировано
2	Время последней модификации
2	Дата последней модификации
2	Номер начального кластера в FAT
4	Размер файла

Структура загрузочной записи DOS

Сектор, содержащий системный загрузчик DOS, является самым первым на логическом диске C:. Напомним, что на дискете системный загрузчик размещается в самом первом секторе; его физический адрес равен 0-0-1. Загрузочная запись состоит, как мы уже знаем, из двух частей: *блока параметров диска* (Disk Parameter Block, DPB) и *системного загрузчика* (System Bootstrap, SB). Блок параметров диска служит для идентификации физического и логического форматов логического диска, а системный загрузчик играет существенную роль в процессе загрузки DOS. Эта информационная структура приведена в табл. 6.4.

Первые два байта загрузочной записи занимает команда безусловного перехода (`JMP`) на программу SB. Третий байт содержит код 90H (`NOP` — нет операции). Далее располагается восьмибайтовый системный идентификатор, включающий информацию о фирме-разработчике и версии операционной системы. Затем следует блок параметров диска, а после него — системный загрузчик.

Для работы с загрузочной записью DOS, как и с другими служебными информационными структурами, удобно использовать уже упомянутую программу Disk

Editor из комплекта утилит Питера Нортона. Используя ее, можно сохранять, модифицировать и восстанавливать загрузочную запись, а также выполнять много других операций. Достаточно подробно работа с этой утилитой описана в [2].

Таблица 6.4. Структура загрузочной записи для FAT16

Смещение поля, байт	Длина поля, байт	Обозначение поля	Содержимое поля
00H (0)	3	JUMP 3EH	Безусловный переход на начало системного загрузчика
03H (3)	8		Системный идентификатор
0BH (11)	2	SectSize	Размер сектора, байт
0DH (13)	1	ClastSize	Число секторов в кластере
0EH (14)	2	ResSecs	Число зарезервированных секторов
10H (16)	1	FATcnt	Число копий FAT
11H (17)	2	RootSize	Максимальное число элементов Rdir
13H (19)	2	TotSecs	Число секторов на логическом диске, если его размер не превышает 32 Мбайт; иначе 0000H
15H (21)	1	Media	Дескриптор носителя
16H (22)	2	FATsize	Размер FAT, секторов
18H (24)	2	TrkSecs	Число секторов на дорожке
1AH (26)	2	HeadCnt	Число рабочих поверхностей
1CH (28)	4	HidnSecs	Число скрытых секторов
20H (32)	4		Число секторов на логическом диске, если его размер превышает 32 Мбайт
24H (36)	1		Тип логического диска (00H — гибкий, 80H — жесткий)
25H (37)	1		Зарезервировано
26H (38)	1		Маркер с кодом 29H
27H (39)	4		Серийный номер тома[1]
2BH (43)	11		Метка тома
36H (54)	8		Имя файловой системы
3EH (62)			Системный загрузчик
1FEH (510)	2		Сигнатура (слово AA55H)

[1] Том (volume) представляет собой единое логическое адресное пространство. Томом может быть обычный логический диск либо несколько дисковых пространств.

Файловые системы VFAT и FAT32

Одной из важнейших характеристик исходной файловой системы FAT было использование имен файлов формата 8.3. К стандартной системе FAT (имеется в виду прежде всего реализация FAT16) добавились еще две разновидности, использу-

мые в широко распространенных ОС от Microsoft (конкретно — в Windows 95 и Windows NT): VFAT (виртуальная система FAT) и система FAT32, используемая в одной из редакций ОС Windows 95 и Windows 98. Ныне файловая система FAT32 поддерживается и такими последними системами, как Windows Millennium Edition, Windows 2000 и Windows XP. Имеются реализации FAT32 и для Windows NT, и для Linux.

Файловая система VFAT впервые появилась в Windows 3.11 (Windows for Workgroups). С выходом Windows 95 в VFAT добавилась поддержка длинных имен файлов (Long File Name, LFN). Тем не менее, VFAT сохраняет совместимость с исходным вариантом FAT; это означает, что наряду с длинными именами в ней поддерживаются имена формата 8.3, а также существует специальный механизм для преобразования имен 8.3 в длинные имена, и наоборот. Именно файловая система VFAT поддерживается исходными версиями Windows 95, Windows NT 4, Windows 2000 и Windows XP. При работе с VFAT крайне важно использовать файловые утилиты, обслуживающие VFAT вообще и длинные имена в частности. Дело в том, что более ранние файловые утилиты DOS запросто модифицируют то, что кажется им исходной структурой FAT. Это может привести к потере или порче длинных имен из таблицы размещения файлов, поддерживаемой VFAT (или FAT32). Следовательно, для томов VFAT необходимо пользоваться файловыми утилитами, которые понимают и сохраняют файловую структуру VFAT.

Основными недостатками файловых систем FAT и VFAT, которые привели к разработке новой реализации файловой системы, основанной на той же идее (таблице размещения файлов), являются большие потери на кластеризацию при больших размерах логического диска и ограничения на сам размер логического диска. Поэтому в Microsoft Windows 95 OEM Service Release 2[1] на смену системе VFAT пришла файловая система FAT32, которая является полностью самостоятельной 32-разрядной файловой системой и содержит многочисленные усовершенствования и дополнения по сравнению с предыдущими реализациями FAT. Самое принципиальное отличие заключается в том, что FAT32 намного эффективнее расходует дисковое пространство. Прежде всего, кластеры в этой системе меньше, чем кластеры в предыдущих версиях, в которых могло быть не более 65 535 кластеров на логический диск (соответственно с увеличением размера диска приходилось увеличивать и размер кластеров). Следовательно, даже для дисков размером до 8 Гбайт FAT32 может использовать 4-килобайтные кластеры. В результате по сравнению с дисками FAT16 экономится значительное дисковое пространство (в среднем 10–15 %). В FAT32 проблема решается за счет того, что собственно сама таблица размещения файлов в этой файловой системе может содержать до 2^{28} кластеров[2].

FAT32 также может перемещать корневой каталог и использовать резервную копию FAT вместо стандартной. Расширенная загрузочная запись FAT32 позволяет

[1] Эту версию Windows 95 часто называют Windows 95 OSR2.

[2] В 32-разрядном слове FAT32, используемом для представления номера кластера, фактически учитываются только 28 разрядов, что приводит к тому, что размер таблицы размещения файлов в этой системе не может превышать 2^{28} элементов.

создавать копии критически важных структур данных; это повышает устойчивость дисков FAT32 к нарушениям структуры таблицы размещения файлов по сравнению с предыдущими версиями. Корневой каталог в FAT32 представлен в виде обычной цепочки кластеров, следовательно, он может находиться в произвольном месте диска, что снимает действовавшее ранее ограничение на размер корневого каталога (512 элементов).

Системы Windows 95 OSR2 и Windows 98 могут работать и с разделами VFAT, созданными Windows NT. То, что говорилось ранее об использовании файловых утилит VFAT с томами VFAT, относится и к FAT32. Поскольку прежние утилиты FAT (для FAT32 в эту категорию входят обе файловые системы, FAT и VFAT) могут повредить или уничтожить важную служебную информацию, для томов FAT32 нельзя пользоваться никакими файловыми утилитами, кроме утилит FAT32.

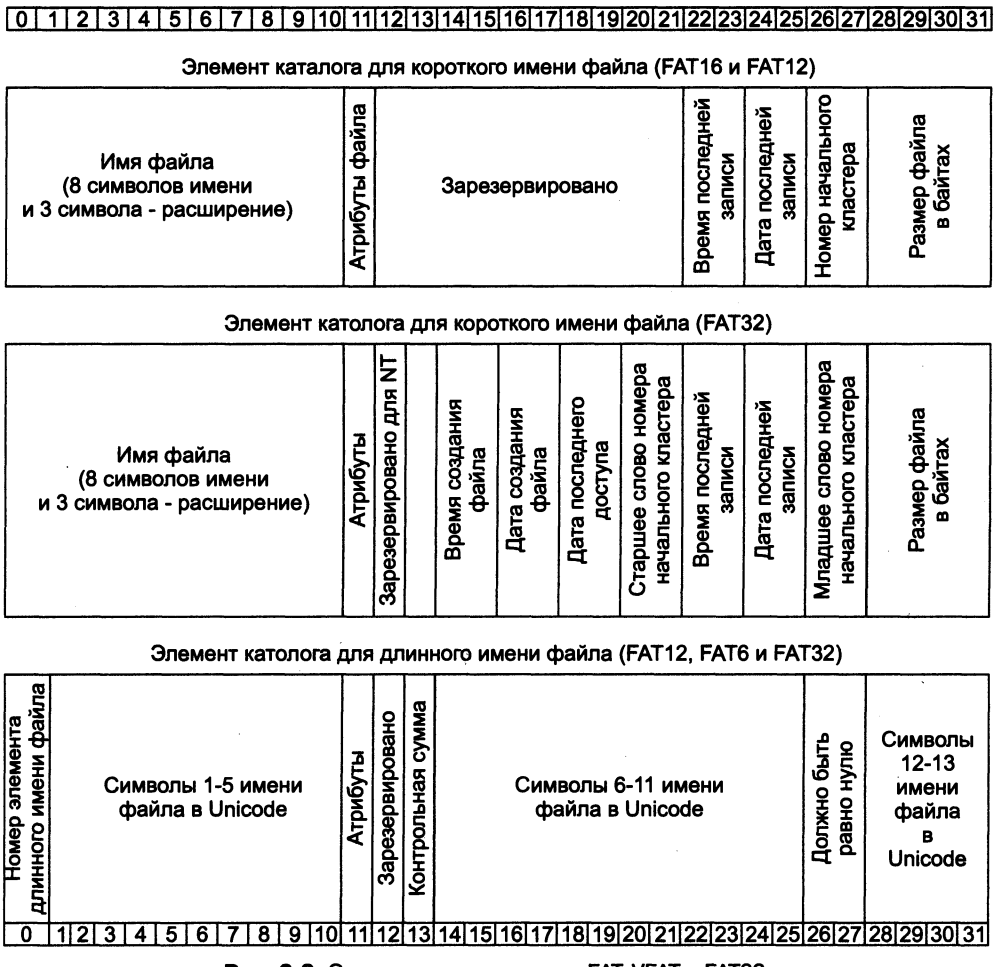


Рис. 6.3. Элементы каталогов для FAT, VFAT и FAT32

Помимо повышения максимального объема логического диска и уменьшения эффекта кластеризации, файловая система FAT32 вносит ряд необходимых усовершенствований в структуру корневого каталога. Предыдущие реализации требовали, чтобы вся информация корневого каталога FAT находилась в одном дисковом кластере. При этом корневой каталог мог содержать не более 512 файлов. Необходимость представлять длинные имена и обеспечить совместимость с прежними версиями FAT привела разработчиков компании Microsoft к компромиссному решению: для представления длинного имени они стали использовать элементы каталога, в том числе и корневого. По этой причине для того, чтобы компенсировать сокращение элементов главного каталога при использовании длинных имен, в FAT32 было увеличено их количество с 512 до 2048. Более того, чтобы не испытывать возможных проблем из-за расходования элементов активного каталога на описания файлов с длинными именами, компания Microsoft не рекомендует давать файлам слишком длинные имена.

Рассмотрим способ представления в VFAT длинного имени файла (рис. 6.3).

Первые 11 байт элемента каталога DOS используются для хранения имени файла. Каждое такое имя разделяется на две части: в первых восьми байтах хранятся символы собственно имени, а в последних трех — символы так называемого расширения, с помощью которого реализуются механизмы предопределенных типов. Были введены соответствующие системные соглашения, и файлы определенного типа желательно именовать с оговоренным расширением. Например, исполняемые файлы с расширением COM определяют исполняемую двоичную программу с простейшей односегментной структурой[1]. Более сложные программы имеют расширение EXE. Определены расширения для большого количества типов файлов и эти расширения используются для ассоциированного запуска программ, обрабатывающих эти файлы.

Если имя файла состоит менее чем из восьми символов, то в элементе каталога оно дополняется символами пробела, чтобы полностью заполнить все восемь байтов соответствующего поля. Аналогично и расширение может содержать от нуля до трех символов. Остальные (незаполненные) позиции в элементе каталога, определяющие расширение имени файла, заполняются символами пробела. Поскольку при работе с именем файла учитываются все одиннадцать свободных мест, то необходимость в отображении точки, которая обычно вводится между именем файла и его расширением, отпадает. В элементе каталога она просто подразумевается.

В двенадцатом байте элемента каталога хранятся *атрибуты файла*. Шесть из восьми указанных разрядов используются DOS[2]. Они перечислены ниже.

- A (Archive — архив). Показывает, что файл был открыт программой таким образом, чтобы у нее была возможность изменить содержимое этого файла. DOS устанавливает этот разряд при открытии файла. Программы резервного копирования (или, как часто говорят, архивирования, то есть составления архивов

[1] Для программных модулей, имеющих такую структуру, может использоваться и расширение BIN.

[2] В некоторых операционных системах, в частности в Novell Netware, используется один или два дополнительных разряда атрибутов.

данных) нередко сбрасывают его в ходе резервного копирования файла. Если применяется подобная методика, то в следующую создаваемую по порядку резервную копию будут добавлены только те файлы, в которых данный разряд установлен.

- D (Directory — каталог). Показывает, что данный элемент каталога указывает на подкаталог, а не на файл.
- V (Volume — том). Применяется только к одному элементу каталога в корневом каталоге. В нем собственно и хранится имя дискового тома. Этот атрибут также применяется в случае длинных имен файлов, о чем можно будет узнать из следующего раздела.
- S (System — системный). Показывает, что файл является частью операционной системы или специально отмечен подобным образом прикладной программой, что иногда делается для защиты от копирования.
- H (Hidden — скрытый). К скрытым относятся также файлы с установленным атрибутом S (системный), которые не отображаются по команде `DIR`.
- R (Read only — только для чтения). Показывает, что данный файл не подлежит изменению. Разумеется, поскольку это лишь разряд байта, хранящегося на диске, то любая программа может изменить этот разряд и, значит, разрешить изменение соответствующего файла. Этот атрибут в основном используется для примитивной защиты от пользовательских ошибок, то есть он помогает избежать неумышленного удаления или изменения ключевых файлов.

Следует отметить, что файл, помеченный одним или более из указанных выше атрибутов, может иметь вполне определенный смысл. Например, большинство файлов, отмечаемых в качестве системных, отмечаются также атрибутами «скрытый» и «только для чтения».

На дисках FAT12 или FAT16 следующие за именем 10 байт не используются. Обыкновенно они заполняются нулями и считаются резервными значениями. А на диске с файловой системой FAT32 эти 10 байт содержат самую разную информацию о файле. При этом байт, отмеченный как зарезервированный для NT, представляет собой, как подразумевает его название, поле, не используемое в DOS или Windows 9x, но применяемое в Windows NT.

Из соображений совместимости поля, которые встречаются в элементах каталога для коротких имен формата FAT12 и FAT16, находятся на тех же местах и в элементах каталога для коротких имен формата FAT32. Остальные поля, которые встречаются только в элементах каталога для коротких имен формата FAT32, соответствуют зарезервированной области длиной 10 байт в элементах каталога для коротких имен форматов FAT12 и FAT16.

Как видно из рис. 6.3, для длинного имени файла используется несколько элементов каталога. Таким образом, появление длинных имен фактически привело к дальнейшему уменьшению количества файлов, находящихся в корневом каталоге. Поскольку длинное имя может содержать до 256 символов, всего один файл с полным длинным именем занимает до 25 элементов FAT (1 для имени 8.3 и еще 24 для

самого длинного имени). Таким образом, количество элементов корневого каталога VFAT уменьшается до 21. Очевидно, что это не вполне красивое решение, поэтому компания Microsoft советует избегать длинных имен в корневых каталогах при отсутствии системы FAT32, у которой количество элементов каталога просто требуемым образом увеличено[1].

Загрузочная запись для системы FAT32 несколько отличается от загрузочной записи FAT16. Так, например, в загрузочном секторе для тома с FAT32 в блоке DPB содержатся дополнительные поля, а те поля, что находятся в привычном для системы FAT16 месте, перенесены. Поэтому операционная система, в которой есть возможность работать с файловой системой FAT16, но нет системы управления файлами, понимающей спецификации FAT32, не может читать данные с томов, отформатированных под файловую систему FAT32. В загрузочном секторе для файловой системы FAT32 по-прежнему байты с 00H по 0AH содержат команду перехода и OEM ID, а в байтах с 0BH по 59H содержатся данные блока параметров диска (PDB). Отличие заключается именно в несколько иной структуре блока DPB (табл. 6.5).

Таблица 6.5. Структура загрузочной записи для FAT32

Смещение поля, байт	Длина поля, байт	Обозначение поля	Содержимое поля
00H (0)	3	JUMP 3EH	Безусловный переход на начало системного загрузчика
03H (3)	8		Системный идентификатор
0BH (11)	2	SectSize	Размер сектора, байт
0DH (13)	1	ClastSize	Число секторов в кластере
0EH (14)	2	ResSecs	Число зарезервированных секторов, для FAT32 равно 32
10H (16)	1	FATcnt	Число копий FAT
11H (17)	2	RootSize	0000H
13H (19)	2	TotSecs	0000H
15H (21)	1	Media	Дескриптор носителя
16H (22)	2	FATsize	0000H
18H (24)	2	TrkSecs	Число секторов на дорожке
1AH (26)	2	HeadCnt	Число рабочих поверхностей
1CH (28)	4	HidnSecs	Число скрытых секторов (располагаются перед загрузочным сектором). Используется при загрузке для вычисления абсолютного смещения корневого каталога и данных

[1] Помните и о том, что длина полной файловой спецификации, включающей путь и имя файла (длинное или в формате 8.3), тоже ограничивается 260 символами. FAT32 успешно справляется с проблемой длинных имен в корневом каталоге, но проблема с ограничением длины полной файловой спецификации остается. По этой причине Microsoft рекомендует ограничивать длинные имена 75–80 символами, чтобы оставить достаточно места для пути (180–185 символов).

Смещение поля, байт	Длина поля, байт	Обозначение поля	Содержимое поля
20H (32)	4		Число секторов на логическом диске
24H (36)	4		Число секторов в таблице FAT
28H (37)	2		Расширенные флаги
2AH (38)	2		Версия файловой системы
2CH (39)	4		Номер кластера для первого кластера корневого каталога
34H (43)	2		Номер сектора с резервной копией загрузочного сектора
36H (54)	12		Зарезервировано

Заметим, что загрузочная запись для диска с FAT32 занимает не один сектор, как в FAT16 и FAT12, а три. Резервная загрузочная запись, как правило, располагается в секторах 7–9.

Файловая система HPFS

Файловая система HPFS (High Performance File System — высокопроизводительная файловая система) впервые появилась в операционных системах OS/2 1.2 и LAN Manager. Она была разработана совместными усилиями лучших специалистов компаний IBM и Microsoft на основе опыта IBM по созданию файловых систем MVS, VM/CMS и виртуального метода доступа[1]. Архитектура HPFS начала создаваться как файловая система для многозадачного режима и была призвана обеспечить высокую производительность при работе с файлами на дисках большого размера.

HPFS стала первой файловой системой для персональных компьютеров, в которой была реализована поддержка длинных имен [26]. HPFS, как и FAT, как и многие другие файловые системы, обладает структурой каталогов, но в ней также предусмотрены автоматическая сортировка каталогов и специальные *расширенные атрибуты* (Extended Attributes, EAs)[2], упрощающие обеспечение безопасности на файловом уровне и создание множественных имен. Помимо расширенных атрибутов, каждый из которых концептуально подобен переменной окружения, HPFS по историческим причинам поддерживает те же самые атрибуты, что и файловая система FAT. Но самым главным отличием этой системы все же являются базовые принципы хранения информации о местоположении файлов.

Принципы размещения файлов на диске, положенные в основу HPFS, увеличивают как производительность файловой системы, так и ее надежность и отказоустойчивость. Для достижения этих целей предложено несколько идей:

---

[1] Так, со стороны компании Microsoft проектом руководил известный системщик Гордон Литвин (Gordon Letwin).

[2] Расширенные атрибуты позволяют хранить дополнительную информацию о файле. Например, каждому файлу может быть сопоставлено его уникальное графическое изображение (значок, миниатюра), описание файла, комментарий, сведения о владельце файла и т. д.

- размещение каталогов в середине дискового пространства;
- использование методов бинарных сбалансированных деревьев для ускорения поиска информации о файле;
- рассредоточение информации о местоположении файловых записей по всему диску, при том что записи каждого конкретного файла размещаются (по возможности) в смежных секторах и поблизости от данных об их местоположении.

Действительно, прежде всего, HPFS пытается расположить файл в смежных кластерах или, если такой возможности нет, поместить его на диск таким образом, чтобы *экстенты* (extents)[1] файла физически были как можно ближе друг к другу. Такой подход существенно сокращает время позиционирования (seek time) головок записи/чтения жесткого диска и время ожидания (rotational latency)[2]. Можно сказать, что файловая система HPFS имеет, по сравнению с FAT, следующие основные преимущества:

- высокая производительность;
- надежность;
- поддержка расширенных атрибутов, позволяющих более гибко управлять доступом к файлам и каталогам;
- эффективное использование дискового пространства.

Все эти преимущества обусловлены структурой диска HPFS. Рассмотрим ее более подробно (рис. 6.4).

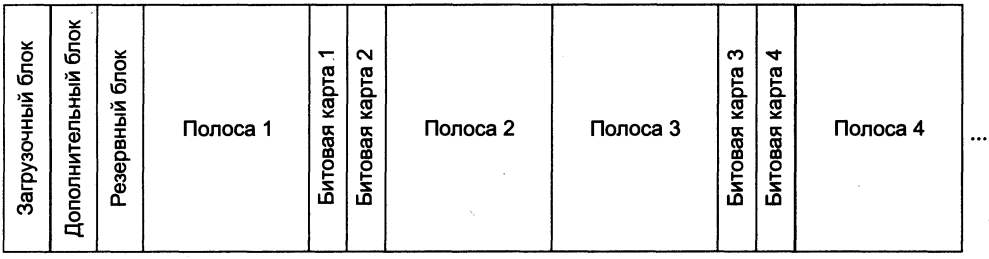


Рис. 6.4. Структура раздела HPFS

В начале диска расположено несколько управляющих блоков. Все остальное дисковое пространство в HPFS разбито на множество областей из смежных секторов, или *полос* (bands). В каждой такой области располагаются и собственно сами данные файлов, и вспомогательная служебная информация о свободных или занятых секторах в этой области. Каждая полоса занимает на диске пространство в 8 Мбайт и имеет собственную *битовую карту* (bit map) распределения секторов, которая, с одной стороны, напоминает таблицу размещения файлов FAT, но, с другой, суще-

[1] Экстент — фрагмент файла, располагающийся в смежных секторах диска. Файл имеет, по крайней мере, один экстент, если он не фрагментирован, в противном случае — несколько экстентов.

[2] Время ожидания — это задержка между установкой головки чтения/записи на нужную дорожку диска и началом чтения данных с диска. Эта задержка обусловлена тем, что система вынуждена ждать, пока диск не повернется таким образом, чтобы нужный сектор оказался под головкой чтения/записи.

ственно от нее отличается. Эти битовые карты показывают, какие секторы данной полосы заняты, а какие свободны. Каждому сектору полосы данных соответствует один бит в ее битовой карте.

Если бит имеет значение 1, то соответствующий сектор занят, если 0 — свободен. Битовые карты двух полос располагаются на диске рядом, также располагаются и сами полосы. То есть последовательность полос и карт выглядит следующим образом: битовая карта, битовая карта, полоса данных, полоса данных, битовая карта, битовая карта и т. д. Такое расположение полос и битовых карт позволяет непрерывно разместить на жестком диске файл размером до 16 Мбайт и в то же время не удалять от самих файлов информацию об их местонахождении.

Очевидно, что если бы на весь логический диск была бы только одна адресная структура данных, как это сделано в FAT, то для работы с ней приходилось бы перемещать головки чтения/записи в среднем через половину диска. Именно для того, чтобы избежать таких потерь, в HPFS диск разбит на полосы. Получается как бы распределенная структура данных (в данном случае — битовая карта) с информацией об используемых и свободных блоках.

Дисковое пространство в HPFS выделяется не кластерами, как в FAT, а *блоками*. В имеющейся на сегодня реализации размер блока равен одному сектору, но, в принципе, он мог бы быть и иного размера. По сути дела, блок — это и есть кластер. Размещение файлов в таких небольших блоках позволяет более эффективно использовать пространство диска, так как непроизводительные потери свободного места составляют в среднем всего 256 байт на каждый файл. Вспомните, чем больше размер кластера, тем больше места на диске расходуется напрасно. Например, кластер на отформатированном под FAT диске объемом от 512 до 1023 Мбайт имеет размер 16 Кбайт. Следовательно, непродуктивные потери свободного пространства на таком разделе в среднем составляют 8 Кбайт (8192 байт) на один файл, в то время как на разделе HPFS эти потери всегда будут составлять всего 256 байт на файл. Таким образом, на каждый файл экономится почти 8 Кбайт.

На рис. 6.4 показано, что помимо полос с записями файлов и битовых карт на томе (volume)[1] с HPFS имеются еще три информационные структуры. Это так называемый *загрузочный блок* (boot block), *дополнительный блок* (super block) и *резервный блок* (spare block). Загрузочный блок OS/2 располагается в секторах с 0 по 15; он содержит имя тома, его серийный номер, блок параметров BIOS[2] и программу начальной загрузки. Программа начальной загрузки находит программу `OS2LDR`, считывает ее в память и передает управление на эту программу загрузки операционной системы, которая, в свою очередь, загружает с диска в память ядро OS/2 — программу `OS2KRNL`. И уже `OS2KRNL` с помощью сведений из файла `CONFIG.SYS` загружает в память все необходимые программные модули и блоки данных.

В дополнительном блоке содержится указатель на *список битовых карт* (bitmap block list). В этом списке перечислены все блоки на диске, в которых расположены

[1] По сути дела, том — это не что иное, как раздел, или логический диск.

[2] Блок параметров BIOS содержит информацию о жестком диске — количестве цилиндров и головок диска, числе секторов на дорожке. Эта информация используется программными модулями HPFS для поиска конкретного сектора (блока), поскольку все блоки пронумерованы 32-разрядными числами.

битовые карты, используемые для обнаружения свободных секторов. Также в дополнительном блоке хранится указатель на *список дефектных блоков* (bad block list), указатель на *полосу каталогов* (directory band), указатель на *файловый узел* (File node, F-node) корневого каталога, а также дата последней проверки раздела программой CHKDSK. В списке дефектных блоков перечислены все поврежденные секторы (блоки) диска. Когда система обнаруживает поврежденный блок, он вносится в этот список и для хранения информации больше не используется. Кроме того, в дополнительном блоке содержится информация о размере полосы. Напомним, что в имеющейся реализации HPFS размер полосы равен 8 Мбайт. В принципе, его можно было бы сделать и больше. Дополнительный блок размещается в секторе с номером 16 логического диска, на котором установлена файловая система HPFS.

Резервный блок содержит указатель на *карту* (HotFix map), или *области* (HotFix areas), *аварийного замещения*, указатель на *список свободных запасных блоков каталогов* (directory emergency free block list), используемых для операций на почти переполненном диске, и ряд системных флагов и дескрипторов. Резервный блок размещается в 17-м секторе диска и обеспечивает высокую отказоустойчивость файловой системы HPFS, позволяя восстанавливать поврежденные данные на диске и перемещать их в надежное место.

Файлы и каталоги в HPFS базируются на фундаментальном объекте, уже упоминавшемся файловом узле[1]. Эта структура характерна для HPFS, и аналога в файловой системе FAT у нее нет. Каждый файл и каталог диска имеет свой файловый узел. Каждый файловый узел занимает один сектор и всегда располагается поблизости от своего файла или каталога (обычно — непосредственно перед файлом или каталогом). Файловый узел содержит размер файла и первые 15 символов имени файла, специальную служебную информацию, статистику по доступу к файлу, расширенные атрибуты файла и *список управления доступом* (Access Control List, ACL) или только часть этого списка, если он очень большой, ассоциативную информацию о расположении и подчинении файла и т. д. Структура распределения информации в файловом узле может иметь несколько форм, в зависимости от размера каталога или файлов. HPFS рассматривает файл как совокупность одного или более секторов. Из прикладной программы этого не видно; файл прикладной программе представляется как непрерывный поток байтов. Если расширенные атрибуты слишком велики для файлового узла, то в него записывается указатель на них.

Сокращенное имя файла (в формате 8.3) используется, когда файл с длинным именем копируется или перемещается на диск с системой FAT, которая не допускает подобных имен. Сокращенное имя образуется из первых 8 символов оригинального имени файла, точки и первых 3 символов расширения имени, если расширение имеется. Если в имени файла присутствует несколько точек, что не противоречит правилам именования файлов в HPFS, то для расширения сокращенного имени используются 3 символа после самой последней из этих точек.

---

[1] Файловый узел — это структура, в которой содержится информация о расположении файла и о его расширенных атрибутах.

Так как HPFS при размещении файла на диске стремится избежать его фрагментации, то структура информации, содержащаяся в файловом узле, достаточно проста. Если файл непрерывен, то его размещение на диске описывается двумя 32-разрядными числами. Первое число представляет собой указатель на первый блок файла, а второе — длину экстента, то есть число следующих друг за другом блоков, принадлежащих файлу[1]. Если файл фрагментирован, то размещение его экстентов описывается в файловом узле дополнительными парами 32-разрядных чисел. Фрагментация происходит, когда на диске нет непрерывного свободного участка, достаточно большого, чтобы разместить файл целиком. В этом случае файл приходится разбивать на несколько экстентов и располагать их на диске раздельно. Файловая система HPFS старается разместить экстенты фрагментированного файла как можно ближе друг к другу, чтобы сократить время позиционирования головок чтения/записи жесткого диска. Для этого HPFS использует статистику, а также старается условно резервировать хотя бы 4 Кбайт места в конце файлов, которые растут. Еще один способ снижения фрагментации файлов — это размещение в разных полосах диска файлов, растущих навстречу друг другу, а также файлов, открытых разными потоками выполнения или процессами.

В файловом узле можно разместить информацию максимум о 8 экстентах файла. Если файл имеет больше экстентов, то в его файловый узел записывается указатель на *блок размещения* (allocation block), который может содержать до 40 указателей на экстенты, или, по аналогии с блоком дерева каталогов, на другие блоки размещения. Таким образом, двухуровневая структура блоков размещения может хранить информацию о 480 секторах, что позволяет работать с файлами размером до 7,68 Гбайт. На практике размер файла не может превышать 2 Гбайт, но это обусловлено текущей реализацией интерфейса прикладного программирования [26].

Упоминавшаяся выше полоса каталогов находится в центре диска и используется для хранения каталогов. Как и все остальные полосы, она имеет размер 8 Мбайт. Однако если она будет полностью заполнена, HPFS начинает располагать каталоги файлов в других полосах. Расположение этой информационной структуры в середине диска значительно сокращает среднее время позиционирования головок чтения/записи, тем более что обращения к корневому каталогу достаточно часты. Действительно, для перемещения головок чтения/записи из произвольного места диска в его центр требуется в два раза меньше времени, чем для перемещения к краю диска, где находится корневой каталог в случае файловой системы FAT. Уже только одно это обеспечивает существенно более высокую производительность файловой системы HPFS по сравнению с FAT. Аналогичное замечание справедливо и для системы NTFS, которая тоже располагает свою главную таблицу файлов в начале дискового пространства, а не в его середине (см. раздел «Файловая система NTFS»). Тестирование показывает, что HPFS является самой быстрой файловой системой.

Однако существенно больший вклад в производительность HPFS (по сравнению с размещением полосы каталогов в середине логического диска) дает использова-

[1] Из этого следует, что максимальный объем диска может составить $(2^{32} - 1) \times 512 = 2$ Тбайт.

ние метода *сбалансированных двоичных деревьев* для хранения и поиска информации о местонахождении файлов. Как известно, в файловой системе FAT каталог имеет линейную неупорядоченную специальным образом структуру, поэтому при поиске файла требуется последовательно просматривать его с самого начала. В HPFS структура каталога представляет собой сбалансированное дерево с записями, расположенными в алфавитном порядке (рис. 6.5). Каждая запись, входящая в состав двоичного дерева (Binary Tree, B-Tree), содержит атрибуты файла, указатель на соответствующий файловый узел, информацию о времени и дате создания файла, о времени и дате последнего обновления и обращения, об объеме данных, содержащих расширенные атрибуты, счетчик обращений к файлу, информацию о длине имени файла и само имя, другую информацию.

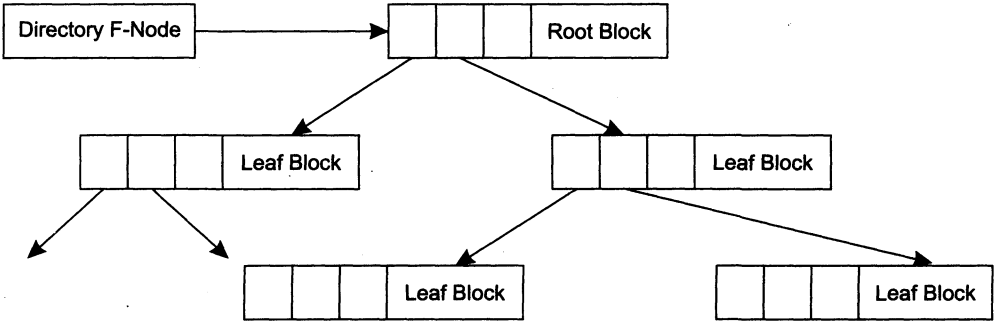


Рис. 6.5. Сбалансированное двоичное дерево

Файловая система HPFS при поиске файла в каталоге просматривает только необходимые ветви двоичного дерева, отбрасывая те записи каталога, про которые заведомо известно, что они не относятся к искомому файлу. Например, если имя файла начинается с символа, расположенного в первой части используемого алфавита, то незачем искать его среди записей каталога, описывающих файлы, имена которых начинаются с символов, расположенных во второй части этого алфавита. Далее, если искомый элемент каталога расположен во второй половине первой части (то есть во второй четверти), то незачем перебирать имена файлов, расположенных в первой четверти каталога. И так далее. Очевидно, что такой метод во много раз эффективнее, чем последовательное чтение всех записей в каталоге, что имеет место в системе FAT. Для того чтобы найти искомый файл в каталоге (точнее, указатель на его информационную структуру F-node), организованном на принципах сбалансированных двоичных деревьев, большинство записей вообще читать не нужно. В результате для поиска информации о файле необходимо выполнить существенно меньшее количество операций чтения с диска.

Действительно, если, например, каталог содержит 4096 файлов, то файловая система FAT потребует чтение в среднем 64 секторов для поиска нужного файла внутри такого каталога, в то время как HPFS осуществит чтение всего только 2–4 секторов (в среднем) и найдет искомый файл. Несложные расчеты позволяют увидеть явные преимущества HPFS над FAT. Так, например, при использовании 40 входов на блок блоки дерева каталогов с двумя уровнями могут содержать 1640 вхо-

дов, а дерева каталогов с тремя уровнями — уже 65 640 входов. Другими словами, некоторый файл может быть найден в типичном каталоге из 65 640 файлов максимум за три обращения. Это намного лучше файловой системы FAT, где в самом плохом случае для нахождения файла нужно прочитать более чем 4000 секторов.

Размер каждого из блоков, в терминах которых выделяются каталоги в текущей реализации HPFS, равен 2 Кбайт. Размер записи, описывающей файл, зависит от размера имени файла. Если имя занимает 13 байт (для формата 8.3) то 2-килобайтовый блок вмещает до 40 дескрипторов файлов. Блоки связаны друг с другом посредством списковой структуры (как и дескрипторы экстентов) для облегчения последовательного обхода.

При переименовании файлов может возникнуть так называемая перебалансировка дерева. Фактически, попытка переименования может потерпеть неудачу из-за недостатка дискового пространства, даже если файл непосредственно в размерах не увеличился. Во избежание этого «бедствия» HPFS поддерживает маленький пул свободных блоков, которые могут использоваться при «аварии». Эта операция может потребовать выделения дополнительных блоков на заполненном диске. Указатель на этот пул свободных блоков сохраняется в резервном блоке.

Важное значение для повышения скорости работы с файлами имеет снижение их фрагментации. В HPFS считается, что если файл содержит больше одного экстента, он считается фрагментированным. Снижение фрагментации файлов сокращает время позиционирования и время ожидания за счет уменьшения количества перемещений головок, необходимых для доступа к данным файла. Алгоритмы работы файловой системы HPFS функционируют таким образом, чтобы по возможности размещать файлы в последовательных смежных секторах диска, что в последующем обеспечит максимально быстрый доступ к данным. В системе FAT, наоборот, запись следующей порции данных в первый же свободный кластер неизбежно приводит к фрагментации файлов. То есть HPFS записывает данные не в первый попавшийся сектор, а, если это предоставляется возможным, в смежные секторы диска. Это позволяет несколько снизить число перемещений головок чтения/записи от дорожки к дорожке. Когда данные дописываются в существующий файл, HPFS сразу же резервирует как минимум 4 Кбайт непрерывного пространства на диске. Если же часть этого пространства не потребовалась, то после закрытия файла она высвобождается для дальнейшего использования. Файловая система HPFS равномерно размещает непрерывные файлы по всему диску для того, чтобы впоследствии без фрагментации обеспечить их возможное увеличение. Если же файл не может быть увеличен без нарушения его непрерывности, HPFS опять-таки резервирует 4 Кбайт смежных блоков как можно ближе к основной части файла с целью сократить время позиционирования головок чтения/записи и время поиска соответствующего сектора.

Очевидно, что степень фрагментации файлов на диске зависит как от числа расположенных на нем файлов, их размеров и размеров самого диска, так и от характера и интенсивности самих дисковых операций. Незначительная фрагментация файлов практически не сказывается на быстродействии операций с файлами. Файлы, состоящие из 2–3 экстентов, практически не снижают производительности HPFS,

так как эта файловая система следит за тем, чтобы области данных, принадлежащие одному и тому же файлу, располагались как можно ближе друг к другу. Файл из трех экстентов имеет только два нарушения непрерывности, и, следовательно, для его чтения потребуется всего лишь два небольших перемещения головки диска. Программы (утилиты) дефрагментации, имеющиеся для этой файловой системы по умолчанию, считают наличие двух-трех экстентов у файла нормой[1]. Практика показывает, что в среднем на диске имеется не более 2 % файлов, имеющих три и более экстентов [26]. Даже общее количество фрагментированных файлов, как правило, не превышает 3 %. Такая ничтожная фрагментация оказывает пренебрежимо малое влияние на общую производительность системы.

Теперь кратко рассмотрим вопрос надежности хранения данных в HPFS. Любая файловая система должна обладать средствами исправления ошибок, возникающих при записи информации на диск. Система HPFS для этого использует *механизм аварийного замещения* (HotFix).

Если файловая система HPFS сталкивается с проблемой в процессе записи данных на диск, она выводит на экран соответствующее сообщение об ошибке. Затем HPFS сохраняет информацию, которая должна была быть записана в дефектный сектор, в одном из запасных секторов, заранее зарезервированных на этот случай. Список свободных запасных блоков хранится в резервном блоке HPFS. При обнаружении ошибки во время записи данных в нормальный блок HPFS выбирает один из свободных запасных блоков и сохраняет эти данные в нем. Затем файловая система обновляет карту аварийного замещения в резервном блоке. Эта карта представляет собой просто пары двойных слов, каждое из которых является 32-разрядным номером сектора. Первый номер указывает на дефектный сектор, а второй — на тот сектор среди имеющихся запасных секторов, который и был выбран для замены плохого. После замены дефектного сектора запасным карта аварийного замещения записывается на диск, и на экране появляется всплывающее окно, информирующее пользователя о произошедшей ошибке записи на диск. Каждый раз, когда система выполняет запись или чтение сектора диска, она просматривает карту аварийного замещения и подменяет все номера дефектных секторов номерами запасных секторов с соответствующими данными. Следует заметить, что это преобразование номеров существенно не влияет на производительность системы, так как оно выполняется только при физическом обращении к диску, а не при чтении данных из дискового кэша. Очистка карты аварийного замещения автоматически выполняется программой CHKDSK при проверке диска HPFS. Для каждого замещенного блока (сектора) программа CHKDSK выделяет новый сектор в наиболее подходящем для файла (которому принадлежат данные) месте жесткого диска. Затем программа перемещает данные из запасного блока в этот сектор и обновляет информацию о положении файла, что может потребовать новой балансировки дерева блоков размещения. После этого CHKDSK вносит поврежденный сектор

[1] Например, программа HPFSOPT из набора утилит GammaTech по умолчанию не дефрагментирует файлы, состоящие из трех и менее экстентов, а файлы, которые имеют большее количестве экстентов, приводятся к 2 или 3 экстентам, ежели это возможно (файлы объемом в несколько десятков мегабайтов всегда будут фрагментированы, ибо максимально возможный размер экстента, как вы помните, составляет 8 Мбайт).

в список дефектных блоков, который хранится в дополнительном блоке HPFS, и возвращает освобожденный сектор в список свободных запасных секторов резервного блока. Затем удаляет запись из карты аварийного замещения и записывает отредактированную карту на диск.

Все основные файловые объекты в HPFS, в том числе файловые узлы, блоки размещения и блоки каталогов, имеют уникальные 32-разрядные идентификаторы и указатели на свои родительские и дочерние блоки. Файловые узлы, кроме того, содержат сокращенное имя своего файла или каталога. Избыточность и взаимосвязь файловых структур HPFS позволяют программе CHKDSK полностью восстанавливать файловую структуру диска, последовательно анализируя все файловые узлы, блоки размещения и блоки каталогов. Руководствуясь собранной информацией, CHKDSK реконструирует файлы и каталоги, а затем заново создает битовые карты свободных секторов диска. Запуск программы CHKDSK следует осуществлять с соответствующими ключами. Так, например, один из вариантов работы этой программы позволяет найти и восстановить удаленные файлы.

HPFS относится к так называемым монтируемым файловым системам. Это означает, что она не встроена в операционную систему, а добавляется к ней при необходимости. Файловая система HPFS монтируется оператором **IFS** (Installable File System — монтируемая файловая система) в файле **CONFIG.SYS**. Оператор **IFS** всегда помещается в первой строке этого конфигурационного файла. В приводимом далее примере оператор **IFS** устанавливает (монтирует) файловую систему HPFS с кэшем в 2 Мбайт, длиной записи кэша в 8 Кбайт и автоматической процедурой проверки дисков **C:** и **D:**.

```
IFS=E:\OS2\HPFS.IFS /CACHE:2048 /CRECL:4 /AUTOCHECK:CD
```

Для запуска программы управления процессом кэширования следует прописать в файле **CONFIG.SYS** еще одну строку:

```
RUN=E:\OS2\CACHE.EXE /Lazy:On /BufferIdle:2000 /DiskIdle:4000 /MaxAge:8000 /DirtyMax:256
/ReadAhead:On
```

В этой строке включается режим отложенной записи, устанавливаются параметры работы этого режима, а также включается режим упреждающего чтения данных, что в целом позволяет существенно сократить количество обращений к диску и ощутимо повысить быстродействие файловой системы. Так, ключ **Lazy** с параметром **On** включает отложенную запись, а с параметром **Off** — выключает. Ключ **BufferIdle** определяет время в миллисекундах, в течение которого буфер кэша должен оставаться в неактивном состоянии, чтобы стало возможным осуществить запись данных из кэша на диск. По умолчанию (то есть если не прописывать этот ключ явным образом) это время равно 500 мс. Ключ **DiskIdle** задает время (в миллисекундах), по истечении которого диск должен оставаться в неактивном состоянии, чтобы стало возможным осуществить запись данных из кэша на диск. По умолчанию это время равно 1 с. Этот параметр позволяет избежать записи из кэша на диск во время выполнения других операций с диском.

Ключ **MaxAge** задает время (тоже в миллисекундах), по истечении которого часто сохраняемые в кэше данные помечаются как «устаревшие» и при переполнении кэша могут быть замещены новыми. По умолчанию это время равно 5 с.

Остальные подробности установки параметров и возможные значения ключей имеются в файлах помощи операционной системы OS/2 Warp. Однако здесь следует сказать и еще об одной системе управления файлами — речь идет о реализации HPFS для работы на серверах, функционирующих под управлением OS/2. Эта система управления файлами получила название HPFS386.IFS. Ее принципиальные отличия от системы управления файлами HPFS.IFS, прежде всего, заключаются в том, что она позволяет посредством более полного использования технологии расширенных атрибутов организовать ограничения на доступ к файлам и каталогам с помощью соответствующих списков управления доступом (ACL). Эта технология, как известно, используется в файловой системе NTFS. Кроме того, в системе управления файлами HPFS386.IFS, в отличие от HPFS.IFS, нет ограничений на объем памяти, выделяемой для кэширования файловых записей. Иными словами, при наличии достаточного объема оперативной памяти объем файлового кэша может составлять несколько десятков мегабайтов, в то время как для обычной HPFS.IFS этот объем не может превышать двух мегабайтов, что по сегодняшним понятиям безусловно мало. Наконец, при установке режимов работы файлового кэша HPFS386.IFS есть возможность явным образом указать алгоритм упорядочивания запросов на запись. Наиболее эффективным алгоритмом можно считать так называемый «элеваторный», при котором операции записи данных из кэша на диск предварительно упорядочиваются таким образом, чтобы минимизировать время, отводимое на позиционирование головок чтения/записи. Головки чтения/записи при этом перемещаются от внешних цилиндров к внутренним и по ходу своего движения осуществляют запись и чтение данных в соответствии со специальным образом упорядочиваемых списков запросов на дисковые операции. Напомним, что такой алгоритм управления запросами на дисковые операции имеет название циклического сканирования (C-Scan).

Приведем пример записи строк в конфигурационном файле **CONFIG.SYS**, в которых устанавливается система HPFS386.IFS и определяются параметры работы ее подсистемы кэширования:

```
IFS=E:\IBM386FS\HPFS386.IFS /AUTOCHECK:EGH
RUN=E:\IBM386FS\CACHE386.EXE /Lazy:On /BufferIdle:4000 /MaxAge:20000
```

Эти записи следует понимать следующим образом. При запуске операционной системы в случае обнаружения флага, означающего, что не все файлы были закрыты в процессе предыдущей работы, система управления файлами HPFS386.IFS сначала запустит программу проверки целостности файловой системы для томов **E:**, **G:** и **H:**. Для кэширования файлов при работе этой системы управления файлами устанавливается режим отложенной записи со временем жизни буферов до 20 с. Остальные параметры и, в частности, алгоритм обслуживания запросов, устанавливаются в файле **HPFS386.INI**, который в данном случае располагается в каталоге **E:\IBM386FS**.

Опишем кратко некоторые наиболее интересные параметры, управляющие работой кэша. Прежде всего, отметим, что файл **HPFS386.INI** разбит на несколько секций. В настоящий момент мы рассмотрим секцию **[ULTIMEDIA]**:

```
[ULTIMEDIA]
QUEUESORT={FIFO|ELEVATOR|DEFAULT|CURRENT}
QUEUEMETHOD={PRIORITY|NOPRIORITY|DEFAULT|CURRENT}
QUEUEDEPTH={1...255|DEFAULT|CURRENT}
```

Параметр QUEUESORT задает способ ведения очереди запросов к диску. Он может принимать значения FIFO, ELEVATOR, DEFAULT и CURRENT. Если задано значение FIFO, то каждый новый запрос просто добавляется в конец очереди, то есть запросы выполняются в том порядке, в котором они поступают в систему. Однако можно упорядочить некоторое количество запросов по возрастанию номеров дорожек. Если задано значение ELEVATOR, то включается режим поддержки упорядоченной очереди запросов. При этом запросы начинают обрабатываться по алгоритму ELEVATOR (он же C-SCAN, или «режим циклического сканирования» [11, 26]). Напомним, что этот алгоритм подразумевает, что головка чтения/записи сканирует диск в выбранном направлении (например, в направлении возрастания номеров дорожек), останавливаясь для выполнения запросов, находящихся на пути следования.

Если для параметра QUEUESORT задано значение DEFAULT, то выбирается значение по умолчанию, которым является ELEVATOR. Если задано значение CURRENT, то остается в силе тот алгоритм, который был выбран при инициализации менеджером дисковых операций (DASD-manager).

Параметр QUEUEMETHOD определяет, должны ли учитываться приоритеты запросов при построении очереди. Он может принимать значения PRIORITY, NOPRIORITY, DEFAULT и CURRENT. Если задано значение NOPRIORITY, то все запросы включаются в общую очередь, а их приоритеты игнорируются. Если задано значение PRIORITY, то менеджер дисковых операций будет поддерживать несколько очередей запросов, по одной на каждый приоритет. Когда менеджер дисковых операций передает запросы на исполнение драйверу диска, он сначала выбирает запросы из самой приоритетной очереди, потом из менее приоритетной и т. д. Приоритеты назначает система управления файлами HPFS386.IFS, а распределены они следующим образом.

1. Останов операционной системы с закрытием всех файлов или экстренная запись из-за сбоя питания. Это самый высокий приоритет.
2. Страничный обмен.
3. Обычные запросы активного сеанса (foreground session), то есть задачи, с которой в данный момент работает пользователь и окно которой является активным.
4. Обычные запросы фонового сеанса (background session), то есть задачи, запущенной пользователем, с которой он в данный момент непосредственно не работает (говорят, что эта задача выполняется на фоне текущих активных вычислений). Приоритеты 3 и 4 равны, если в файле CONFIG.SYS имеется строка:
   ```
   RIORITY_DISK_IO=NO
   ```
5. Опережающее чтение и низкоприоритетные запросы страничного обмена (предварительная выборка страниц).
6. Отложенная запись и прочие запросы, не требующие немедленной реакции.
7. Предварительная выборка. Это самый низкий приоритет.

Если для параметра QUEUEMETHOD задано значение DEFAULT, то выбирается значение по умолчанию, которым является PRIORITY. Если задано значение CURRENT, то остается в силе тот метод, который был выбран менеджером дисковых операций при инициализации.

Параметр QUEUEDEPTH задает глубину просмотра очереди при выборке запросов. Он может принимать значения из диапазона (1...255), а также DEFAULT и CURRENT. Если в качестве значения параметра QUEUEDEPTH задано число, то оно определяет количество запросов, которые должны находиться в очереди дискового адаптера одновременно. Например, для SCSI-адаптеров имеет смысл поддерживать такую длину очереди, при которой они смогут загрузить все запросы в свои аппаратные структуры. Если очередь запросов к адаптеру будет слишком короткой, то аппаратура будет работать с неполной загрузкой, а если она будет слишком длинной, драйвер SCSI-адаптера окажется перегруженным «лишними» запросами. Поэтому разумным значением для QUEUEDEPTH является число, немного превышающее длину аппаратной очереди команд адаптера. Если для параметра QUEUEDEPTH задано значение DEFAULT, то глубина просмотра очереди определяется автоматически на основании значения, которое рекомендовано драйвером дискового адаптера. Если задано значение CURRENT, то глубина просмотра очереди не изменяется. В текущей реализации значение CURRENT эквивалентно значению DEFAULT.

Итак, текущие умолчания для системы управления файлами HPFS386.IFS имеют вид:

```
QUEUESORT=FIFO
QUEUEMETHOD=DEFAULT
QUEUEDEPTH=2
```

А текущие умолчания для менеджера дисковых операций таковы:

```
QUEUESORT=ELEVATOR
QUEUEMETHOD=PRIORITY
QUEUEDEPTH=<зависит от адаптера диска>
```

Значения, устанавливаемые для менеджера дисковых операций по умолчанию, можно поменять с помощью параметра /QF:

```
BASEDEV=OS2DASD.DMD /QF:{1|2|3}
```

Здесь: 1 соответствует выражению QUEUESORT=FIFO, 2 — выражению QUEUEMETHOD=NOPRIORITY, 3 — выражениям QUEUESORT=FIFO и QUEUEMETHOD=NOPRIORITY.

Наконец, скажем еще несколько слов о монтируемых системах управления файлами (Installable File System, IFS), представляющих собой специальное системное программное обеспечение («драйверы») для доступа к разделам, отформатированным под другую файловую систему. Это очень удобный и мощный механизм добавления в ОС новых файловых систем и замены одной системы управления файлами на другую. Сегодня, например, для OS/2 уже реально существуют IFS-модули для файловой системы VFAT (FAT с поддержкой длинных имен), FAT32, Ext2FS (файловая система Linux), NTFS (правда, пока только для чтения). Для работы с данными на компакт-дисках имеется система CDFS.IFS. Есть и система управления файлами FTP.IFS, позволяющая монтировать ftp-архивы как локальные диски. Механизм монтируемых систем управления файлами был перенесен и в систему Windows NT.

Файловая система NTFS

В название файловой системы NTFS (New Technology File System — файловая система новой технологии) входят слова «новая технология». Действительно, файловая система NTFS по сравнению с широко известной FAT16 (и даже FAT32)

содержит ряд значительных усовершенствований и изменений. С точки зрения пользователей файлы по-прежнему хранятся в каталогах, ныне при работе в среде Windows часто называемых *папками* (folders). Однако в ней появилось много новых особенностей и возможностей.

Основные возможности файловой системы NTFS

При проектировании NTFS особое внимание было уделено надежности, механизмам ограничения доступа к файлам и каталогам, расширенной функциональности, поддержке дисков большого объема и пр. Начала разрабатываться эта система в рамках проекта OS/2 v.3, поэтому она переняла многие интересные особенности файловой системы HPFS.

Надежность

Высокопроизводительные компьютеры и системы совместного использования должны обладать повышенной надежностью, которая является ключевым элементом структуры и функционирования NTFS. Система NTFS обладает определенными средствами самовосстановления. Она поддерживает различные механизмы проверки целостности системы, включая ведение журналов транзакций, позволяющих воспроизвести файловые операции записи по специальному системному журналу. При протоколировании файловых операций система управления файлами фиксирует в специальном служебном файле (журнале) происходящие изменения. В начале операции, связанной с изменением файловой структуры, делается соответствующая пометка. Если во время файловых операций происходит какой-нибудь сбой, то из-за упомянутой отметки операция остается помеченной как незавершенная. При выполнении процедуры проверки целостности файловой системы после перезагрузки машины эти незавершенные операции отменяются, и файлы возвращаются в исходное состояние. Если же операция изменения данных в файлах завершается нормальным образом, то в файле журнала эта операция отмечается как завершенная.

Поскольку NTFS разрабатывалась как файловая система для серверов, для которых очень важно обеспечить бесперебойную работу без перезагрузок, в ней, как и в HPFS, для повышения надежности был введен механизм аварийной замены дефектных секторов резервными. Другими словами, если обнаруживается сбой при чтении данных, то система постарается прочесть эти данные, переписать их в специально зарезервированное для этой цели пространство диска, а дефектные сектора пометить как плохие и более к ним не обращаться.

Ограничения доступа к файлам и каталогам

Файловая система NTFS поддерживает объектную модель безопасности операционной системы Windows NT и рассматривает все тома, каталоги и файлы как самостоятельные объекты. Система NTFS обеспечивает безопасность на уровне файлов и каталогов. Это означает, что разрешения доступа к томам, каталогам и файлам могут зависеть от учетной записи пользователя и тех групп, к которым он принадлежит. Каждый раз, когда пользователь обращается к объекту файловой системы,

его разрешения на доступ проверяются по уже упоминавшемуся *списку управления доступом* (ACL) для данного объекта. Если пользователь обладает необходимым уровнем разрешений, его запрос удовлетворяется; в противном случае запрос отклоняется. Эта модель безопасности (см. подраздел «Модель безопасности Windows NT/2000/XP» в главе 11) применяется как при локальной регистрации пользователей на компьютерах с Windows NT, так и при удаленных сетевых запросах.

Расширенная функциональность

Система NTFS проектировалась с учетом возможного расширения. В ней были воплощены многие дополнительные возможности — повышенная отказоустойчивость, эмуляция других файловых систем, мощная модель безопасности, параллельная обработка потоков данных и создание файловых атрибутов, определяемых пользователем. Эта система также позволяет сжимать как отдельные файлы, так и целые каталоги. В последней, пятой, версии NTFS введена возможность шифрования хранимых файлов. Здесь следует, однако, заметить, что у шифрующей файловой системы пока больше недостатков, чем достоинств, поэтому на практике ее применять не рекомендуется.

Наконец, в системах Windows 2000/XP в случае использования файловой системы NTFS можно включить квотирование, при котором пользователи могут хранить свои файлы только в пределах отведенной им квоты на дисковое пространство.

Поддержка дисков большого объема

Система NTFS создавалась с расчетом на работу с большими дисками. Она уже достаточно хорошо проявляет себя при работе с томами объемом 300–400 Мбайт и выше. Чем больше объем диска и чем больше на нем файлов, тем больший выигрыш мы получаем, используя NTFS вместо FAT16 или FAT32. Максимально возможные размеры тома (и размеры файла) составляют 16 Эбайт (один экзабайт равен 2^{64} байт, или приблизительно 16 000 млрд гигабайт), в то время как при работе под Windows NT/2000/XP диск с FAT16 не может иметь размер более 4 Гбайт, а с FAT32 — 32 Гбайт. Количество файлов в корневом и некорневом каталогах при использовании NTFS не ограничено. Поскольку в основу структуры каталогов NTFS заложена эффективная структура данных, называемая «двоичным деревом», время поиска файлов в NTFS не связано линейной зависимостью с их количеством (в отличие от систем на базе FAT). Наконец, помимо немыслимых размеров томов и файлов, система NTFS также обладает встроенными средствами сжатия, что позволяет экономить дисковое пространство и размещать в нем больше файлов. Напомним, что сжатие можно применять как к отдельным файлам, так и целым каталогам и даже томам (и впоследствии отменять или назначать их по своему усмотрению).

Структура тома с файловой системой NTFS

Рассмотрим теперь структуру файловой системы NTFS. Наиболее полно она описана в [16] и [42]. Мы же здесь опишем только основные моменты.

Прежде всего, одним из основных понятий, используемых при работе с NTFS, является понятие *тома* (volume). Том означает логическое дисковое пространство, которое может быть воспринято как логический диск, то есть том может иметь букву (буквенный идентификатор) диска. Частным случаем тома является логический диск. Возможно также создание отказоустойчивого тома, занимающего несколько разделов, то есть поддерживается использование RAID-технологии. RAID — это сокращение от Redundant Array of Inexpensive Disks, что дословно переводится как «избыточный массив недорогих дисков». RAID-технология позволяет получать дисковые подсистемы из нескольких обычных дисков, которые обладают либо существенно более высоким быстродействием, либо более высокой надежностью, либо тем и другим одновременно. К сожалению, в файловой системе NTFS5, применяемой в Windows 2000/XP, для использования RAID-технологии в случае, когда эти системы устанавливаются не поверх старой системы Windows NT 4.0, а заново, требуются так называемые *динамические диски*. Это фирменный закрытый стандарт распределения дискового пространства, не имеющий ничего общего с тем промышленным стандартом, который использует главную загрузочную запись и был описан в предыдущей главе. Основным недостатком нового стандарта от Microsoft является абсолютная несовместимость с другими операционными системами. Другими словами, если жесткий диск с помощью оснастки Управление дисками был преобразован в динамический, то на этот компьютер более не удастся установить никакую операционную систему, а установленные ранее системы, отличные от Windows 2000/XP/2003, не смогут даже запуститься. Кроме этого, обратное преобразование динамического диска до так называемой «базовой модели» (так компания Microsoft назвала промышленный стандарт описания логической структуры диска) невозможно без полной потери данных. Единственным достоинством динамической модели дисков является возможность преобразования томов или изменения размера логического диска прямо «на лету», то есть без последующей обязательной перезагрузки операционной системы. Технологию изменения размеров дисковых томов «на лету» разработала фирма Veritas Software. Компания Microsoft лицензировала эту технологию, ввела дополнительные ограничения на ее использование и назвала динамическими дисками.

Как и многие другие файловые системы, NTFS делит все полезное дисковое пространство тома на кластеры — блоки данных, адресуемые как единицы данных. Файловая система NTFS поддерживает размеры кластеров от 512 байт до 64 Кбайт; неким стандартом же считается кластер размером 2 или 4 Кбайт. К сожалению, при увеличении размера кластера свыше 4 Кбайт становится невозможным сжимать файлы и каталоги.

Все дисковое пространство в NTFS делится на две неравные части (рис. 6.6). Первые 12 % диска отводятся под так называемую зону MFT (Master File Table — главная таблица файлов). Эта зона предназначена для таблицы MFT (с учетом ее будущего роста), представляющей собой специальный файл со служебной информацией, позволяющей определять местонахождение всех остальных файлов. Запись каких-либо данных в зону MFT невозможна — она всегда остается пустой, чтобы при росте MFT по возможности не было фрагментации. Остальные 88 % тома представляют собой обычное пространство для хранения файлов.

Рис. 6.6. Структура тома NTFS

Очевидно, что структуру данных, называемую главной таблицей файлов, можно рассматривать как файл. В этом файле MFT хранится информация обо всех остальных файлах диска, в том числе и о самом файле MFT. Таблица MFT поделена на записи фиксированного размера в 1 Кбайт, и каждая запись соответствует какому-либо файлу (в общем смысле этого слова). Первые 16 файлов носят служебный характер и недоступны через интерфейс операционной системы — они называются *метафайлами*, причем самый первый метафайл — это сам файл MFT. Часть диска с метафайлами — единственная часть диска, имеющая строго фиксированное положение. Копия этих же 16 записей таблицы MFT (для надежности, поскольку они очень важны) хранится в середине тома. Оставшаяся часть файла MFT может располагаться, как и любой другой файл, в произвольных местах диска — восстановить его положение можно с помощью самого файла MFT. Для этого достаточно взять первую запись таблицы MFT.

Упомянутые первые 16 файлов NTFS (метафайлы) являются служебными; каждый из них отвечает за какой-либо аспект работы системы. Метафайлы находятся в корневом каталоге тома NTFS. Их имена начинаются с символа «$», хотя получить какую-либо информацию о них стандартными средствами сложно. В табл. 6.6 приведены основные метафайлы и указано их назначение. Таким образом, можно узнать, например, сколько операционная система тратит на каталогизацию тома, посмотрев размер файла $MFT.

Таблица 6.6. Метафайлы NTFS

Имя метафайла	Описание
$MFT	Сам файл с таблицей MFT
$MFTmirr	Копия первых 16 записей таблицы MFT, размещенная посередине тома
$LogFile	Файл журнала
$Volume	Служебная информация — метка тома, версия файловой системы т. д.
$AttrDef	Список стандартных атрибутов файлов на томе
$	Корневой каталог
$Bitmap	Битовая карта свободного места тома
$Boot	Загрузочный сектор (если раздел загрузочный)

Имя метафайла	Описание
$Quota	Файл, в котором записаны права пользователей на использование дискового пространства (этот файл начал использоваться лишь в Windows 2000 с системой NTFS 5.0)
$Upcase	Файл с таблицей соответствия строчных и прописных букв в именах файлов. В NTFS имена файлов записываются в кодировке Unicode (всего доступно 65 тысяч различных символов, поэтому искать сточные и прописные эквиваленты символов — нетривиальная задача)

Итак, все файлы тома представлены в таблице MFT. За исключением собственно данных, в этой структуре хранится вся информация о файлах: имя файла, размер, положение на диске отдельных фрагментов и т. д. Если для информации не хватает одной записи MFT, то используются несколько записей, причем не обязательно последовательных. Если файл имеет не очень большой размер, тогда в ход идет довольно удачное решение: данные файла хранятся прямо в соответствующей записи таблицы MFT в оставшемся от служебных данных месте. Таким образом, файлы, занимающие не более сотни байтов, обычно не имеют своего «физического» воплощения в основной файловой области — все данные таких файлов хранятся прямо в таблице MFT.

Файл на томе в системе NTFS идентифицируется так называемой *файловой ссылкой* (file reference), которая представляется как 64-разрядное число. Файловая ссылка состоит из номера файла, который соответствует позиции его файловой записи в таблице MFT, и номера последовательности. Последний увеличивается всякий раз, когда данная позиция в MFT используется повторно, что позволяет файловой системе NTFS выполнять внутренние проверки целостности.

Каждый файл на диске в системе NTFS представлен с помощью *потоков данных* (streams)[1], то есть у файла нет «просто данных», а есть «потоки данных». Чтобы правильнее понять эту сущность (поток данных), достаточно знать, что один из потоков имеет привычный нам смысл — это собственно данные файла. Кстати, большинство атрибутов файла (за исключением основных) — это тоже потоки данных. Таким образом, получается, что основой файла является номер записи в таблице MFT, а все остальное, включая его потоки данных, не обязательно. Данный подход довольно удобен. Так, файлу можно назначить еще один поток данных, записав в него любые данные, например информацию об авторе и содержании файла, как это сделано в Windows 2000 (эта информация представлена на одной из вкладок диалогового окна свойств файла). Здесь имеется определенная аналогия с расширенными атрибутами в HPFS. Интересно, что эти дополнительные потоки не видны стандартными средствами для работы с файлами операционной системы: наблюдаемый размер файла — это лишь размер потока основных (традиционных) данных. Можно, к примеру, удалить файл нулевой длины, и при этом освободится несколько мегабайтов свободного места — просто потому, что какая-нибудь «хитрая» программа или технология назначила ему поток дополнительных (альтернативных) данных такого большого размера. Однако на самом деле опасаться

[1] Не путать с *потоками выполнения* (threads).

подобных ситуаций не следует (хотя гипотетически они возможны), поскольку пока механизм потоков данных в полной мере не используются. Просто необходимо иметь в виду, что файл в системе NTFS — это более глубокое и глобальное понятие, чем мы себе представляем.

Стандартные атрибуты файлов и каталогов на томе NTFS имеют фиксированные имена и коды типа (табл. 6.7).

Таблица 6.7. Атрибуты файлов в системе NTFS

Системный атрибут	Описание атрибута
Стандартная информация о файле	Традиционные атрибуты («только для чтения», «скрытый», «архивный», «системный»), отметки времени, включая время создания или последней модификации, число каталогов, ссылающихся на файл
Список атрибутов	Список атрибутов файла и файловая ссылка на запись в таблице MFT, в которой расположен каждый из атрибутов. Файловая ссылка используется, если файлу необходимо более одной записи в MFT
Имя файла	Имя файла в кодировке Unicode. Файл может иметь несколько имен, подобно тому как это имеет место в UNIX. Это случается, когда имеется связь POSIX к данному файлу или если у файла есть автоматически сгенерированное имя в формате 8.3
Дескриптор защиты	Структура данных, соответствующая списку управления доступом (ACL) и предохраняющая файл от несанкционированного доступа. Дескриптор защиты определяет, кто владелец файла и кто имеет те или иные разрешения доступа к нему
Данные	Собственно данные файла, его содержимое. В NTFS у файла по умолчанию есть один безымянный атрибут данных и могут быть дополнительные именованные атрибуты данных. У каталога нет атрибута данных по умолчанию, но он может иметь необязательные именованные атрибуты данных
Корень индекса, размещение индекса, битовая карта (только для каталогов)	Атрибуты, используемые для индексов имен файлов в больших каталогах
Расширенные атрибуты HPFS	Атрибуты, используемые для реализации расширенных атрибутов HPFS для подсистемы OS/2, а также OS/2-клиентов файл-серверов Windows NT

Разрешения NTFS

Разрешения NTFS (NTFS permissions) — это набор специальных расширенных атрибутов файла или каталога (папки), заданных для ограничения доступа пользователей к этим объектам. Они имеются только на томах, где установлена файловая система NTFS. Разрешения обеспечивают гибкую защиту, так как их можно применять и к каталогам, и к отдельным файлам; они распространяются как на локальных пользователей (работающих на компьютерах, где находятся защищенные папки и файлы), так и на пользователей, подключающихся к ресурсам по сети.

Не следует путать *разрешения* с *правами*. Это совершенно разные понятия; подробнее об этом написано в подразделе «Модель безопасности Windows NT/2000/XP». К сожалению, в технической литературе да и в обиходе часто путают эти термины. Истоком этого прежде всего являются ошибки перевода оригинальных англоязычных материалов.

Разрешения NTFS служат, прежде всего, для защиты ресурсов от локальных пользователей, работающих за компьютером, на котором располагается ресурс. Однако их можно использовать и для удаленных пользователей, подключающихся к общей папке по сети. Очевидно, что в этом случае на пользователей действуют два механизма ограничения в доступе к ресурсам: сначала сетевой, а уже затем локальный, файловый. Поэтому итоговые разрешения на доступ будут определяться как минимальные из сетевых и файловых разрешений. Здесь необходимо сказать, что итоговые сетевые разрешения на доступ к ресурсам, которыми будет обладать пользователь при работе в сети, вычисляются как максимум разрешений в списке разрешений доступа, поскольку пользователь может быть членом нескольких групп, которые упомянуты в списке. Аналогично и для разрешений NTFS: пользователь получает максимальные разрешения, перечисленные в списке управления доступом, и только разрешение No Access (нет доступа)[1] может перечеркнуть все остальные разрешения.

Разрешения NTFS обеспечивают высокую избирательность защиты: для каждого файла в папке можно установить свои разрешения. Например, одному пользователю можно позволить считывать и изменять содержимое файла, другому только считывать, третьему вообще запретить доступ. Заметим, однако, что настоятельно рекомендуется устанавливать разрешения в списках ACL, используя не учетные записи отдельных пользователей, а учетные записи групп пользователей.

Итак, каждый файловый объект имеет свой список управления доступом. Этот список имеет приоритет над списком управления доступом того каталога, в котором находится файловый объект. Подобно корневому каталогу файл-каталог, в отличие от простого файла, является объектом контейнерного типа, то есть он может содержать другие файловые объекты. При создании нового файлового объекта он наследует разрешения NTFS. Поэтому при копировании файловых объектов они получают разрешения доступа, совпадающие с родительскими. Однако при перемещении файлов и каталогов в пределах одного диска списки управления доступом не меняются. Объясняется этот факт просто. Списки управления доступом являются одним из потоков данных файлового объекта, и доступ к ним осуществляется через элемент каталога. Поэтому изменение информации о местонахождении файла никак не должно влиять ни на один из потоков данных файла. Если же файлы (и подкаталоги) переместить с одного диска с NTFS на другой, то на новом диске создаются новые элементы каталогов и они должны унаследовать разрешения доступа того контейнера, в котором они создаются. Это очень важное обстоятельство, и при работе с разрешениями NTFS не следует о нем забывать.

---

[1] В Windows 2000/XP вместо стандартного разрешения No Access устанавливается запрет (deny) на соответствующее разрешение.

Мы уже упоминали про списки ACL. Они могут быть у многих объектов. В NTFS у каждого файлового объекта на самом деле имеется два списка. Первый называется *DACL* (Discretionary ACL — *дискреционный список управления доступом*). Именно этот список описывает ограничения на доступ к файловому объекту, перечисляя группы и пользователей и указывая те операции, которые разрешены и запрещены. Этот список может изменить любой пользователь, имеющий разрешение на изменение разрешений (change permissions) для данного файлового объекта. Такое разрешение обычно обозначается буквой P (от permissions — разрешения).

Второй список называется *SACL* (System ACL — системный список управления доступом). Этот список предназначен для аудита, и его могут составлять и редактировать только администраторы системы. Изначально списки SACL пусты, но их можно сформировать. В зависимости от того, успех или отказ в той или иной операции над файловым объектом необходимо проконтролировать, администратор формирует список SACL. Элементами такого списка являются записи типа:

```
SID – разрешение – успех/отказ
```

Здесь аббревиатура SID означает Security Identifier (идентификатор безопасности). Напомним, что во многих операционных системах для аутентификации и авторизации пользователей используются учетные записи (см. главы 1 и 11). Учетные записи бывают групповыми и пользовательскими. В системах класса Windows NT (2000/XP) каждой учетной записи поставлен в однозначное соответствие ее идентификатор (в данном случае — SID).

Обрабатываться элементы списка SACL будут только в том случае, если в системе включен аудит на доступ к файловым объектам. Если в системе разрешен аудит файловых операций, то операционная система при их выполнении сравнивает записи в SACL с запросом и с записями в списке DACL и фиксирует в журнале безопасности соответствующие события.

Нас, прежде всего, должны интересовать списки DACL, которые и определяют разрешения на доступ к файлам и каталогам.

Каждый файловый объект имеет так называемую *маску доступа* (access mask). Маска доступа включает *стандартные* (standard), *специфичные* (specific) и *родовые* (generic) права доступа. Мы называем их здесь правами, чтобы отличать от тех разрешений, которые перечисляются в пользовательском интерфейсе.

- Стандартные права доступа определяют операции, которые являются общими для всех защищенных объектов. Право Read_Control позволяет прочитать информацию из дескриптора безопасности объекта. Право Write_DAC дает возможность изменить дискреционный список прав доступа. Право Write_Owner позволяет записать (изменить) владельца объекта. Право Synchronize дает возможность использовать объект для синхронизации. Наконец, есть право Delete, которое позволяет удалить объект.
- Специфичные права доступа указывают основные права, характерные для файловых объектов. Так, например, специфичные права Read_Data, Write_Data и Append_Data позволяют прочитать данные, записать информацию и, соответственно, добавить данные к файлу. Права Read_Attributes, Write_Attributes

и Read_EA, Write_EA позволяют, соответственно, прочитать или записать атрибуты или расширенные атрибуты файла или каталога. Наконец, такое специфичное право доступа, как Execute, позволяет запустить файл на выполнение.

- Родовые права доступа используются системой; они определяют комбинации стандартных и специфичных прав. Например, родовое право доступа Generic_Read, примененное к файлу, включает в себя следующие специфичные и стандартные права: Read_Control, File_Read_Data, File_Read_Attributes, File_Read_EA, Synchronize.

На основе рассмотренных выше прав доступа, которые используются при программировании, для пользователей, работающих с файлами, создан механизм разрешений. Дело в том, что управлять доступом пользователей к файлам и каталогам на основе маски доступа, то есть путем указания соответствующих битов, неудобно. Поэтому для практического администрирования применяются разрешения NTFS, которые позволяют скрыть от пользователя низкоуровневый механизм прав доступа.

Поскольку операционные системы Windows 2000/XP нынче становятся основными для персональных компьютеров, а в дисциплинах учебного плана многие очень важные вопросы, касающиеся практической работы в этих системах, к сожалению, не изучаются, мы изложим не только основные теоретические вопросы работы с разрешениями NTFS, но и осветим некоторые детали интерфейса.

Итак, разрешения NTFS по-разному представлены в операционных системах Windows NT 4.0 и семействе систем Windows 2000/XP. Отличия эти, прежде всего, касаются интерфейса, то есть программа Проводник (Explorer) по-разному отображает те разрешения, которые на самом деле присвоены файловому объекту в виде разрешений доступа и обрабатываются на программном уровне. Разрешения в Windows 2000/XP ближе к тем специфичным, стандартным и родовым правам доступа, о которых мы говорили выше, однако для управления доступом к файлам они не так удобны, как разрешения Windows NT 4.0.

Для начала рассмотрим механизм разрешений NTFS для систем Windows NT 4.0. Во многих отношениях он является более простым и, соответственно, более понятным.

Разрешения NTFS в Windows NT 4.0

В NTFS для Windows NT 4.0 разрешения на доступ к файлам и каталогам бывают *индивидуальными*, *стандартными* и *специальными*.

Индивидуальные разрешения. Под индивидуальными разрешениями понимают набор прав, позволяющий предоставлять пользователю доступ того или иного типа. В Windows NT 4.0 этих разрешений всего шесть: Read (чтение), Write (запись), Execute (выполнение), Delete (удаление), Change Permissions (смена разрешений) и Take Ownership (смена владельца). В табл. 6.8 описаны разрешенные пользователю операции с файлом или каталогом при предоставлении одного из индивидуальных разрешений NTFS на файловый объект.

Таблица 6.8. Индивидуальные разрешения на файлы и каталоги

Индивидуальное разрешение NTFS	Разрешенные операции с каталогом	Разрешенные операции с файлом
Чтение (R — Read)	Просмотр имен каталога, файлов в нем, разрешений на доступ к нему, атрибутов каталога и сведений о его владельце	Просмотр содержимого файла, разрешений на доступ к нему, его атрибутов и сведений о его владельце
Запись (W — Write)	Добавление в каталог файлов и папок; изменение атрибутов каталога; просмотр атрибутов каталога, сведений о владельце и разрешений на доступ к нему	Просмотр разрешений на доступ к файлу и сведений о владельце; изменение атрибутов файла; изменение и добавление данных файла
Выполнение (X — eXecute)	Просмотр атрибутов каталога; изменения во вложенных папках; просмотр разрешений на доступ к каталогу и сведений о его владельце	Просмотр разрешений на доступ к файлу, его атрибутов и сведений о его владельце; запуск файла (если он является исполняемым)
Удаление (D — Delete)	Удаление каталога	Удаление файла
Смена разрешений (P — change Permissions)	Изменение разрешений на доступ к каталогу	Изменение разрешений на доступ к файлу
Смена владельца (O — take Ownership)	Назначение себя владельцем каталога	Назначение себя владельцем файла

Индивидуальные разрешения по отдельности дают весьма ограниченные возможности на доступ к файлам и каталогам и управление ими в разделах NTFS. Обычно же для выполнения над файлами или папками действий определенного уровня требуются наборы индивидуальных разрешений. Такие наборы в файловой системе NTFS называются *стандартными разрешениями*. Именно они доступны в списке Type of Access (Тип доступа) диалоговых окон File Permissions и Directory Permissions программы Explorer (Проводник) в Windows NT.

Стандартные разрешения. Для того чтобы не использовать каждый раз сочетания индивидуальных разрешений, введены так называемые стандартные разрешения NTFS, которыми все и пользуются в большинстве случаев. Они представляют собой наиболее применяемые (с точки зрения разработчиков Microsoft) комбинации индивидуальных разрешений. Одновременное назначение нескольких индивидуальных разрешений для файла или каталога значительно упрощает администрирование.

Приведем таблицу стандартных разрешений на файлы и каталоги системе NTFS 4 (табл. 6.9). В таблице перечислены стандартные разрешения для папок и указаны соответствующие им индивидуальные разрешения NTFS.

Теперь поясним эти разрешения. Далее первыми в скобках записаны разрешения на каталог, вторыми — разрешения на файлы в этом каталоге.

- List (RX, разрешения не указаны) — просмотр. Пользователь может только просмотреть содержимое папки (список файлов и вложенных папок) и перей-

ти во вложенную папку, но не может получить доступ к новым файлам, созданным в этой папке.

- Add (WX, разрешения не указаны) — добавление. Пользователь может создать в папке новые файлы и вложенные папки, но не может просмотреть ее текущее содержимое.
- Add & Read (RWX, RX) — чтение и запись. Пользователь может создавать в папке новые файлы или вложенные папки, читать содержимое самой папки и содержащихся в ней файлов и вложенных папок, а также запускать приложения, которые находятся в этой папке, но не может изменить содержимое файлов в этой папке.
- Change (RWXD, RWXD) — изменение. Пользователь может прочитать, создавать и удалять файлы и вложенные папки, а также запускать находящиеся в этой папке приложения.
- Full Control (все разрешения, все разрешения) — полный доступ. Пользователь может читать, создавать и изменять файлы и вложенные папки, изменять разрешения на папку и файлы внутри нее, а также стать владельцем папки и содержащихся в ней файлов.

Таблица 6.9. Стандартные разрешения на файлы и каталоги

Стандартные разрешения		Комбинации индивидуальных разрешений	
		Каталоги	Файлы
No Access	Нет доступа	Нет разрешений	Нет разрешений
List	Просмотр	(RX)	Не указано
Read	Чтение	(RX)	(RX)
Add	Добавление	(WX)	Не указано
Add & Read	Чтение и запись	(RWX)	(RX)
Change	Изменение	(RWXD)	(RWXD)
Full Control	Полный доступ	Все разрешения	Все разрешения

Разрешение No Access (нет доступа) является самым сильным в том плане, что оно запрещает любой доступ к файлу или папке, даже если пользователь является членом группы, которой дано разрешение на доступ. Стандартное разрешение No Access устанавливается, когда снимают все индивидуальные разрешения NTFS. Имейте в виду: оно обозначает не отсутствие разрешений, а явный запрет на доступ и отменяет для пользователя все разрешения, установленные в остальных строках списка управления доступом.

Разрешения Full Control (полный доступ) и Change (изменение) отличаются тем, что второе не позволяет менять разрешения и владельца объекта, то есть среди составляющих его индивидуальных разрешений отсутствуют разрешения на смену разрешений (P) и смену владельца (O).

Специальные разрешения. И наконец, специальные разрешения. Это комбинации индивидуальных разрешений R, W, X, D, P и O, не совпадающие ни с одним из разреше-

ний стандартного набора. Установить специальные разрешения NTSF в диалоговом окне разрешений (File Permissions или Directory Permissions) можно только правкой существующих разрешений для пользователя или группы. Иными словами, чтобы установить для кого-нибудь специальное разрешение NTFS, вам придется установить сначала какое-либо из стандартных разрешений и лишь затем преобразовать его в специальное. При этом для папок можно отдельно регулировать доступ как к самой папке (Special Directory Access), так и к находящимся в ней файлам (Special File Access). Таким образом, удается весьма дифференцированно управлять доступом пользователей к файлам и папкам на томах с файловой системой NTFS.

Применение разрешений NTFS

Разрешения NTFS присваиваются учетным записям пользователей и групп так же, как и разрешения доступа к общим сетевым ресурсам. Пользователь может получить разрешение либо непосредственно, либо являясь членом одной или нескольких групп, имеющих разрешение.

Применение разрешений NTFS для каталогов сходно с применением разрешений доступа к общим ресурсам. Управление разрешениями на каталог или файл осуществляется, как правило, через Проводник. Для этого необходимо щелкнуть на объекте правой кнопкой мыши, выбрать в контекстном меню команду Properties (Свойства) и в открывшемся окне перейти на вкладку Security (Безопасность). На этой вкладке имеется три раздела. Первый (верхний) с кнопкой Permissions (Разрешения) как раз и позволяет просмотреть и/или изменить разрешения, то есть управлять списком DACL. Второй (средний) с кнопкой Audit (Аудит) предназначен для управления списком SACL. Наконец, последний (нижний) раздел с кнопкой Owner (Владелец) предназначен для просмотра и/или смены владельца файлового объекта.

Кроме того, имеется возможность устанавливать и/или изменять списки разрешений NTFS через интерфейс командной строки. Для этого используется следующая команда:

```
CACLS ИмяФайла [/T] [/E] [/C] [/G ИмяПользователя:доступ] [/R ИмяПользователя [...]] [/P
ИмяПользователя:доступ [...]] [/D ИмяПользователя [...]]
```

Здесь:

- **ИмяФайла** — имя файла со списком управления доступом;
- /T — замена списка управления доступом для указанных файлов в текущем каталоге и всех подкаталогах;
- /E — изменение списка управления доступом вместо его замены;
- /C — продолжение выполнения при ошибках отказа в доступе;
- /G **ИмяПользователя:доступ** — определение разрешений для указанных пользователей, где параметр **доступ** равен:
  - R — чтение,
  - C — изменение (запись),
  - F — полный доступ;
- /R **ИмяПользователя** — отзыв разрешений для пользователя (только вместе с ключом /E);

- /P ИмяПользователя:доступ — замена разрешений для указанного пользователя, где параметр доступ равен:
  - N — отсутствует,
  - R — чтение,
  - C — изменение (запись),
  - F — полный доступ;
- /D ИмяПользователя — запрет на доступ для указанного пользователя.

Для выбора нескольких файлов используются подстановочные знаки. В команде можно указать несколько пользователей.

У каждого файлового объекта имеется его владелец и создатель. Пользователь, создавший файл или папку на томе NTFS, становится владельцем этого файла или папки. Владелец всегда имеет право назначать и изменять разрешения на доступ к своему файлу или папке, даже если у него нет соответсвующего разрешения. Если этот пользователь является членом группы Administrators (Администраторы), фактическим владельцем становится вся группа Administrators.

Изначально все пользователи имеют все разрешения на файлы и каталоги. Очевидно, что при этом они могут изменять эти разрешения, которые оформляются в виде списка. Напомним, что такой список называют списком ACL, хотя на самом деле, как уже упоминалось, речь идет о списке DACL. Список DACL состоит из записей ACE (Access Control Entry — запись списка управления доступом); в каждой из них указывается идентификатор безопасности (SID)[1] и соответствующая ему маска доступа, которая строится на основе заданных пользователем разрешений. Другими словами, каждому идентификатору безопасности ставится в соответствие перечень индивидуальных разрешений. Например, список на некий каталог может выглядеть следующим образом:

- Everyone — List;
- Engineers — Add & Read;
- Managers — Change;
- Administrators — Full Control.

Этот список следует понимать так: все имеют разрешение на просмотр содержимого данного каталога, члены группы Engineers имеют разрешение на чтение содержимого каталога и запись в него новых файлов, члены группы Managers могут изменять свободно каталог и его содержимое, а члены группы Administrators имеют все разрешения.

В отличие от разрешений доступа к общим (сетевым) ресурсам, разрешения NTFS защищают локальные ресурсы. В частности, файлы и папки, содержащиеся в данном каталоге, могут иметь иные разрешения, нежели он сам.

---

[1] Это уникальная 128-разрядная кодовая запись, на основании которой операционная система может идентифицировать пользователей. Именно SID сопровождает все запросы к операционной системе на получение того или иного ресурса, в результате чего она может вычислить и разрешения, и права пользователей на запрашиваемый ресурс.

Напомним, что в системе Windows NT 4.0 разрешения NTFS для файла превалируют над разрешениями для каталога, в котором он содержится. Например, если пользователь имеет разрешения Read (чтение) для каталога и Write (запись) для вложенного в него файла, то он сможет записать данные в файл, но не сможет создать новый файл в этом каталоге.

Как и разрешения доступа к общим (сетевым) ресурсам, фактические разрешения NTFS для пользователя — это комбинация разрешений пользователя и групп, членом которых он является. Единственное исключение — разрешение No Access (нет доступа): оно отменяет все остальные разрешения.

При указании разрешений в соответствующем окне, в которое мы попадаем после выбора в контекстном меню команды Properties (Свойства), перехода на вкладку Security (Безопасность) и щелчка на кнопке Permissions (Разрешения), следует обратить внимание на информацию, указанную в списке Type of Access (Тип доступа) в скобках рядом с типом разрешения. В первой паре скобок представлены индивидуальные разрешения на доступ к самой папке, во второй — на доступ к файлам, создаваемым в этой папке. Некоторые разрешения для папки не меняют разрешений для файлов (Not Specified). При этом пользователь не сможет обращаться к файлам в этой папке, если только разрешения на доступ для него не заданы как-нибудь иначе (например, через разрешения, устанавливаемые на отдельные файлы).

Если при форматировании тома на него устанавливается файловая система NTFS, группе Everyone (все) автоматически присваивается разрешение Full Control (полный доступ) на этот том. Папки и файлы, создаваемые на этом томе, по умолчанию наследуют это разрешение.

Разрешения, установленные для пользователя, складываются (аккумулируются) с разрешениями, установленными для групп, к которым он принадлежит. Например, если на доступ к какому-либо файлу для пользователя установлено разрешение Read, а для группы Everyone — разрешение Change, пользователь сможет изменить содержимое файла или удалить его, поскольку любой пользователь всегда входит в эту группу. Из этого правила есть исключение, когда одним из установленных разрешений доступа является разрешение No Access. При этом не важно, кому именно это разрешение предоставлено, пользователю или группе. Разрешение No Access всегда имеет приоритет, поэтому пользователь *не сможет* получить доступ к файлу или папке.

Пользователи, имеющие разрешение Full Control на папку, могут удалять файлы в этой папке *независимо* от разрешений, установленных на файл (даже если разрешением на файл является разрешение No Access). Это следствие того, что система NTFS удовлетворяет стандарту POSIX.1. Чтобы решить проблему (если полный набор разрешений доступа к папке действительно необходим), надо установить для папки специальный тип доступа, включающий все индивидуальные разрешения R, W, X, D, P и O. При этом пользователи получают тот же набор разрешенных действий, что и при разрешении Full Control, но теряют возможность несанкционированного удаления файлов в этой папке.

Также важно отметить, что, имея одно только разрешение Change Permission, позволяющее изменять разрешения, пользователь может установить любые разрешения на доступ к файлу или папке.

Пользователь, создающий папку или файл на разделах с файловой системой NTFS, становится владельцем созданного объекта. Кроме того, владельцем папки или файла может стать любой пользователь, обладающий стандартным разрешением Full Control или специальным разрешением Take Ownership. Владелец *всегда* имеет возможность прочитать информацию о разрешениях на доступ к папке или файлу и *изменить* их, даже если ему ничего не разрешено или ему предоставлено разрешение No Access. Отсюда следует, что достаточно дать пользователю разрешение Take Ownership и он, в конечном счете, сможет получить доступ к файлу или папке на разделе NTFS.

Разрешения NTFS в Windows 2000/XP

В семействе операционных систем Windows 2000 и Windows XP были существенно переработаны и сама система управления файлами, получившая название NTFS5, и интерфейс, посредством которого можно управлять разрешениями NTFS. Вместо описанных выше индивидуальных, стандартных и специальных разрешений Windows NT 4.0 теперь в пользовательском интерфейсе имеется перечень из 13 разрешений, которые можно (по аналогии с предыдущей системой) назвать индивидуальными, хотя Microsoft более этот термин не употребляет[1] и называет их специальными разрешениями. Опишем кратко эти индивидуальные (специальные) разрешения.

- Traverse Folder/Execute File (Обзор папок/Выполнение файлов):
  - Traverse Folder — разрешает (или запрещает) перемещение по папке в поисках файлов или вложенных папок, даже если пользователь не обладает разрешением на доступ к просматриваемой папке (это разрешение применимо только к папкам и только если группа или пользователь не обладает правом перекрестной проверки, а по умолчанию группа Everyone наделена правом перекрестной проверки);
  - Execute File — разрешает (или запрещает) запуск программ (применимо только к файлам).
- List Folder/Read Data (Содержание папки/Чтение данных):
  - List Folder — разрешает (или запрещает) просмотр имен файлов и вложенных папок внутри папки (применимо только к папкам);
  - Read Data — разрешает (или запрещает) чтение данных из файлов (применимо только к файлам).
- Read Attributes (Чтение атрибутов). Разрешает (или запрещает) просмотр атрибутов файла или папки, таких как «только для чтения» или «скрытый». Атрибуты определяются файловой системой NTFS.
- Read Extended Attributes (Чтение дополнительных атрибутов). Разрешает (или запрещает) просмотр дополнительных атрибутов файла или папки. Дополнительные атрибуты определяются программами и зависят от них. Атрибуты сжатия файлов NTFS и шифрования относятся к дополнительным.

[1] В целях преемственности терминологии и удобства изложения материала мы тем не менее будем использовать этот термин.

- Create Files/Write Data (Создание файлов/Запись данных):
  - Create Files — разрешает (или запрещает) создание файлов в папке (применимо только к папкам);
  - Write Data — разрешает (или запрещает) внесение изменений в файл и замену имеющегося содержимого (применимо только к файлам).
- Create Folders/Append Data (Создание папок/Дозапись данных):
  - Create Folders — разрешает (или запрещает) создание папок в папке (применимо только к папкам);
  - Append Data — разрешает (или запрещает) внесение изменений в конец файла, но не изменение, удаление и замену имеющихся данных (применимо только к файлам).
- Write Attributes (Запись атрибутов). Разрешает (или запрещает) смену атрибутов файла или папки, таких как «только для чтения» или «скрытый». Атрибуты определяются файловой системой NTFS.
- Write Extended Attributes (Запись дополнительных атрибутов). Разрешает (или запрещает) смену дополнительных атрибутов файла или папки. Дополнительные атрибуты определяются программами и зависят от них.
- Delete Subfolders and Files (Удаление подпапок и файлов). Разрешает (или запрещает) удаление вложенных папок и файлов даже при отсутствии разрешения Delete.
- Delete (Удаление). Разрешает (или запрещает) удаление файла или папки. При отсутствии этого разрешения требуемый объект (файл или папку) все же можно удалить при наличии разрешения Delete Subfolders and Files для родительской папки.
- Read Permissions (Чтение разрешений). Разрешает или запрещает чтение разрешений на доступ к файлу или папке, таких как Full Control, Read и Write.
- Change Permissions (Смена разрешений). Разрешает или запрещает чтение разрешений на доступ к файлу или папке, таких как Full Control, Read и Write.
- Take Ownership (Смена владельца). Разрешает или запрещает возможность стать владельцем файла или папки. Владелец файла или папки всегда может изменить разрешения на доступ к ним независимо от любых разрешений, защищающих файл или папку.

Из перечисленных выше «индивидуальных» разрешений формируются так называемые *основные* разрешения. Они являются аналогом *стандартных* разрешений в NTFS4. Принципиальное отличие между стандартными и индивидуальными разрешениями в NTFS4 и NTFS5 заключается в том, что теперь имеется 6 *основных* разрешений на каталог и 5 *основных* разрешений на файл. Причем каждое из этих разрешений может быть в явном виде разрешено или запрещено. То есть каждое *основное* разрешение пользователь может разрешить (allow) или запретить (deny). Если разрешение не отмечено как разрешенное или запрещенное, то считается, что оно *не запрещено*. Таким образом, конкретное разрешение может быть задано тремя способами: *разрешено* (в явном виде), *не запрещено*, *запрещено*. Напомним, что итоговые разрешения для конкретного пользователя вычисляются как сумма всех

разрешений по записям ACE, образующим список DACL. Например, если в списке DACL у пользователя имеется разрешение на запись, а членам группы, в которую он входит, присвоено разрешение на чтение, то этот пользователь будет иметь итоговое разрешение и на чтение, и на запись.

Запрет имеет большую силу, нежели явное разрешение. Другими словами, если встречается ACE с явным запретом на некоторое разрешение для конкретного пользователя или группы, в которую он входит, то это разрешение всегда будет запрещено для данного пользователя и его группы, даже если в остальных записях данное разрешение будет помечено как разрешенное.

Если вас не устраивают *основные* разрешения, то можно сформировать *специальные* разрешения как конкретную комбинацию «индивидуальных» разрешений. Для этого необходимо щелкнуть на кнопке Advanced (Дополнительно). При этом открывается окно управления разрешениями, в котором они перечислены. В этом окне есть кнопки Add (Добавить), Change (Изменить) и Delete (удалить), которые позволяют добавлять, изменять или удалять выбранные разрешения.

Если в окне свойств безопасности объекта флажки затенены, значит, разрешения на доступ к данному объекту унаследованы от родительского объекта. Существуют три способа изменения унаследованных разрешений.

- Внесите в разрешения на доступ к родительскому объекту изменения, которые будут унаследованы данным объектом.
- Явно разрешите (если оно было помечено как запрещенное) или запретите (если оно было помечено как разрешенное) данное унаследованное разрешение.
- Снимите флажок Inherit from parent the permission entries that apply to child objects. Include these with entries explicity defined here (Переносить наследуемые от родительского объекта разрешения на этот объект). В появившемся диалоговом окне будет предложено выбрать одну из трех альтернатив: скопировать разрешения родительского объекта (к ним можно будет впоследствии добавить новые), удалить разрешения и сформировать их заново или ничего не трогать и вернуться к исходному состоянию разрешений. После снятия флажка можно изменять список разрешений: изменять сами разрешения, удалять пользователей или группы из списка разрешений, поскольку данный объект больше не будет наследовать разрешения на доступ к родительскому объекту.

Разрешения на доступ к файловым объектам должны быть максимально строгими. Это снизит вероятность случайного удаления или изменения важной информации. Рекомендуется всегда, когда возможно, назначать разрешения для групп, а не для отдельных пользователей. Другими словами, следует создавать группы безопасности, исходя из требований уровня доступа к файлам, и именно этим группам предоставлять необходимые разрешения. Отдельным пользователям следует предоставлять разрешения на доступ только в исключительных случаях, когда это действительно требуется.

При назначении разрешений для папок, в которых расположены приложения или данные справочного характера, то есть практически неизменяемые при рядовой работе пользователей, следует заменить стандартное разрешение Full Contorol (пол-

ный доступ) для группы Everyone (все) на разрешение Read & Execute (чтение и выполнение). Это позволит предотвратить случайное удаление файлов или заражение их вирусами. Тем пользователям, которые ответственны за обновление хранящихся в папке файлов, можно дать разрешения Change (изменение), Read & Execute (чтение и выполнение), Read (чтение) и Write (запись). Имея их, они смогут выполнять порученную им работу, но не смогут изменять разрешения. Право на изменение разрешений следует оставлять за членами группы Администраторы.

В качестве примера управления разрешениями NTFS при работе в Windows 2000/XP рассмотрим следующую несложную задачу. Пусть требуется создать папку Контрольные работы, в которой члены группы Студенты должны иметь возможность размещать свои файлы и при необходимости даже исправлять их, но чтобы они не имели возможности читать чужие контрольные работы и удалять файлы. Для группы Преподаватели должно быть разрешение на чтение этих файлов. Администраторы должны иметь разрешение Full Control (Полный доступ), чтобы иметь возможность управлять разрешениями и удалять старые ненужные файлы и папки. Последовательность действий, которые нужно выполнить для решения этой задачи, может быть следующей.

1. Создаем папку Контрольные работы. Переходим на вкладку Security (Безопасность) в окне Properties (Свойства папки).
2. Снимаем в левом нижнем углу этого окна флажок Inherit from parent the permission entries that apply to child objects (Переносить наследуемые от родительского объекта разрешения на этот объект) и копируем разрешения родительского каталога.
3. Щелкаем на кнопке Add (Добавить), в открывшемся окне находим группу Администраторы, щелкаем на кнопке Add (Добавить), после чего щелкаем на кнопке OK в окне добавления. В окне Security (Безопасность) для каждой новой учетной записи по умолчанию устанавливается разрешение Read & Execute (Чтение и выполнение), которое предполагает наличие разрешений List (Список содержимого папки) и Read (Чтение).
4. Устанавливаем для группы Администраторы разрешение Full Control (Полный доступ), для чего достаточно установить соответствующий флажок. Флажки для остальных разрешений установятся автоматически.
5. Добавляем группу Преподаватели. В окне безопасности для них автоматически устанавливается разрешение Read & Execute (Чтение и выполнение), что нас вполне устраивает.
6. Добавляем группу Студенты. В окне безопасности снимаем флажки для разрешений Read & Execute (Чтение и выполнение) и Read (Чтение), оставив разрешение на получение списка содержимого папки.
7. Поскольку члены группы Студенты должны иметь возможность поместить в папку Контрольные работы свои файлы, в окне безопасности устанавливаем флажок для разрешения Write (Запись).
8. Чтобы студенты могли читать и исправлять только свои файлы, добавляем специальную учетную запись СОЗДАТЕЛЬ-ВЛАДЕЛЕЦ. Поля с разрешениями для нее окажутся пустыми, однако это не должно нас смущать. Если щелкнуть на кноп-

ке Advanced (Дополнительно), то в открывшемся окне Advanced security settings for Контрольные работы (Параметры управления доступом для Контрольные работы) мы увидим, что для учетной записи СОЗДАТЕЛЬ-ВЛАДЕЛЕЦ имеется разрешение Full Control (Полный доступ).

9. Для того чтобы запретить студентам удалять файлы (и папки) в папке Контрольные работы, необходимо в окне Advanced security settings for Контрольные работы (Параметры управления доступом для Контрольные работы) выделить группу Студенты. Далее, щелкнув на кнопке Edit (Показать/Изменить), в открывшемся окне специальных разрешений установить флажок в столбце Deny (Запретить) для разрешений, связанных с удалением.

По умолчанию выставленные нами разрешения будут действовать для этой папки, ее вложенных папок и файлов. Если бы нас не устраивало такое положение вещей, то, щелкнув на кнопке Edit (Показать/Изменить), в открывшемся окне Permission Entry for Контрольные работы (Элемент разрешения для Контрольные работы) можно было бы с помощью переключателей Apply onto (Применять) указать, к каким объектам должны относиться установленные разрешения.

В качестве дополнительной рекомендации можно посоветовать при просмотре существующих разрешений NTFS на папки не закрывать окно безопасности щелчком на кнопке OK и не щелкать без необходимости на кнопке Apply (Применить), поскольку в этом случае с достаточно большой вероятностью будут изменены существующие разрешения на файлы и вложенные папки. После просмотра разрешений, если ничего не нужно менять, следует щелкнуть на кнопке Cancel (Отмена).

Контрольные вопросы и задачи

Вопросы для проверки

1. Что такое «файловая система»? Что дает использование той или иной файловой системы? Какие файловые системы применяются на персональных компьютерах?
2. Объясните общие принципы устройства файловой системы FAT. Что представляет собой таблица FAT? Что такое кластер, от чего зависит его размер?
3. Сравните файловые системы FAT16 и FAT32. В чем их достоинства и недостатки?
4. Изложите основные принципы работы системы HPFS. За счет чего в файловой системе HPFS обеспечена высокая производительность?
5. Что означает протоколирование файловых операций? Что оно дает?
6. Перечислите основные возможности файловой системы NTFS. Объясните понятие потока данных в NTFS.
7. Расскажите о правилах, которые определяют состояние разрешений на доступ при перемещении и копировании файловых объектов на томах с файловой системой NTFS.

8. Что такое стандартные, индивидуальные и специальные разрешения на доступ? Перечислите их и постройте таблицы соответствия стандартных и индивидуальных разрешений для NTFS4.
9. Постройте таблицы соответствия стандартных и индивидуальных разрешений для NTFS5. Не забудьте, что индивидуальные разрешения в Windows 2000/XP стали называть специальными.

Задания

1. Используя персональный компьютер с установленной на нем ОС Windows NT или Windows 2000/XP, проверьте правила, которые определяют состояние разрешений доступа при перемещении или копировании объектов при использовании NTFS. Расскажите о полученных результатах.
2. Создайте папку с двумя программами (для простоты можно взять Блокнот и Калькулятор) и обеспечьте, чтобы можно было запускать эти программы, но нельзя было бы их скопировать, переместить, удалить.
3. Создайте папку Examen, в которую пользователи — члены группы Students — могли бы записать результаты своей работы, но не смогли бы прочитать чужую работу и, соответственно, исправить ошибки в своей.

Глава 7. Организация параллельных взаимодействующих вычислений

Мультипрограммные и мультизадачные операционные системы позволяют организовать не только независимые, но взаимодействующие вычисления. Сама операционная система как комплекс управляющих и обрабатывающих программных модулей также функционирует как множество взаимодействующих вычислений. Проблема синхронизации взаимодействия параллельных вычислительных процессов, обмена данными между ними является одной из важнейших. Существующие методы синхронизации вычислений и обмена сообщениями различаются по таким параметрам, как удобство программирования параллельных процессов, стоимость реализации, эффективность функционирования созданных приложений и всей вычислительной системы в целом. Операционные системы имеют в своем составе различные средства синхронизации. Знание этих средств и их правильное использование позволяет создавать программы, которые при работе осуществляют корректный обмен информацией, а также исключают возможность возникновения тупиковых ситуаций.

В этой главе рассматриваются основные понятия и проблемы, характерные для параллельных процессов. Описываются основные механизмы синхронизации, дается их сравнительный анализ, приводятся характерные примеры программ, использующих данные механизмы.

Независимые и взаимодействующие вычислительные процессы

Основной особенностью мультипрограммных операционных систем является то, что в их среде параллельно развивается несколько (последовательных) вычислительных процессов. С точки зрения внешнего наблюдателя эти последовательные

вычислительные процессы выполняются одновременно, мы же будем говорить «параллельно». При этом под *параллельными* понимаются не только процессы, одновременно развивающиеся на различных процессорах, каналах и устройствах ввода-вывода, но и те последовательные процессы, которые разделяют центральный процессор и в своем выполнении хотя бы частично перекрываются во времени. Любая мультизадачная операционная система вместе с параллельно выполняющимися в ней задачами может быть логически представлена как совокупность последовательных вычислений, которые, с одной стороны, состязаются за ресурсы, переходя из одного состояния в другое, а с другой — действуют почти независимо один от другого, но при этом образуя единую систему посредством установления разного рода связей между собой (путем пересылки сообщений и синхронизирующих сигналов).

Итак, *параллельными* мы будем называть такие последовательные вычислительные процессы, которые одновременно находятся в каком-нибудь активном состоянии. Два параллельных процесса могут быть *независимыми* (independed processes) либо *взаимодействующими* (cooperating processes).

Независимыми являются процессы, множества переменных которых не пересекаются. Под переменными в этом случае понимают файлы данных, а также области оперативной памяти, сопоставленные промежуточным и определенным в программе переменным. Независимые процессы не влияют на результаты работы друг друга, так как не могут изменять значения переменных друг у друга. Они могут только явиться причиной в задержках исполнения друг друга, так как вынуждены разделять ресурсы системы.

Взаимодействующие процессы совместно используют некоторые (общие) переменные, и выполнение одного процесса может повлиять на выполнение другого. Как мы уже говорили, при выполнении вычислительные процессы разделяют ресурсы системы. Подчеркнем, что при рассмотрении вопросов синхронизации вычислительных процессов из числа разделяемых ими ресурсов исключаются центральный процессор и программы, реализующие эти процессы, то есть с логической точки зрения каждому процессу соответствуют свои процессор и программа, хотя в реальных системах обычно несколько процессов разделяют один процессор и одну или несколько программ. Многие ресурсы вычислительной системы могут совместно использоваться несколькими процессами, но в каждый момент времени к разделяемому ресурсу может иметь доступ только один процесс. Ресурсы, которые не допускают одновременного использования несколькими процессами, называются *критическими*.

Если несколько вычислительных процессов хотят пользоваться критическим ресурсом в режиме разделения, им следует синхронизировать свои действия таким образом, чтобы ресурс всегда находился в распоряжении не более чем одного из них. Если один процесс пользуется в данный момент критическим ресурсом, то все остальные процессы, которым нужен этот ресурс, должны ждать, пока он не освободится. Если в операционной системе не предусмотрена защита от одновременного доступа процессов к критическим ресурсам, в ней могут возникать ошибки, которые трудно обнаружить и исправить. Основной причиной возникновения этих ошибок является то, что процессы в мультипрограммных операционных сис-

темах развиваются с различными скоростями и относительные скорости развития каждого из взаимодействующих процессов не подвластны и не известны ни одному из них. Более того, на их скорости могут влиять решения планировщиков, касающиеся других процессов, с которыми ни одна из этих программ не взаимодействует. Кроме того, содержание и скорость исполнения одного из них обычно не известны другому процессу. Поэтому влияние, которое оказывают друг на друга взаимодействующие процессы, не всегда предсказуемо и воспроизводимо.

Взаимодействовать могут либо конкурирующие процессы, либо процессы, обрабатывающие информацию совместно. *Конкурирующие процессы* действуют относительно независимо друг от друга, однако они имеют доступ к некоторым общим переменным. Их независимость заключается в том, что они могут работать друг без друга, поодиночке. Но при своем выполнении они могут работать и параллельно, и тогда они иногда начинают конкурировать при обращении к этим общим переменным. Таким образом, их независимость относительна.

Приведем несколько наиболее известных примеров конкурирующих процессов и продемонстрируем появление ошибок. В качестве первого примера рассмотрим работу двух процессов P1 и P2 с общей переменной X. Пусть оба процесса асинхронно, независимо один от другого, изменяют (например, увеличивают) значение переменной X, считывая ее значение в локальную область памяти Ri[1], при этом каждый процесс выполняет во времени некоторые последовательности операций (табл. 7.1). Здесь мы рассмотрим не все операторы каждого из процессов, а только те, в которых осуществляется работа с общей переменной X. Каждому из операторов мы присвоили некоторый условный номер.

Таблица 7.1. Пример конкурирующих процессов

Номер оператора	Процесс P1	Номер оператора	Процесс P2
1	R1 := X	4	R2 := X
2	R1 := R1 + 1	5	R2 := R2 + 1
3	X := R1	6	X := R2

Поскольку при мультипрограммировании процессы могут иметь различные скорости исполнения, то может иметь место любая последовательность выполнения операций во времени. Например, если сначала будут выполнены все операции процесса P1, а уже потом — все операции процесса P2 (рис. 7.1) или, наоборот, сначала — операции 4–6, а затем — операции 1–3, то в итоге переменная X получит значение, равное X + 2.

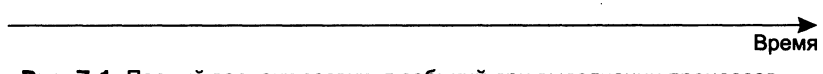


Рис. 7.1. Первый вариант развития событий при выполнении процессов

[1] Ri — это просто имя переменной для процесса с номером i.

Однако если в промежуток времени между выполнением операций 1 и 3 будет выполнена хотя бы одна из операций 4–6 (рис. 7.2), то значение переменной X после выполнения всех операций будет не (X + 2), а (X + 1).

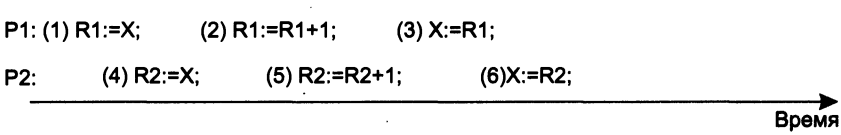


Рис. 7.2. Второй вариант развития событий при выполнении процессов

Понятно, что это очень серьезная ошибка. Например, если бы процессы P1 и P2 осуществляли продажу билетов и переменная X фиксировала количество уже проданных, то в результате некорректного взаимодействия было бы продано несколько билетов на одно и то же место. К сожалению, такого рода ошибки трудноуловимы, поскольку они иногда возникают, иногда нет.

В качестве второго примера рассмотрим ситуацию, которая еще совсем недавно была достаточно актуальной для первых персональных компьютеров. Пусть на персональном компьютере с простейшей однопрограммной операционной системой (типа MS DOS) установлена некоторая резидентная программа с условным названием TIME, которая по нажатию комбинации клавиш (например, Ctrl+T) воспроизводит на экране дисплея время в виде 18:20:59, и допустим, что значения переменных, обозначающих час, минуты и секунды, равны 18, 20 и 59 соответственно, причем вывод на дисплей осуществляется справа налево (сначала секунды, затем минуты и, наконец, часы). Пусть сразу же после передачи программой TIME на дисплей информации «59 секунд» генерируется прерывание от таймера, и значение времени обновляется: 18:21:00.

После этого программа TIME, прерванная таймером, продолжит свое выполнение, и на дисплей будут выданы значения: минуты (21), часы (18). В итоге на экране мы увидим: 18:21:59.

Рассмотрим теперь несколько иной случай развития событий обновления значений времени по сигналу таймера. Если программа ведения системных часов после вычислений количества секунд 59 + 1 = 60 и замены его на 00 прерывается от нажатия клавиш Ctrl+T, то есть программа не успевает осуществить пересчет количества минут, то время, индицируемое на дисплее, станет равным 18:20:00. И в этом случае мы получим неверное значение времени.

Наконец, в качестве третьего примера приведем пару процессов, которые изменяют различные поля записей служащих какого-либо предприятия [17]. Пусть процесс АДРЕС изменяет домашний адрес служащего, а процесс СТАТУС — его должность и зарплату. Пусть каждый процесс для выполнения своей функции копирует всю запись СЛУЖАЩИЙ в свою рабочую область. Предположим, что каждый процесс должен обработать некоторую запись ИВАНОВ. Предположим также, что после того как процесс АДРЕС скопировал запись ИВАНОВ в свою рабочую область, но до того как он записал скорректированную запись обратно, процесс СТАТУС скопировал первоначальную запись ИВАНОВ в свою рабочую область.

Изменения, выполненные тем из процессов, который первым запишет скорректированную запись назад в файл СЛУЖАЩИЕ, будут утеряны, и, возможно, никто об этом не узнает.

Чтобы предотвратить некорректное исполнение конкурирующих процессов вследствие нерегламентированного доступа к разделяемым переменным, необходимо ввести понятие *взаимного исключения*, согласно которому два процесса не должны одновременно обращаться к разделяемым переменным.

Процессы, выполняющие общую совместную работу таким образом, что результаты вычислений одного процесса в явном виде передаются другому, то есть они обмениваются данными и именно на этом построена их работа, называются *сотрудничающими*. Взаимодействие сотрудничающих процессов удобнее всего рассматривать в схеме *производитель–потребитель* (produces–consumer), или, как часто говорят, *поставщик–потребитель*.

Кроме реализации в операционной системе средств, организующих взаимное исключение и, тем самым, регулирующих доступ процессов к критическим ресурсам, в ней должны быть предусмотрены средства, синхронизирующие работу взаимодействующих процессов. Другими словами, процессы должны обращаться друг к другу не только ради синхронизации с целью взаимного исключения при обращении к критическим ресурсам, но и ради обмена данными.

Допустим, что «поставщик» — это процесс, который отправляет порции информации (сообщения) другому процессу, имя которого — «потребитель». Например, некоторая задача пользователя, порождающая данные для вывода их на печать, может выступать как поставщик, а системный процесс, который выводит эти строки на устройство печати, — как потребитель. Один из методов, применяемых при передаче сообщений, состоит в том, что заводится *пул* (pool)[1] свободных буферов, каждый из которых может содержать одно сообщение. Заметим, что длина сообщения может быть произвольной, но ограниченной размером буфера.

В этом случае между процессами «поставщик» и «потребитель» будем иметь очередь заполненных буферов, содержащих сообщения. Когда поставщик хочет послать очередное сообщение, он добавляет в конец этой очереди еще один буфер. Потребитель, чтобы получить сообщение, забирает из очереди буфер, который стоит в ее начале. Такое решение, хотя и кажется тривиальным, требует, чтобы поставщик и потребитель синхронизировали свои действия. Например, они должны следить за количеством свободных и заполненных буферов. Поставщик может передавать сообщения только до тех пор, пока имеются свободные буферы. Аналогично, потребитель может получать сообщения, только если очередь не пуста. Ясно, что для учета заполненных и свободных буферов нужны разделяемые переменные, поэтому, так же как и для конкурирующих процессов, для сотрудничающих процессов тоже возникает необходимость во взаимном исключении.

Таким образом, до окончания обращения одной задачи к общим переменным следует исключить возможность обращения к ним другой задачи. Эта ситуация и на-

[1] Совокупность однородных динамически распределяемых объектов, например блоков памяти одинаковой длины.

зывается взаимным исключением. Другими словами, при организации различного рода взаимодействующих процессов приходится организовывать взаимное исключение и решать проблему корректного доступа к общим переменным (критическим ресурсам). Те места в программах, в которых происходит обращение к критическим ресурсам, называются *критическими секциями* (Critical Section, CS). Решение проблемы заключается в организации такого доступа к критическому ресурсу, при котором только одному процессу разрешается входить в критическую секцию. Данная задача только на первый взгляд кажется простой, ибо критическая секция, вообще говоря, не является последовательностью операторов программы, а является процессом, то есть последовательностью действий, которые выполняются этими операторами. Другими словами, несколько процессов могут выполнять критические секции, использующие одну и ту же последовательность операторов программы.

Когда какой-либо процесс находится в своей критической секции, другие процессы могут, конечно, продолжать свое исполнение, но без входа в их критические секции. Взаимное исключение необходимо только в том случае, когда процессы обращаются к разделяемым (общим) данным. Если же они выполняют операции, которые не ведут к конфликтным ситуациям, процессы должны иметь возможность работать параллельно. Когда процесс выходит из своей критической секции, то одному из остальных процессов, ожидающих входа в свои критические секции, должно быть разрешено продолжить работу (если в этот момент действительно есть процесс в состоянии ожидания входа в свою критическую секцию).

Обеспечение взаимного исключения является одной из ключевых проблем параллельного программирования. При этом можно перечислить требования к критическим секциям [17, 54].

- В любой момент времени только один процесс должен находиться в своей критической секции.
- Ни один процесс не должен бесконечно долго находиться в своей критической секции.
- Ни один процесс не должен бесконечно долго ожидать разрешение на вход в свою критическую секцию. В частности:
  - никакой процесс, бесконечно долго находящийся вне своей критической секции (что допустимо), не должен задерживать выполнение других процессов, ожидающих входа в свои критические секции (другими словами, процесс, работающий вне своей критической секции, не должен блокировать критическую секцию другого процесса);
  - если два процесса хотят войти в свои критические секции, то принятие решения о том, кто первым войдет в критическую секцию, не должно быть бесконечно долгим.
- Если процесс, находящийся в своей критической секции, завершается естественным или аварийным путем, то режим взаимного исключения должен быть отменен, с тем чтобы другие процессы получили возможность входить в свои критические секции.

Было предложено несколько способов решения этой проблемы: программных и аппаратных; локальных низкоуровневых и глобальных высокоуровневых; предусматривающих свободное взаимодействие между процессами и требующих строгого соблюдения протоколов.

Средства синхронизации и связи взаимодействующих вычислительных процессов

Все известные средства решения проблемы взаимного исключения основаны на использовании специально введенных аппаратных возможностей. К этим аппаратным возможностям относятся: блокировка памяти, специальные команды типа «проверка и установка» и механизмы управления системой прерываний, которые позволяют организовать такие средства, как семафорные операции, мониторы, почтовые ящики и др. С помощью перечисленных средств можно разрабатывать взаимодействующие процессы, при исполнении которых будут корректно решаться все задачи, связанные с проблемой критических секций. Рассмотрим эти средства в следующем порядке по мере их усложнения, перехода к функциям операционной системы и увеличения предоставляемых ими удобств, опираясь на уже древнюю, но все же еще достаточно актуальную работу Дейкстры [10]. Заметим, что этот материал позволяет в полной мере осознать проблемы, возникающие при организации параллельных взаимодействующих вычислений. Эта работа Дейкстры полезна, прежде всего, с методической точки зрения, поскольку она позволяет понять наиболее тонкие моменты в этой проблематике.

Использование блокировки памяти при синхронизации параллельных процессов

Все вычислительные машины и системы (в том числе и с многопортовыми блоками оперативной памяти) имеют средство для организации взаимного исключения, называемое *блокировкой памяти*. Блокировка памяти запрещает одновременное исполнение двух (и более) команд, которые обращаются к одной и той же ячейке памяти. Блокировка памяти имеет место всегда, то есть это обязательное условие функционирования компьютера. Соответственно, поскольку в некоторой ячейке памяти хранится значение разделяемой переменной, то получить доступ к ней может только один процесс, несмотря на возможное совмещение выполнения команд во времени на различных процессорах (или на одном процессоре, но с конвейерной организацией параллельного выполнения команд).

Механизм блокировки памяти предотвращает одновременный доступ к разделяемой переменной, но не предотвращает чередование доступа. Таким образом, если критические секции исчерпываются одной командой обращения к памяти, данное средство может быть достаточным для непосредственной реализации взаимного исключения. Если же критические секции требуют более одного обращения к памяти, то задача становится сложной, но алгоритмически разрешимой. Рассмотрим

различные попытки использования механизма блокировки памяти для организации взаимного исключения при выполнении критических секций и покажем некоторые важные моменты, пренебрежение которыми приводит к неприемлемым или даже к ошибочным решениям.

Возможные проблемы при организации взаимного исключения при условии использования только блокировки памяти

Пусть имеется два или более циклических процессов с абстрактными критическими секциями, то есть каждый процесс состоит из двух частей: некоторой критической секции и оставшейся части кода, которая не работает с общими (критическими) переменными. Пусть эти два процесса асинхронно разделяют во времени единственный процессор либо выполняются на отдельных процессорах, то есть каждый из них имеет доступ к некоторой общей области памяти, с которой и работают критические секции. Проиллюстрируем эту ситуацию с помощью рис. 7.3.

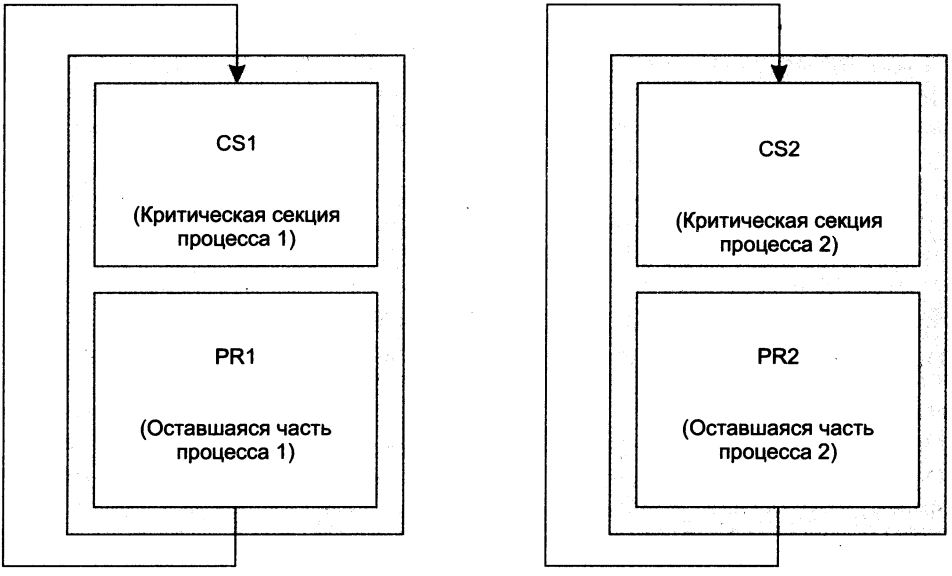


Рис. 7.3. Модель взаимодействующих процессов

Задача вроде бы легко решается, если потребовать, чтобы процессы ПР1 и ПР2 входили в свои критические секции попеременно. Для этого одна общая переменная может хранить указатель на тот процесс, чья очередь войти в критическую секцию. Текст этого решения на языке, близком к Паскалю, приведен в листинге 7.1.

Листинг 7.1. Текст программы для первого решения

```
Var перекл : integer;
Begin перекл := 1; {при перекл=1 в критической секции находится процесс ПР1}
Parbegin
```

```
  While true do
    Begin
      while перекл = 2 do begin end;
      CS1; { критическая секция процесса ПР1 }
      перекл := 2;
      PR1; { оставшаяся часть процесса ПР1 }
    End
And
  While true do
    Begin
      while перекл = 1 do begin end;
      CS2; { критическая секция процесса ПР2 }
      перекл := 1;
      PR2; { оставшаяся часть процесса ПР2 }
    End
Parend
End.
```

Здесь и далее языковая конструкция следующего типа означает параллельность выполнения K описываемых последовательных процессов:

```
parbegin...S11; S12; ... ; S1N1
  and... S21; S22; ... ; S2N2
  ...
  and... SK1; SK2; ... ; SKN1k
parend
```

Конструкция из операторов **S11; S12; ...; S1N1** выполняется последовательно (оператор за оператором), о чем свидетельствует наличие точки с запятой между ними.

Следующая языковая конструкция означает, что каждый процесс может выполняться неопределенно долгое время фактически бесконечное количество раз:

```
while true do
begin S1; S2; SN end
```

Наконец, конструкция типа **begin end** означает просто «пустой» оператор.

Итак, решение, представленное в листинге 7.1, обеспечивает нам взаимное исключение в работе критических секций. Однако если бы фрагмент программы **PR1** был намного длиннее, чем фрагмент **PR2**, или если бы процесс ПР1 был заблокирован в секции **PR1**, или если бы процессор для ПР2 обладал более высоким быстродействием, то процесс ПР2 вскоре вынужден был бы ждать входа в свою критическую секцию **CS2**, хотя процесс ПР1 и был бы вне **CS1**. Такое ожидание могло бы оказаться слишком долгим, то есть для этого решения один процесс вне своей критической секции может помешать другому войти в свою критическую секцию.

Попробуем устранить это блокирование с помощью двух общих переменных, которые будут использоваться как флаги, указывая, находится или нет соответствующий процесс в своей критической секции. То есть с каждым из процессов ПР1 и ПР2 будет связана переменная, которая принимает значение **true**, когда процесс находится в своей критической секции, и **false** — в противном случае. Прежде чем войти в свою критическую секцию, процесс проверит значение флага другого процесса. Если это значение равно **true**, процессу не разрешается входить в свою критическую секцию. В противном случае процесс установит собственный флаг и вой-

дет в критическую секцию. Этот алгоритм взаимного исключения представлен в листинге 7.2.

Листинг 7.2. Второй вариант реализации взаимного исключения

```
Var перекл1,перекл2.: boolean;
begin перекл1:=false;
  перекл2:=false;
parbegin
   while true do
    begin
      while перекл2 do
        begin
        end;
      перекл1:=true;
      CS1 { критическая секция процесса ПР1 }
      перекл1:=false;
      PR1 { процесс ПР1 после критической секции }
    end
and
   while true do
    begin
      while перекл1 do
        begin
        end;
      перекл2:=true;
        CS2 { Критическая секция процесса ПР2 }
      перекл2:=false;
        PR2 { процесс ПР2 после критической секции }
    end
parend
end.
```

Данный алгоритм, увы, не гарантирует полного выполнения условия нахождения только одного процесса внутри критической секции. Отсутствие гарантий связано с различными, в общем случае, скоростями развития процессов. Поэтому, например, между проверкой значения переменной **перекл2** процессом ПР1 и последующей установкой им значения переменной **перекл1** параллельно выполняющийся процесс ПР2 может установить **перекл2** в значение **true**, так как переменная **перекл1** еще не успела установиться в значение **true**. Отсюда следует, что оба процесса могут войти в свои критические секции одновременно.

Следующий (третий) вариант решения этой задачи (листинг 7.3) усиливает взаимное исключение, так как в процессе ПР1 проверка значения переменной **перекл2** выполняется только после установки переменной **перекл1** в значение **true** (аналогично для ПР2).

Листинг 7.3. Третий вариант реализации взаимного исключения

```
var перекл1, перекл2 : boolean;
begin перекл1:=false; перекл2:=false;
parbegin
  ПР1: while true do
       begin
         перекл1:=true;
```

```
        while перекл2 do
          begin  end;
        CS1 { критическая секция процесса ПР1 }
        перекл1:=false;
        PR1 { ПР1 после критической секции }
      end
and
  ПР2: while true do
      begin
        перекл2:=true;
        while перекл1 do
          begin  end;
        CS2 { критическая секция процесса ПР2 }
        перекл2:=false;
        PR2 { ПР2 после критической секции }
      end
parend
end.
```

Алгоритм, приведенный в листинге 7.3, также имеет свои недостатки. Действительно, возможна ситуация, когда оба процесса одновременно установят свои флаги в значение true и войдут в бесконечный цикл. В этом случае будет нарушено требование отсутствия бесконечного ожидания входа в свою критическую секцию. То есть, предположив, что скорости исполнения процессов произвольны, мы получили такую последовательность событий, при которой процессы вообще перестанут нормально выполняться.

Все рассмотренные попытки решить задачу взаимного исключения при выполнении критических секций иллюстрируют нам некоторые тонкие моменты, лежащие в основе этой проблемы, и показывают, что не все так просто.

Последний рассматриваемый вариант решения задачи взаимного исключения, опирающийся только на блокировку памяти, — это известный алгоритм, предложенный математиком Деккером.

Алгоритм Деккера

Алгоритм Деккера основан на использовании трех переменных (листинг 7.4): перекл1, перекл2 и ОЧЕРЕДЬ. Пусть по-прежнему переменная перекл1 устанавливается в true тогда, когда процесс ПР1 хочет войти в свою критическую секцию (для ПР2 аналогично), а значение переменной ОЧЕРЕДЬ указывает, чье сейчас право сделать попытку входа при условии, что оба процесса хотят выполнить свои критические секции.

Листинг 7.4. Алгоритм Деккера

```
label 1, 2;
var   перекл1, перекл2: boolean;
  ОЧЕРЕДЬ : integer;
Begin  перекл1:=false; перекл2:=false;
  ОЧЕРЕДЬ:=1;
  parbegin
     while true do
      begin перекл1:=true;
```

продолжение ⇩

Листинг 7.4 (*продолжение*)

```
    1: if перекл2=true then
          if ОЧЕРЕДЬ=1 then go to 1
            else begin перекл1:=false;
                while ОЧЕРЕДЬ=2 do
                  begin    end
                end
        else begin
              CS1 { критическая секция ПР1 }
              ОЧЕРЕДЬ:=2; перекл1:=false
             end
      end
  and
    while true do
      begin перекл2:=1;
    2: if перекл1=true then
          if ОЧЕРЕДЬ=2 then go to 2
          else begin перекл2:=false;
                 while ОЧЕРЕДЬ=1 do
                 begin    end
               end
        else begin
              CS2 { критическая секция ПР2 }
              ОЧЕРЕДЬ:=1; перекл2:=false
             end
      end
  parend
end.
```

Если перекл2 = true и перекл1 = false, то выполняется критическая секция процесса ПР2 независимо от значения переменной ОЧЕРЕДЬ. Аналогично для случая перекл2 = false и перекл1 = true.

Если же оба процесса хотят выполнить свои критические секции, то есть перекл2 = = true и перекл1 = true, то выполняется критическая секция того процесса, на который указывает значение переменной ОЧЕРЕДЬ, независимо от скоростей развития обоих процессов. Использование переменной ОЧЕРЕДЬ совместно с переменными перекл1 и перекл2 в алгоритме Деккера позволяет гарантированно решать проблему критических секций. То есть переменные перекл1 и перекл2 гарантируют, что взаимное выполнение не может иметь места; переменная ОЧЕРЕДЬ гарантирует, что не может быть взаимной блокировки, так как переменная ОЧЕРЕДЬ не меняет своего значения во время выполнения программы принятия решения о том, кому же сейчас проходить свою критическую секцию.

Тем не менее реализаций критических секций на основе описанного алгоритма практически не встречается из-за их чрезмерной сложности, особенно тогда, когда требуется обобщить алгоритм Деккера с двух до N процессов.

Синхронизация процессов с помощью операции проверки и установки

Операция проверки и установки является, так же как и блокировка памяти, одним из аппаратных средств, которые могут быть использованы для решения задачи вза-

имного исключения при выполнении критической секции. Данная операция реализована во многих компьютерах. Так, в знаменитой машине IBM 360 (370) эта команда называлась TS (Test and Set — проверка и установка). Команда TS является двухадресной (двухоперандной). Ее действие заключается в том, что процессор присваивает значение второго операнда первому, после чего второму операнду присваивается значение, равное единице. Команда TS является неделимой операцией, то есть между ее началом и концом не могут выполняться никакие другие команды.

Чтобы использовать команду TS для решения проблемы критической секции, свяжем с ней переменную common, которая будет общей для всех процессов, использующих некоторый критический ресурс. Данная переменная будет принимать единичное значение, если какой-либо из взаимодействующих процессов находится в своей критической секции. Кроме того, с каждым процессом свяжем свою локальную переменную, которая принимает значение, равное единице, если данный процесс хочет войти в свою критическую секцию. Операция TS будет присваивать значение common локальной переменной и устанавливать common в единицу. Соответствующая программа решения проблемы критической секции на примере двух параллельных процессов приведена в листинге 7.5.

Листинг 7.5. Взаимное исключение с помощью операции проверки и установки

```
var common, local1, local2 : integer;
begin
  common:=0;
  parbegin
  ПР1: while true do
       begin
         local1:=1;
         while local1=1 do TS(local1,common);
         CS1; { критическая секция процесса ПР1 }
         common:=0;
         PR1; { ПР1 после критической секции }
       end
  and
  ПР2: while true do
       begin
         local2:=1;
         while local2=1 do TS(local2,common);
         CS2; { критическая секция процесса ПР2 }
         common:=0;
         PR2; { ПР2 после критической секции }
       end
  parend
end.
```

Предположим, что первым хочет войти в свою критическую секцию процесс ПР1. В этом случае значение local1 сначала установится в единицу, а после цикла проверки с помощью команды TS(local1,common) — в нуль. При этом значение common станет равным единице. Процесс ПР1 войдет в свою критическую секцию. После выполнения критической секции переменная common примет значение, равное нулю, что даст возможность второму процессу ПР2 войти в свою критическую секцию.

Безусловно, мы предполагаем, что в компьютере реализована блокировка памяти, то есть операция common := 0 неделима. Команда проверки и установки значительно упрощает решение проблемы критических секций. Главная черта этой команды — ее неделимость.

Основной недостаток использования команд типа проверки и установки состоит в следующем: находясь в цикле проверки переменной common, процессы впустую потребляют время центрального процессора и другие ресурсы. Действительно, если предположить, что произошло прерывание процесса ПР1 во время выполнения своей критической секции в соответствии с некоторой дисциплиной обслуживания, и начал выполняться процесс ПР2, то он войдет в цикл проверки, впустую тратя процессорное время. В этом случае, до тех пор пока диспетчер супервизора не поставит на выполнение процесс ПР1 и не даст ему закончиться, процесс ПР2 не сможет войти в свою критическую секцию.

В микропроцессорах архитектуры ia32, с которыми мы теперь сталкиваемся постоянно, работая на персональных компьютерах, имеются специальные команды BTC, BTS, BTR, которые как раз и являются вариантами реализации команды проверки и установки. Рассмотрим одну из них — BTS.

Команда BTS (Bit Test and Reset — проверка и установка бита) является двухадресной [20]. По этой команде процессор сохраняет значение бита из первого операнда со смещением, указанным вторым операндом, во флаге CF (Carry Flag — флаг переноса)[1] регистра флагов, а затем устанавливает указанный бит в 1. Значение индекса выбираемого бита может быть представлено постоянным непосредственным значением в команде BTS или значением в общем регистре. В этой команде используется только 8-разрядное непосредственное значение. Значение этого операнда берется по модулю 32, таким образом, смещение битов находится в диапазоне от 0 до 31. Это позволяет выбрать любой бит внутри регистра. Для битовых строк в памяти это поле непосредственного значения дает только смещение внутри слова или двойного слова.

С учетом изложенного можно привести фрагмент кода, в котором данная команда используется для решения проблемы взаимного исключения (листинг 7.6).

Листинг 7.6. Фрагмент программы с критической секцией на ассемблере

```
    .
    .
L:  BTC M,1
    JC  L
    .
    .       ; критическая секция
    .
    AND M,0B
    .
    .
```

Однако здесь следует заметить, что некоторые ассемблеры поддерживают значения битовых смещений больше 31, используя поле непосредственного значения

[1] Располагается в слове состояния программы.

в комбинации с полем смещения операнда в памяти. В этом случае младшие 3 или 5 битов (3 — для 16-разрядных операндов, 5 — для 32-разрядных операндов), определяющие смещение бита (второй операнд команды), сохраняются в поле непосредственного операнда, а старшие биты сдвигаются и комбинируются с полем смещения. Процессор же игнорирует ненулевые значения старших битов поля второго операнда [20]. При доступе к памяти процессор обращается к четырем байтам (для 32-разрядного операнда), начинающимся по полученному следующим образом адресу:

Effective Address + (4 × (BitOffset DIV 32))

Либо (для 16-разрядного операнда) процессор обращается к двум байтам, начинающимся по адресу:

Effective Address + (2 × (BitOffset DIV 16))

Такое обращение происходит, даже если необходим доступ только к одному байту. Поэтому избегайте ссылок к областям памяти, близким к «пустым» адресным пространствам. В частности, избегайте ссылок на распределенные в памяти регистры ввода-вывода. Вместо этого используйте команду `MOV` для загрузки и сохранения значений по таким адресам и регистровую форму команды `BTS` для работы с данными.

Несмотря на то, что и алгоритм Деккера, основанный только на блокировке памяти, и операция проверки и установки пригодны для реализации взаимного исключения, оба эти приема очень неэффективны. Всякий раз, когда один из процессов выполняет свою критическую секцию, любой другой процесс, который пытается войти в свою критическую секцию, попадает в цикл проверки соответствующих переменных-флагов, регламентирующих доступ к критическим переменным. При таком неопределенном пребывании в цикле, которое называется *активным ожиданием*, напрасно расходуется процессорное время, поскольку процесс имеет доступ к тем общим переменным, которые и определяют возможность или невозможность входа в критическую секцию. При этом процесс отнимает ценное время центрального процессора, на самом деле ничего реально не выполняя. Как результат мы получаем общее замедление работы вычислительной системы.

До тех пор пока процесс, занимающий в данный момент критический ресурс, не решит его уступить, все другие процессы, ожидающие этого ресурса, могли бы вообще не конкурировать за процессорное время. Для этого их нужно перевести в состояние ожидания (заблокировать). Когда вход в критическую секцию снова освободится, можно будет опять перевести заблокированный процесс в состояние готовности к выполнению и дать ему возможность получить процессорное время. Самый простой способ предоставить процессорное время только одному вычислительному процессу — отключить систему прерываний, поскольку тогда никакое внешнее событие не сможет прервать выполняющийся процесс. Однако это, как мы уже знаем, приведет к тому, что система не сможет реагировать на внешние события.

Вместо того чтобы связывать с каждым процессом собственную переменную, как это было в рассмотренных нами решениях, можно со всем множеством конкури-

рующих критических секций связать одну переменную, которую Дейкстра предложил рассматривать как некоторый «ключ». Вначале доступ к критической секции открыт. Однако перед входом в свою критическую секцию процесс забирает ключ и тем самым блокирует другие процессы. Покидая критическую секцию, процесс открывает доступ, возвращая ключ на место. Если процесс, который хочет войти в свою критическую секцию, обнаруживает отсутствие ключа, он должен быть переведен в состояние блокирования до тех пор, пока процесс, имеющий ключ, не вернет его. Таким образом, каждый процесс, входящий в критическую секцию, должен вначале проверить, доступен ли ключ, и если это так, то сделать его недоступным для других процессов. Причем самым главным должно быть то, что эти два действия должны быть неделимыми, чтобы два или более процессов не могли одновременно получить доступ к ключу. Более того, проверку возможности входа в критическую секцию лучше всего выполнять не самим конкурирующим процессам, так как это приводит к активному ожиданию, а возложить эту функцию на операционную систему. Таким образом, мы подошли к одному из самых главных механизмов решения проблемы взаимного исключения — семафорам Дейкстры.

Семафорные примитивы Дейкстры

Понятие семафорных механизмов было введено Дейкстрой [10]. *Семафор* (semaphore) — это переменная специального типа, которая доступна параллельным процессам только для двух операций — закрытия и открытия, названных соответственно операциями P и V[1]. Эти операции являются примитивами относительно семафора, который указывается в качестве параметра операций. Здесь семафор играет роль вспомогательного критического ресурса, так как операции P и V неделимы при своем выполнении и взаимно исключают друг друга.

Семафорный механизм работает по схеме, в которой сначала исследуется состояние критического ресурса, идентифицируемое значением семафора, а затем уже осуществляется допуск к критическому ресурсу или отказ от него на некоторое время. При отказе доступа к критическому ресурсу используется режим *пассивного ожидания*. Поэтому в состав механизма включаются средства формирования и обслуживания очереди ожидающих процессов. Эти средства реализуются супервизором операционной системы. Необходимо отметить, что в силу взаимного исключения примитивов попытка в различных параллельных процессах одновременно выполнить примитив над одним и тем же семафором приведет к тому, что она окажется успешной только для одного процесса. Все остальные процессы на время выполнения примитива будут взаимно исключены.

Основным достоинством семафорных операций является отсутствие состояния активного ожидания, что может существенно повысить эффективность работы мультизадачной системы.

В настоящее время на практике используется много различных видов семафорных механизмов [41]. Варьируемыми параметрами, которые отличают различные виды примитивов, являются начальное значение и диапазон изменения значений

[1] P — от голландского Proberen (проверить), V — от голландского Verhogen (увеличить).

семафора, логика действий семафорных операций, количество семафоров, доступных для обработки при исполнении отдельного примитива.

Обобщенный смысл примитива P(S) состоит в проверке текущего значения семафора S. Если оно не меньше нуля, то осуществляется переход к следующей за примитивом операции. В противном случае процесс снимается на некоторое время с выполнения и переводится в состояние пассивного ожидания. Находясь в списке заблокированных, ожидающий процесс не проверяет семафор непрерывно, как в случае активного ожидания. Вместо него процессор может исполнять другой процесс, реально выполняющий какую-то полезную работу.

Операция V(S) связана с увеличением значения семафора на единицу и переводом одного или нескольких процессов в состояние готовности к исполнению центральным процессором.

Отметим еще раз, что операции P и V выполняются операционной системой в ответ на запрос, выданный некоторым процессом и содержащий имя семафора в качестве параметра.

Рассмотрим первый вариант алгоритма работы семафорных операций (листинг 7.7). Допустимыми значениями семафоров являются только целые числа. Двоичным семафором будем называть семафор, максимально возможное значение которого равно единице. Двоичный семафор[1] либо открыт, либо закрыт. В случае, когда семафор может принимать более двух значений, его называют N-ичным. Есть реализации, в которых семафорные переменные не могут быть отрицательными, а есть и такие, где отрицательное значение указывает на длину очереди процессов, стоящих в состоянии ожидания открытия семафора.

Листинг 7.7. Вариант реализации семафорных примитивов

```
P(S):  S:=S-1;
    if S<0  then { остановить процесс и поместить его в очередь ожидания к семафору S };

V(s): if S<0 then { поместить один из ожидающих процессов очереди семафора S в очередь
готовности };
      S:=S+1;
```

Заметим, что для работы с семафорными переменными необходимо еще иметь операцию инициализации самого семафора, то есть задания ему начального значения. Обычно эту операцию называют InitSem; как правило, она имеет два параметра — имя семафорной переменной и ее начальное значение, то есть обращение имеет вид

```
InitSem ( Имя_семафора, Начальное_значение_семафора );
```

Попытаемся на нашем примере двух конкурирующих процессов ПР1 и ПР2 проанализировать использование данных семафорных примитивов для решения проблемы критической секции. Программа, иллюстрирующая это решение, представлена в листинге 7.8.

Листинг 7.8. Взаимное исключение с помощью семафорных операций

```
var S:semafor;
begin InitSem(S,1);
```

продолжение ⇩

[1] Двоичные семафоры иногда называют мьютексами (см. далее).

Листинг 7.8 (продолжение)

```
parbegin
  ПР1: while true do
       begin
         P(S);
         CS1 ;   { критическая секция процесса ПР1 }
         V(S)
       end
and
  ПР2: while true do
       begin
         P(S);
         CS2 ;   { критическая секция процесса ПР2 }
         V(S)
       end
parend
end.
```

Семафор S имеет начальное значение, равное 1. Если процессы ПР1 и ПР2 попытаются одновременно выполнить примитив P(S), то это удастся успешно сделать только одному из них. Предположим, это сделал процесс ПР2, тогда он закрывает семафор S, после чего выполняется его критическая секция. Процесс ПР1 в рассматриваемой ситуации будет заблокирован на семафоре S. Тем самым гарантируется взаимное исключение.

После выполнения примитива V(S) процессом ПР2 семафор S открывается, указывая на возможность захвата каким-либо процессом освободившегося критического ресурса. При этом производится перевод процесса ПР1 из заблокированного состояния в состояние готовности.

На уровне реализации возможно одно из двух решений в отношении процессов, которые переводятся из очереди ожидания в очередь готовности при выполнении примитива V:

- процесс при его активизации (выборка из очереди готовности) вновь пытается выполнить примитив P, считая предыдущую попытку неуспешной;
- процесс при помещении его в очередь готовности отмечается как успешно выполнивший примитив P, тогда при его активизации управление будет передано не на повторное выполнение примитива P, а на команду, следующую за ним.

Рассмотрим первый способ реализации. Пусть процесс ПР2 в некоторый момент времени выполняет операцию P(S). Тогда семафор S становится равным нулю. Пусть далее процесс ПР1 пытается выполнить операцию P(S). Процесс ПР1 в этом случае блокируется на семафоре S, так как значение семафора S равнялось нулю, а теперь станет равным −1. После выполнения критической секции процесс ПР2 выполняет операцию V(S), при этом значение семафора S становится равным нулю, а процесс ПР1 переводится в очередь готовности. Пусть через некоторое время процесс ПР1 будет активизирован, то есть выведен из состояния ожидания, и сможет продолжить свое исполнение. Он повторно попытается выполнить операцию P(S), однако это ему не удастся, так как S=0. Процесс ПР1 блокируется на семафоре, а его значение становится равным −1. Если через некоторое время процесс ПР2 попытается выполнить P(S), то он тоже заблокируется. Таким образом, возникнет

так называемая *тупиковая ситуация*, так как разблокировать процессы ПР1 и ПР2 некому.

При втором способе реализации тупика не будет. Действительно, пусть все происходит так же до момента окончания исполнения процессом ПР2 примитива V(S). Пусть примитив V(S) выполнен, и S=0. Через некоторое время процесс ПР1 активизируется. Согласно данному способу реализации он сразу же попадет в свою критическую секцию. При этом никакой другой процесс не попадет в свою критическую секцию, так как семафор остается закрытым. После исполнения своей критической секции процесс ПР1 выполнит V(S). Если за время выполнения критической секции процесса ПР1 процесс ПР2 не сделает попыток выполнить операцию P(S), семафор S установится в единицу. В противном случае значение семафора будет не больше нуля. Но в любом варианте после завершения операции V(S) процессом ПР1 доступ к критическому ресурсу со стороны процесса ПР2 будет разрешен.

Заметим, что возникновение тупиков возможно в случае несогласованного выбора механизма извлечения процессов из очереди, с одной стороны, и выбора алгоритмов семафорных операций, с другой.

Возможен другой алгоритм работы семафорных операций:

```
P(S):   if S>=1 then S:=S-1
        else WAIT(S){ остановить процесс и поместить в очередь ожидания к семафору S }
V(S):   if S<0 then RELEASE(S){ поместить один из ожидающих процессов очереди семафора S
в очередь готовности };
    S:=S+1.
```

Здесь вызов WAIT(S) означает, что супервизор ОС должен перевести задачу в состояние ожидания, причем очередь процессов связана с семафором S. Вызов RELEASE(S) означает обращение к диспетчеру задач с просьбой перевести первый из процессов, стоящих в очереди S, в состояние готовности к исполнению.

Использование семафорных операций, выполненных подобным образом, позволяет решать проблему критических секций на основе первого способа реализации, причем без опасности возникновения тупиков. Действительно, пусть ПР2 в некоторый момент времени выполнит операцию P(S). Тогда семафор S становится равным нулю. Пусть далее процесс ПР1 пытается выполнить операцию P(S). Процесс ПР1 в этом случае блокируется на семафоре S, так как S=0, причем значение S не изменится. После выполнения своей критической секции процесс ПР2 выполнит операцию V(S), при этом значение семафора S станет равным единице, а процесс ПР1 переведется в очередь готовности. Если через некоторое время процесс ПР1 продолжит свое исполнение, он успешно выполнит примитив P(S) и войдет в свою критическую секцию.

В однопроцессорной вычислительной системе неделимость операций P и V можно обеспечить с помощью простого запрета прерываний. Сам же семафор S можно реализовать в виде записи с двумя полями (листинг 7.9.). В одном поле будет храниться целое значение S, во втором — указатель на список процессов, заблокированных на семафоре S.

Листинг 7.9. Реализация операций P и V для однопроцессорной системы

```
type Semaphore = record
        счетчик   :integer;
        указатель :pointer;
```

продолжение ⇩

Листинг 7.9 (продолжение)

```
        end;
var S :Semaphore;

procedure P ( var S : Semaphore);
begin  ЗАПРЕТИТЬ_ПРЕРЫВАНИЯ;
  S.счетчик:= S.счетчик-1;
  if S.счетчик < 0 then
      WAIT(S); { вставить обратившийся процесс в список по S.указатель и передать
                 на процессор готовый к выполнению процесс }
  РАЗРЕШИТЬ_ПРЕРЫВАНИЯ
end;

procedure V ( var S : Semaphore);
begin  ЗАПРЕТИТЬ ПРЕРЫВАНИЯ;
  S.счетчик:= S.счетчик+1;
  if S.счетчик <= 0 then
    RELEASE (S);   { деблокировать первый процесс из списка по S.указатель }
  РАЗРЕШИТЬ_ПРЕРЫВАНИЯ
end;

procedure InitSem (var S : Semaphore);
begin
  S.счетчик:=1;
  S.указатель:=nil;
end;
```

Реализация семафоров в мультипроцессорных системах сложнее, чем в однопроцессорных. Одновременный доступ к семафору S двух процессов, выполняющихся на однопроцессорной вычислительной системе, предотвращается запретом прерываний. Однако этот механизм не подходит для мультипроцессорных систем, так как он не препятствует двум или более процессам одновременно обращаться к одному семафору. В силу того что такой доступ должен реализовываться через критическую секцию, необходимо дополнительное аппаратное взаимное исключение доступа для различных процессоров. Одним из решений является использование уже знакомых нам неделимых команд проверки и установки (TS). Двухкомпонентный семафор в этом случае расширяется включением третьего компонента — логического признака **взаимоискл** (листинг 7.10).

Листинг 7.10. Реализация операций P и V для мультипроцессорной системы

```
type Semaphore = record
          счетчик : integer;
          указатель : pointer;
          взаимоискл : boolean;
        end;
var S : Semaphore;

procedure InitSem (var S : Semaphore);
begin
With S do
  begin
    счетчик:=1;
    указатель:=nil;
    взаимоискл:=true;
  end;
end;
```

```
procedure P ( var S : Semaphore);
var разрешено : boolean;
begin
  ЗАПРЕТИТЬ_ПРЕРЫВАНИЯ;
  repeat TS(разрешено, S.взаимоискл) until разрешено;
  S.счетчик:=S.счетчик-1;
  if S.счетчик < 0 then WAIT(S); { вставить обратившийся процесс в список по S.указатель
                                   и передать на процессор готовый к выполнению процесс }
  S.взаимоискл:=true;
  РАЗРЕШИТЬ_ПРЕРЫВАНИЯ
end;

procedure V ( var S : Semaphore );
var разрешено : boolean;
begin
  ЗАПРЕТИТЬ_ПРЕРЫВАНИЯ;
  repeat TS(разрешено,S.взаимоискл) until разрешено;
  S.счетчик:=S.счетчик+1;
  if S.счетчик <= 0 then RELEASE(S);   { деблокировать первый процесс из списка
                                         по S.указатель }
  S.взаимоискл:=true;
  РАЗРЕШИТЬ_ПРЕРЫВАНИЯ;
end;
```

Обратите внимание, что в данном тексте команда проверки и установки — **TS(разрешено,S.взаимоискл)** — работает не с целочисленными, а с булевыми значениями. Практически это ничего не меняет, ибо текст программы и ее машинная (двоичная) реализация — это разные вещи.

Мьютексы

Одним из вариантов реализации семафорных механизмов для организации взаимного исключения является так называемый *мьютекс* (mutex). Термин «mutex» произошел от словосочетания «mutual exclusion semaphore», что дословно переводится с английского как «семафор взаимного исключения». Мьютексы реализованы во многих операционных системах, их основное назначение — организация взаимного исключения для задач (потоков выполнения) одного или нескольких процессов. Мьютексы — это простейшие двоичные семафоры, которые могут находиться в одном из двух состояний — отмеченном и неотмеченном (открыт и закрыт соответственно). Когда какая-либо задача, принадлежащая любому процессу, становится владельцем объекта мьютекс, последний переводится в неотмеченное состояние. Если задача освобождает мьютекс, его состояние становится отмеченным.

Организация последовательного (а не параллельного) доступа к ресурсам с использованием мьютексов становится несложной, поскольку в каждый конкретный момент только одна задача может владеть этим объектом. Для того чтобы мьютекс стал доступен задачам (потокам выполнения), принадлежащим разным процессам, при создании ему необходимо присвоить имя, впоследствии передаваемое «по наследству» задачам, которые должны его использовать для взаимодействия. Для этого вводятся специальные системные вызовы (**CreateMutex**), в которых указываются начальное значение мьютекса, его имя и, возможно, атри-

буты защиты. Если начальное значение мьютекса равно true, считается, что задача, создающая этот объект, сразу будет им владеть. Можно указать в качестве начального значение false — в этом случае мьютекс не будет принадлежать ни одной из задач, и только специальным обращением к нему удастся изменить его состояние.

Для работы с мьютексом имеется несколько функций. Помимо уже упомянутой функции создания такого объекта (CreateMutex), есть функции открытия (OpenMutex), ожидания событий (WaitForSingleObject и WaitForMultipleObjects) и, наконец, освобождения этого объекта (ReleaseMutex).

Конкретные обращения к этим функциям и перечни передаваемых и получаемых параметров имеются в документации на соответствующую операционную систему.

Использование семафоров при проектировании взаимодействующих вычислительных процессов

Семафорные примитивы чрезвычайно широко используются при проектировании разнообразных вычислительных процессов. При этом некоторые задачи являются настолько «типичными», что их детальное рассмотрение уже стало классическим в соответствующих учебных пособиях. Не будем делать исключений и мы.

Задача «поставщик–потребитель»

Решение задачи «поставщик–потребитель» является характерным примером использования семафорных операций. Содержательная постановка этой задачи уже была нами описана в начале этой главы. Разделяемыми переменными здесь являются счетчики свободных и занятых буферов, которые должны быть защищены со стороны обоих процессов, то есть действия по посылке и получению сообщений должны быть синхронизированы.

Использование семафоров для решения данной задачи иллюстрирует листинг 7.11.

Листинг 7.11. Решение задачи «поставщик–потребитель»

```
var S_свободно,S_заполнено,S_взаимоискл : semaphore;
begin
  InitSem(S_свободно,N);
  InitSem(S_заполнено,0);
  InitSem(S_взаимоискл,1);
parbegin
  ПОСТАВЩИК: while true do
          begin
            ...      { подготовить сообщение }
            P(S_свободно);
            P(S_взаимоискл);
            ...      { послать сообщение }
            V(S_заполнено);
            V(S_взаимоискл);
          end
and
```

```
    ПОТРЕБИТЕЛЬ: while true do
            begin
              P(S_заполнено);
              P(S_взаимоискл);
              ...    { получить сообщение }
              V(S_свободно);
              V(S_взаимоискл);
              ...    { обработать сообщение }
            end
parend
end.
```

Здесь переменные S_свободно, S_заполнено являются числовыми семафорами, S_взаимоискл — двоичный семафор. Переменная S_свободно имеет начальное значение, равное N, где N — количество буферов, с помощью которых процессы сотрудничают. Предполагается, что в начальный момент количество свободных буферов равно N; соответственно, количество занятых равно нулю. Двоичный семафор S_взаимоискл гарантирует, что в каждый момент только один процесс сможет работать с критическим ресурсом, выполняя свою критическую секцию. Семафоры S_свободно и S_заполнено используются как счетчики свободных и заполненных буферов.

Действительно, перед посылкой сообщения поставщик уменьшает значение S_свободно на единицу в результате выполнения операции P(S_свободно), а после посылки сообщения увеличивает значение S_заполнено на единицу в результате выполнения операции V(S_заполнено). Аналогично, перед получением сообщения потребитель уменьшает значение S_заполнено в результате выполнения операции P(S_заполнено), а после получения сообщения увеличивает значение S_свободно в результате выполнения операции V(S_свободно). Семафоры S_заполнено, S_свободно могут также использоваться для блокировки соответствующих процессов. Если пул буферов оказывается пустым, и к нему первым обратится процесс «потребитель», он заблокируется на семафоре S_заполнено в результате выполнения операции P(S_заполнено). Если пул буферов заполнится и к нему обратится процесс «поставщик», то он будет заблокирован на семафоре S_свободно в результате выполнения операции P(S_свободно).

В решении задачи о поставщике и потребителе общие семафоры применены для учета свободных и заполненных буферов. Их можно также применить и для распределения иных ресурсов.

Синхронизация взаимодействующих процессов с помощью семафоров

Можно использовать семафорные операции для решения таких задач, в которых успешное завершение одного процесса связано с ожиданием завершения другого. Предположим, что существуют два процесса ПР1 и ПР2. Необходимо, чтобы процесс ПР1 запускал процесс ПР2 с ожиданием его выполнения, то есть ПР1 не будет продолжать свое выполнение до тех пор, пока процесс ПР2 до конца не выполнит свою работу. Программа, реализующая такое взаимодействие, представлена в листинге 7.12.

Листинг 7.12. Пример синхронизации процессов

```
var S : Semaphore;
begin
  InitSem(S,0);

  ПР1: begin
       ПР11; { первая часть ПР1 }
       ON ( ПР2 ); { поставить на выполнение ПР2 }
       P(S);
       ПР12; { оставшаяся часть ПР1 }
       STOP
     end;


  ПР2:   begin
       ПР2; { вся работа программы ПР2 }
       V(S);
       STOP
     end
end
```

Начальное значение семафора S равно нулю. Если процесс ПР1 начал выполняться первым, то через некоторое время он поставит на выполнение процесс ПР2, после чего выполнит операцию P(S) и «заснет» на семафоре, перейдя в состояние пассивного ожидания. Процесс ПР2, осуществив все необходимые действия, выполнит примитив V(S) и откроет семафор, после чего процесс ПР1 будет готов к дальнейшему выполнению.

Задача «читатели–писатели»

Другой важной и часто встречающейся задачей, решение которой также требует синхронизации, является задача «читатели–писатели». Эта задача имеет много вариантов. Наиболее характерная область ее использования — построение систем управления файлами и базами данных, информационно-справочных систем. Два класса процессов имеют доступ к некоторому ресурсу (области памяти, файлам). «Читатели» — это процессы, которые могут параллельно считывать информацию из некоторой общей области памяти, являющейся критическим ресурсом. «Писатели» — это процессы, записывающие информацию в эту область памяти, исключая друг друга, а также процессы «читатели». Имеются различные варианты взаимодействия между писателями и читателями. Наиболее широко распространены следующие условия.

Устанавливается приоритет в использование критического ресурса процессам «читатели». Это означает, что если хотя бы один читатель пользуется ресурсом, то он закрыт для всех писателей и доступен для всех читателей. Во втором варианте, наоборот, больший приоритет у процессов «писатели». При появлении запроса от писателя необходимо закрыть дальнейший доступ всем тем читателям, которые запросят критический ресурс после него.

Помимо системы управления файлами другим типичным примером решения задачи «читатели–писатели» может служить система автоматизированной продажи билетов. Процессы «читатели» обеспечивают нас справочной информацией о наличии свободных билетов на тот или иной рейс. Процессы «писатели» запускают-

ся с пульта кассира, когда он оформляет для нас тот или иной билет. Имеется большое количество как читателей, так и писателей.

Пример программы, реализующей решение данной задачи в первой постановке, представлен в листинге 7.13. Процессы «читатели» и «писатели» описаны в виде соответствующих процедур.

Листинг 7.13. Решение задачи «читатели–писатели» с приоритетом в доступе к критическому ресурсу читателей

```
var  R, W : semaphore;
     N_R : integer;
procedure ЧИТАТЕЛЬ;
begin
  P(R);
  Inc(NR);      { NR:=NR +1 }
  if NR = 1 then P(W);
  V(R);
  Read_Data; { критическая секция }
  P(R);
  Dec(NR);
  if N_R = 0 then V(W);
  V(R)
end;

procedure ПИСАТЕЛЬ;
begin
  P(W);
  Write_Data; { критическая секция }
  V(W)
end

begin
  NR:=0;
  InitSem(S,1); InitSem(W,1);
  parbegin
    while true do ЧИТАТЕЛЬ
  and
    while true do ЧИТАТЕЛЬ
  and
    ...
    ...
    while true do ЧИТАТЕЛЬ
  and
    while true do ПИСАТЕЛЬ
  and
    while true do ПИСАТЕЛЬ
  and
    ...
    ...
    while true do ПИСАТЕЛЬ
  parend
end.
```

При решении данной задачи используются два семафора R и W, а также переменная NR, предназначенная для подсчета текущего числа процессов типа «читатели», находящихся в критической секции. Доступ к разделяемой области памяти осу-

ществляется через семафор W. Семафор R требуется для взаимного исключения процессов типа «читатели».

Если критический ресурс не используется, то первый появившийся процесс при входе в критическую секцию выполнит операцию P(W) и закроет семафор. Если процесс является читателем, то переменная NR увеличится на единицу, и последующие читатели будут обращаться к ресурсу, не проверяя значения семафора W, что обеспечит параллельность их доступа к памяти. Последний читатель, покидающий критическую секцию, является единственным, кто выполнит операцию V(W) и откроет семафор W. Семафор R предохраняет от некорректного изменения значения NR, а также от выполнения читателями операций P(W) и V(W). Если в критической секции находится писатель, то на семафоре W может быть заблокирован только один читатель, все остальные будут блокироваться на семафоре R. Другие писатели блокируются на семафоре W.

Когда писатель выполняет операцию V(W), неясно, какого типа процесс войдет в критическую секцию. Чтобы гарантировать получение читателями наиболее свежей информации, необходимо при постановке в очередь готовности использовать дисциплину обслуживания, учитывающую более высокий приоритет писателей. Однако этого оказывается недостаточно, ибо если в критической секции продолжает находиться по крайней мере один читатель, то он не даст обновить данные, но и не воспрепятствует вновь приходящим процессам «читателям» войти в свою критическую секцию. Необходим дополнительный семафор. Пример правильного решения этой задачи приведен в листинге 7.14.

Листинг 7.14. Решение задачи «читатели–писатели» с приоритетом в доступе к критическому ресурсу писателей

```
var   S, W, R : semaphore;
   NR : integer;
procedure ЧИТАТЕЛЬ;
begin
  P(S); P(R);
  Inc(NR);
  if NR = 1 then P(W);
  V(S); V(R);
  Read_Data; { критическая секция }
  P(R);
  Dec(NR);
  if NR = 0 then V(W);
  V(R)
end;

procedure ПИСАТЕЛЬ;
begin
  P(S); P(W);
  Write_Data; { критическая секция }
  V(S); V(W)
end;

begin
  NR:=0;
  InitSem(S,1); InitSem(W,1); InitSem(R,1);
  parbegin
```

```
    while true do ЧИТАТЕЛЬ
  and
    while true do ЧИТАТЕЛЬ
  and
    ...
    ...
    while true do ЧИТАТЕЛЬ
  and
    while true do ПИСАТЕЛЬ
  and
    while true do ПИСАТЕЛЬ
  and
    ...
    ...
    while true do ПИСАТЕЛЬ
  parend
end.
```

Как можно заметить, семафор S блокирует приход новых читателей, если появился хотя бы один писатель. Обратите внимание, что в процедуре ЧИТАТЕЛЬ использование семафора S имеет место только при входе в критическую секцию. После выполнения чтения уже категорически нельзя использовать этот семафор, ибо он тут же заблокирует первого же читателя, если хотя бы один писатель захочет войти в свою критическую секцию. И получится так называемая тупиковая ситуация, ибо писатель не сможет войти в критическую секцию, поскольку в ней уже находится читатель. А читатель не сможет покинуть критическую секцию, потому что писатель желает войти в свою критическую секцию.

Обычно программы, решающие проблему «читатели–писатели», используют как семафоры, так и мониторные схемы с взаимным исключением, то есть такие, которые блокируют доступ к критическим ресурсам для всех остальных процессов, если один из них модифицирует значения общих переменных. Взаимное исключение требует, чтобы писатель ждал завершения всех текущих операций чтения. При условии, что писатель имеет более высокий приоритет, чем читатель, такое ожидание в ряде случаев весьма нежелательно. Кроме того, реализация принципа взаимного исключения в многопроцессорных системах может вызвать определенную избыточность. Поэтому схема, представленная в листинге 7.15 и применяемая иногда для решения задачи «читатели–писатели», в случае одного писателя допускает одновременное выполнение операций чтения и записи. После чтения данных процесс «читатель» проверяет, мог ли он получить неправильное значение, некорректные данные (вследствие того, что параллельно с ним процесс «писатель» мог их изменить), и если обнаруживает, что это именно так, то операция чтения повторяется.

Листинг 7.15. Синхронизация процессов «читатели» и «писатель» без взаимного исключения

```
Var V1, V2 : integer;

Procedure ПИСАТЕЛЬ;
Begin
  Inc(V1);
  Write_Data;
  V2:=V1
End;
```

продолжение ⇩

Листинг 7.15 (*продолжение*)

```
Procedure ЧИТАТЕЛЬ;
Var V: integer
Begin
  Repeat   V:= V2;
      Read_Data
  Until V1 = V
End;

Begin
  V1 := 0;
  V2 := 0;
  Parbegin
    while true do ЧИТАТЕЛЬ
  and
    while true do ЧИТАТЕЛЬ
  and
    ...
    ...
    while true do ЧИТАТЕЛЬ
  and
    while true do ПИСАТЕЛЬ
  parend
end.
```

Этот алгоритм использует для данных два номера версий, которым соответствуют переменные V1 и V2. Перед записью порции новых данных процесс «писатель» увеличивает на 1 значение переменной V1, а после записи — переменной V2. Читатель обращается к V2 перед чтением данных, а к V1 — после. Если при этом переменные V1 и V2 равны, то очевидно, что получена правильная версия данных. Если же данные обновлялись за время чтения, то операция повторяется. Этот алгоритм может быть использован в случае, если нежелательно заставлять процесс «писатель» ждать, пока читатели закончат операцию чтения, или если вероятность повторения операции чтения достаточно мала и обусловленное повторными операциями снижение эффективности системы меньше потерь, связанных с избыточностью решения с помощью взаимного исключения. Однако необходимо иметь в виду ненулевую вероятность зацикливания чтения при высокой интенсивности операций записи. Наконец, если само чтение представляет собой достаточно длительную операцию, то оператор V := V2 для процесса «читатель» может быть заменен следующим оператором:

```
Repeat V := V2 Until V1 = V
```

Это предотвратит выполнение читателем операции чтения, если писатель уже начал запись.

Мониторы Хоара

Анализ рассмотренных задач показывает, что, несмотря на очевидные достоинства (простота, независимость от количества процессов, отсутствие активного ожидания), семафорные механизмы имеют и ряд недостатков. Эти механизмы являются слишком примитивными, так как семафор не указывает непосредственно на синх-

ронизирующее условие, с которым он связан, или на критический ресурс. Поэтому при построении сложных схем синхронизации алгоритмы решения задач порой получаются весьма непростыми, ненаглядными и трудными для доказательства их правильности.

Необходимо иметь очевидные решения, которые позволят прикладным программистам без лишних усилий, связанных с доказательством правильности алгоритмов и отслеживанием большого числа взаимосвязанных объектов, создавать параллельные взаимодействующие программы. К таким решениям можно отнести так называемые мониторы, предложенные Хоаром [52].

В параллельном программировании *монитор* — это пассивный набор разделяемых переменных и повторно входимых процедур доступа к ним, которым процессы пользуются в режиме разделения, причем в каждый момент им может пользоваться только один процесс.

Рассмотрим, например, некоторый ресурс, который разделяется между процессами каким-либо планировщиком [17]. Каждый раз, когда процесс желает получить в свое распоряжение какие-то ресурсы, он должен обратиться к программе-планировщику. Этот планировщик должен иметь переменные, с помощью которых можно отслеживать, занят ресурс или свободен. Процедуру планировщика разделяют все процессы, и каждый процесс может в любой момент захотеть обратиться к планировщику. Но планировщик не в состоянии обслуживать более одного процесса одновременно. Такая процедура-планировщик и представляет собой пример монитора.

Таким образом, монитор — это механизм организации параллелизма, который содержит как данные, так и процедуры, необходимые для динамического распределения конкретного общего ресурса или группы общих ресурсов. Процесс, желающий получить доступ к разделяемым переменным, должен обратиться к монитору, который либо предоставит доступ, либо откажет в нем. Необходимость входа в монитор с обращением к какой-либо его процедуре (например, с запросом на выделение требуемого ресурса) может возникать у многих процессов. Однако вход в монитор находится под жестким контролем — здесь осуществляется взаимное исключение процессов, так что в каждый момент времени только одному процессу разрешается войти в монитор. Процессам, которые хотят войти в монитор, когда он уже занят, приходится ждать, причем режимом ожидания автоматически управляет сам монитор. При отказе в доступе монитор блокирует обратившийся к нему процесс и определяет условие ожидания. Проверка условия выполняется самим монитором, который и деблокирует ожидающий процесс. Поскольку механизм монитора гарантирует взаимное исключение процессов, исключаются серьезные проблемы, связанные с организацией параллельных взаимодействующих процессов.

Внутренние данные монитора могут быть либо глобальными (относящимися ко всем процедурам монитора), либо локальными (относящимися только к одной конкретной процедуре). Ко всем этим данным можно обращаться только изнутри монитора; процессы, находящиеся вне монитора и, по существу, только вызывающие его процедуры, просто не могут получить доступ к данным монитора. При

первом обращении монитор присваивает своим переменным начальные значения. При каждом последующем обращении используются те значения переменных, которые остались от предыдущего обращения.

Если процесс обращается к некоторой процедуре монитора, а соответствующий ресурс уже занят, эта процедура выдает команду ожидания WAIT с указанием условия ожидания. Процесс мог бы оставаться внутри монитора, однако, если в монитор затем войдет другой процесс, это будет противоречить принципу взаимного исключения. Поэтому процесс, переводящийся в режим ожидания, должен вне монитора ждать того момента, когда необходимый ему ресурс освободится.

Со временем процесс, который занимал данный ресурс, обратится к монитору, чтобы возвратить ресурс системе. Соответствующая процедура монитора при этом может просто принять уведомление о возвращении ресурса, а затем ждать, пока не поступит запрос от другого процесса, которому потребуется этот ресурс. Однако может оказаться, что уже имеются процессы, ожидающие освобождения данного ресурса. В этом случае монитор выполняет команду извещения (сигнализации) SIGNAL, чтобы один из ожидающих процессов мог получить данный ресурс и покинуть монитор. Если процесс сигнализирует о возвращении (иногда называемом освобождением) ресурса и в это время нет процессов, ожидающих данного ресурса, то подобное оповещение не вызывает никаких других последствий, кроме того, что монитор, естественно, вновь внесет ресурс в список свободных. Очевидно, что процесс, ожидающий освобождения некоторого ресурса, должен находиться вне монитора, чтобы другой процесс имел возможность войти в монитор и возвратить ему этот ресурс.

Чтобы гарантировать, что процесс, находящийся в ожидании некоторого ресурса, со временем получит этот ресурс, считается, что ожидающий процесс имеет более высокий приоритет, чем новый процесс, пытающийся войти в монитор. В противном случае новый процесс мог бы перехватить ожидаемый ресурс до того, как ожидающий процесс вновь войдет в монитор. Если допустить многократное повторение подобной нежелательной ситуации, то ожидающий процесс мог бы откладываться бесконечно. Для систем реального времени можно допустить использование дисциплины обслуживания на основе абсолютных или динамически изменяемых приоритетов.

В качестве примера рассмотрим простейший монитор для выделения одного ресурса (листинг 7.16).

Листинг 7.16. Пример монитора Хоара

```
monitor Resourse;
condition free; { условие - свободный }
var busy : boolean; { занят }

procedure REQUEST; { запрос }
begin
    if busy then WAIT ( free );
    busy:=true;
    TakeOff; { выдать ресурс }
end;
```

```
procedure RELEASE;
begin
    TakeOn; { взять ресурс }
    busy:=false;
    SIGNAL ( free )
end;

begin
  busy:=false;
end
```

Единственный ресурс динамически запрашивается и освобождается процессами, которые обращаются к процедурам REQUEST (запрос) и RELEASE (освободить). Если процесс обращается к процедуре REQUEST в тот момент, когда ресурс используется, значение переменной busy (занято) будет равно true, и процедура REQUEST выполнит операцию монитора WAIT(free). Эта операция блокирует не процедуру REQUEST, а обратившийся к ней процесс, который помещается в конец очереди процессов, ожидающих, пока не будет выполнено условие free (свободно).

Когда процесс, использующий ресурс, обращается к процедуре RELEASE, операция монитора SIGNAL деблокирует процесс, находящийся в начале очереди, не позволяя исполняться никакой другой процедуре внутри того же монитора. Этот деблокированный процесс будет готов возобновить исполнение процедуры REQUEST сразу же после операции WAIT(free), которая его и блокировала. Если операция SIGNAL(free) выполняется в то время, когда нет процесса, ожидающего условия free, то никаких действий не выполняется.

Использование монитора в качестве основного средства синхронизации и связи освобождает процессы от необходимости явно разделять между собой информацию. Напротив, доступ к разделяемым переменным всегда ограничен телом монитора, и, поскольку мониторы входят в состав ядра операционной системы, разделяемые переменные становятся системными переменными. Это автоматически исключает необходимость в критических секциях (так как в каждый момент монитором может пользоваться только один процесс, то два процесса никогда не смогут получить доступ к разделяемым переменным одновременно).

Монитор является пассивным объектом в том смысле, что это не процесс; его процедуры выполняются только по требованию процесса.

Хотя по сравнению с семафорами мониторы не представляют собой существенно более мощного инструмента для организации параллельных взаимодействующих вычислительных процессов, у них есть некоторые преимущества перед более примитивными синхронизирующими средствами. Во-первых, мониторы очень гибки. В форме мониторов можно реализовать не только семафоры, но и многие другие синхронизирующие операции. Например, разобранный в разделе «Средства синхронизации и связи взаимодействующих вычислительных процессов» механизм решения задачи «поставщик–потребитель» легко запрограммировать в виде монитора. Во-вторых, локализация всех разделяемых переменных внутри тела монитора позволяет избавиться от малопонятных конструкций в синхронизируемых процессах — сложные взаимодействия процессов можно синхронизировать наглядным образом. В-третьих, мониторы дают процессам возможность совместно ис-

пользовать программные модули, представляющие собой критические секции. Если несколько процессов совместно используют ресурс и работают с ним совершенно одинаково, то в мониторе достаточно только одной процедуры, тогда как решение с семафорами требует, чтобы в каждом процессе имелся собственный экземпляр критической секции. Таким образом, мониторы по сравнению с семафорами позволяют значительно упростить организацию взаимодействующих вычислительных процессов и дают большую наглядность при совсем незначительной потере в эффективности.

Почтовые ящики

Тесное взаимодействие между процессами предполагает не только синхронизацию — обмен временными сигналами, но также передачу и получение произвольных данных, то есть обмен сообщениями. В системе с одним процессором посылающий и получающий процессы не могут работать одновременно. В мультипроцессорных системах также нет никакой гарантии их одновременного исполнения. Следовательно, для хранения посланного, но еще не полученного сообщения необходимо место. Оно называется *буфером сообщений*, или *почтовым ящиком*[1].

Если процесс P1 хочет общаться с процессом P2, то P1 просит систему предоставить или образовать почтовый ящик, который свяжет эти два процесса так, чтобы они могли передавать друг другу сообщения. Для того чтобы послать процессу P2 какое-то сообщение, процесс P1 просто помещает это сообщение в почтовый ящик, откуда процесс P2 может его в любое время получить. При применении почтового ящика процесс P2 в конце концов обязательно получит сообщение, когда обратится за ним (если вообще обратится). Естественно, что процесс P2 должен знать о существовании почтового ящика. Поскольку в системе может быть много почтовых ящиков, необходимо обеспечить доступ процессу к конкретному почтовому ящику. Почтовые ящики являются системными объектами, и для пользования таким объектом необходимо получить его у операционной системы, что осуществляется с помощью соответствующих запросов.

Если объем передаваемых данных велик, то эффективнее не передавать их непосредственно, а отправлять в почтовый ящик сообщение, информирующее процесс-получатель о том, где можно их найти.

Почтовый ящик может быть связан с парой процессов, только с отправителем, только с получателем, или его можно получить из множества почтовых ящиков, которые используют все или несколько процессов. Почтовый ящик, связанный с процессом-получателем, облегчает посылку сообщений от нескольких процессов в фиксированный пункт назначения. Если почтовый ящик не связан жестко с процессами, то сообщение должно содержать идентификаторы и процесса-отправителя, и процесса-получателя.

---

[1] Название «почтовый ящик» происходит от обычного приспособления для отправки почты.

Итак, почтовый ящик — это информационная структура, поддерживаемая операционной системой. Она состоит из головного элемента, в котором находится информация о данном почтовом ящике, и нескольких буферов (гнезд), в которые помещают сообщения. Размер каждого буфера и их количество обычно задаются при образовании почтового ящика.

Правила работы почтового ящика могут быть различными в зависимости от его сложности [17]. В простейшем случае сообщения передаются только в одном направлении. Процесс P1 может посылать сообщения до тех пор, пока имеются свободные гнезда. Если все гнезда заполнены, то P1 может либо ждать, либо заняться другими делами и попытаться послать сообщение позже. Аналогично процесс P2 может получать сообщения до тех пор, пока имеются заполненные гнезда. Если сообщений нет, то он может либо ждать сообщений, либо продолжать свою работу. Эту простую схему работы почтового ящика можно усложнять в нескольких направлениях и получать более хитроумные системы общения — двунаправленные и многовходовые почтовые ящики.

Двунаправленный почтовый ящик, связанный с парой процессов, позволяет подтверждать прием сообщений. При наличии множества гнезд каждое из них хранит либо сообщение, либо подтверждение. Чтобы гарантировать передачу подтверждений, когда все гнезда заняты, подтверждение на сообщение помещается в то же гнездо, в котором находится сообщение, и это гнездо уже не используется для другого сообщения до тех пор, пока подтверждение не будет получено. Из-за того, что некоторые процессы не забрали свои сообщения, связь может быть приостановлена. Если каждое сообщение снабдить пометкой времени, то управляющая программа может периодически удалять старые сообщения.

Процессы могут быть также остановлены в связи с тем, что другие процессы не смогли послать им сообщения. Если время поступления каждого остановленного процесса в очередь заблокированных процессов регистрируется, то управляющая программа может периодически посылать им пустые сообщения, чтобы они не ждали чересчур долго.

Реализация почтовых ящиков требует использования примитивных операторов низкого уровня, таких как операции **P** и **V** или каких-либо других, но пользователям может дать средства более высокого уровня (наподобие мониторов Хоара), например, такие, как представлены ниже.

`SEND_MESSAGE ( Получатель, Сообщение, Буфер )`

Эта операция переписывает сообщение в некоторый буфер, помещает его адрес в переменную **Буфер** и добавляет буфер к очереди **Получатель**. Процесс, выдавший операцию **SEND_MESSAGE**, продолжит свое исполнение.

`WAIT_MESSAGE ( Отправитель, Сообщение, Буфер )`

Эта операция блокирует процесс, выдавший операцию, до тех пор, пока в его очереди не появится какое-либо сообщение. Когда процесс передается на процессор, он получает имя отправителя с помощью переменной **Отправитель**, текст сообщения через переменную **Сообщение** и адрес буфера в переменной **Буфер**. Затем буфер удаляется из очереди, и процесс может записать в него ответ отправителю.

`SEND_ANSWER ( Результат, Ответ, Буфер )`

Эта операция записывает информацию, определяемую через переменную **Ответ**, в тот буфер, номер которого указывается переменной **Буфер** (из этого буфера было получено сообщение), и добавляет буфер к очереди отправителя. Если отправитель ждет ответ, он деблокируется.

`WAIT_ANSWER ( Результат, Ответ, Буфер )`

Эта операция блокирует процесс, выдавший операцию, до тех пор, пока в буфер не поступит ответ; доступ к нему возможен через переменную **Буфер**. После того как ответ поступил и процесс передан на процессор, ответ, доступ к которому определяется через переменную **Ответ**, переписывается в память процессу, а буфер освобождается. Значение переменной **Результат** указывает, является ли ответ пустым, то есть выданным операционной системой, так как сообщение было адресовано несуществующему (или так и не ставшему активным) процессу.

Основные достоинства почтовых ящиков:

- процессу не нужно знать о существовании других процессов до тех пор, пока он не получит сообщения от них;
- два процесса могут обменяться более чем одним сообщением за один раз;
- операционная система может гарантировать, что никакой иной процесс не вмешается во взаимодействие процессов, ведущих между собой «переписку»;
- очереди буферов позволяют процессу-отправителю продолжать работу, не обращая внимания на получателя.

Основным недостатком буферизации сообщений является появление еще одного ресурса, которым нужно управлять. Этим ресурсом являются сами почтовые ящики.

К другому недостатку можно отнести статический характер этого ресурса: количество буферов для передачи сообщений через почтовый ящик фиксировано. Поэтому естественным стало появление механизмов, подобных почтовым ящикам, но реализованных на принципах динамического выделения памяти под передаваемые сообщения.

В операционных системах компании Microsoft тоже имеются почтовые ящики (mailslots). В частности, они достаточно часто используются при создании распределенных приложений для сети. При работе с ними в приложении, которое должно отправить сообщение другому приложению, необходимо указывать класс доставки сообщений. Различают два класса доставки. Первый класс (first-class delivery) гарантирует доставку сообщений; он ориентирован на сеансовое взаимодействие между процессами и позволяет организовать посылки типа «один к одному» и «один ко многим». Второй класс (second-class delivery) основан на механизме датаграмм, и он уже не гарантирует доставку сообщений получателю.

Конвейеры и очереди сообщений

Конвейеры

Программный *канал связи* (pipe), или, как его иногда называют, *конвейер*, *транспортер*, является средством, с помощью которого можно обмениваться данными

между процессами. Принцип работы конвейера основан на механизме ввода-вывода файлов в UNIX, то есть задача, передающая информацию, действует так, как будто она записывает данные в файл, в то время как задача, для которой предназначается эта информация, читает ее из этого файла. Операции записи и чтения осуществляются не записями, как это делается в обычных файлах, а потоком байтов, как это принято в UNIX-системах. Таким образом, функции, с помощью которых выполняется запись в канал и чтение из него, являются теми же самыми, что и при работе с файлами. По сути, канал представляет собой поток данных между двумя (или более) процессами. Это упрощает программирование и избавляет программистов от использования каких-то новых механизмов. На самом деле конвейеры не являются файлами на диске, а представляют собой буферную память, работающую по принципу FIFO, то есть по принципу обычной очереди. Однако не следует путать конвейеры с очередями сообщений; последние реализуются иначе и имеют другие возможности.

Конвейер имеет определенный размер[1], который не может превышать 64 Кбайт и работает циклически. Вспомните реализацию очереди на массивах, когда имеются указатели начала и конца очереди, которые перемещаются циклически по массиву. То есть имеется некий массив и два указателя: один показывает на первый элемент (указатель на начало — head), а второй — на последний (указатель на конец — tail).

В начальный момент оба указателя равны нулю. Добавление самого первого элемента в пустую очередь приводит к тому, что указатели на начало и на конец принимают значение, равное 1 (в массиве появляется первый элемент). В последующем добавление нового элемента вызывает изменение значения второго указателя, поскольку он отмечает расположение именно последнего элемента очереди. Чтение (и удаление) элемента (читается и удаляется всегда первый элемент из созданной очереди) приводит к необходимости модифицировать значение указателя на ее начало. В результате операций записи (добавления) и чтения (удаления) элементов в массиве, моделирующем очередь элементов, указатели будут перемещаться от начала массива к его концу. При достижении указателем значения индекса последнего элемента массива значение указателя вновь становится единичным (если при этом не произошло переполнение массива, то есть количество элементов в очереди не стало большим числа элементов в массиве). Можно сказать, что мы как бы замыкаем массив в кольцо, организуя круговое перемещение указателей на начало и на конец, которые отслеживают первый и последний элементы в очереди. Сказанное иллюстрирует рис. 7.4. Именно так функционирует конвейер.

Как информационная структура конвейер описывается идентификатором, размером и двумя указателями. Конвейеры представляют собой системный ресурс. Чтобы начать работу с конвейером, процесс сначала должен заказать его у операционной системы и получить в свое распоряжение. Процессы, знающие идентификатор конвейера, могут через него обмениваться данными.

---

[1] Механизм конвейеров, впервые введенный в UNIX-системах, имеет максимальный размер 64 Кбайт, поскольку в 16-разрядных мини-ЭВМ, для которых создавалась эта ОС, нельзя было иметь массив данных большего размера.

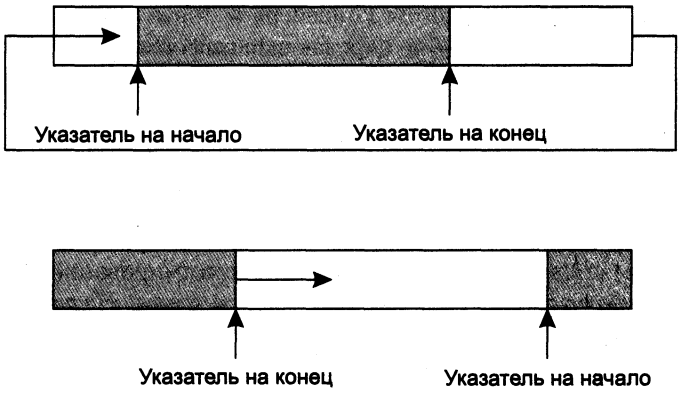


Рис. 7.4. Организация очереди в массиве

В качестве иллюстрации приведем основные системные запросы для работы с конвейерами, которые имеются в API OS/2.

- Функция создания конвейера:

  ```
  DosCreatePipe (&ReadHandle, &WriteHandle, PipeSize);
  ```

  Здесь **ReadHandle** — дескриптор чтения из конвейера, **WriteHandle** — дескриптор записи в конвейер, **PipeSize** — размер конвейера.

- Функция чтения из конвейера:

  ```
  DosRead (&ReadHandle, (PVOID)&Inform, sizeof(Inform), &BytesRead);
  ```

  Здесь **ReadHandle** — дескриптор чтения из конвейера, **Inform** — переменная любого типа, **sizeof(Inform)** — размер переменной **Inform**, **BytesRead** — количество прочитанных байтов. Данная функция при обращении к пустому конвейеру будет ожидать, пока в нем не появится информация для чтения.

- Функция записи в конвейер:

  ```
  DosWrite (&WriteHandle, (PVOID)&Inform, sizeof(Inform), &BytesWrite);
  ```

  Здесь **WriteHandle** — дескриптор записи в конвейер, **BytesWrite** — количество записанных байтов.

Читать из конвейера может только тот процесс, который знает идентификатор соответствующего конвейера. При работе с конвейером данные непосредственно помещаются в него. Еще раз отметим, что из-за ограничения на размер конвейера программисты сталкиваются и с ограничениями на размеры передаваемых через него сообщений.

Очереди сообщений

Очереди (queues) *сообщений* предлагают более удобный метод связи между взаимодействующими процессами по сравнению с каналами, но в своей реализации они сложнее. С помощью очередей также можно из одной или нескольких задач независимым образом посылать сообщения некоторой задаче-приемнику. При этом только процесс-приемник может читать и удалять сообщения из очереди, а про-

цессы-клиенты имеют право лишь помещать в очередь свои сообщения. Таким образом, очередь работает только в одном направлении. Если же необходима двухсторонняя связь, то можно создать две очереди.

Работа с очередями сообщений отличается от работы с конвейерами. Во-первых, очереди сообщений предоставляют возможность использовать несколько дисциплин обработки сообщений:

- FIFO — сообщение, записанное первым, будет первым и прочитано;
- LIFO — сообщение, записанное последним, будет прочитано первым;
- приоритетный доступ — сообщения читаются с учетом их приоритетов;
- произвольный доступ — сообщения читаются в произвольном порядке.

Тогда как канал обеспечивает только дисциплину FIFO.

Во-вторых, если при чтении сообщения оно удаляется из конвейера, то при чтении сообщения из очереди этого не происходит, и сообщение при желании может быть прочитано несколько раз.

В-третьих, в очередях присутствуют не непосредственно сами сообщения, а только их адреса в памяти и размер. Эта информация размещается системой в сегменте памяти, доступном для всех задач, общающихся с помощью данной очереди.

Каждый процесс, использующий очередь, должен предварительно получить разрешение на доступ в общий сегмент памяти с помощью системных запросов API, ибо очередь — это системный механизм, и для работы с ним требуются системные ресурсы и, соответственно, обращение к самой ОС. Во время чтения из очереди задача-приемник пользуется следующей информацией:

- идентификатор процесса (Process Identifier, PID), который передал сообщение;
- адрес и длина переданного сообщения;
- признак необходимости ждать, если очередь пуста;
- приоритет переданного сообщения;
- номер освобождаемого семафора, когда сообщение передается в очередь.

Наконец, приведем перечень основных функций, управляющих работой очереди (без подробного описания передаваемых параметров, поскольку в различных ОС обращения к этим функциям могут существенно различаться):

- `CreateQueue` — создание новой очереди;
- `OpenQueue` — открытие существующей очереди;
- `ReadQueue` — чтение и удаление сообщения из очереди;
- `PeekQueue` — чтение сообщения без его последующего удаления из очереди;
- `WriteQueue` — добавление сообщения в очередь;
- `CloseQueue` — завершение использования очереди;
- `PurgeQueue` — удаление из очереди всех сообщений;
- `QueryQueue` — определение числа элементов в очереди.

Контрольные вопросы и задачи

1. Какие последовательные вычислительные процессы мы называем параллельными и почему? Какие параллельные процессы называются независимыми, а какие — взаимодействующими?
2. Изложите алгоритм Деккера, позволяющий разрешить проблему взаимного исключения путем использования одной только блокировки памяти.
3. Объясните, как действует команда проверки и установки. Расскажите о работе команд BTS и BTR, которые имеются в процессорах с архитектурой ia32.
4. Расскажите о семафорах Дейкстры. Чем обеспечивается взаимное исключение при выполнении примитивов P и V?
5. Изложите, как могут быть реализованы семафорные примитивы для мультипроцессорной системы?
6. Что такое мьютекс?
7. Изложите алгоритм решения задачи «поставщик–потребитель» при использовании семафоров Дейкстры.
8. Изложите алгоритм решения задачи «читатели–писатели» при использовании семафоров Дейкстры.
9. Что такое «монитор Хоара»? Приведите пример такого монитора.
10. Что представляют собой почтовые ящики?
11. Что представляют собой конвейеры (программные каналы)?
12. Что представляют собой очереди сообщений? Чем отличаются очереди сообщений от почтовых ящиков?

Глава 8. Проблема тупиков и методы борьбы с ними

Рассмотрим одну из самых серьезных и трудноразрешимых проблем, возникающих при организации мультипрограммного режима работы, — проблему тупиков и основные подходы при борьбе с ними. В этой главе представлены некоторые модели параллельных вычислительных процессов, позволяющие проводить их анализ в аспекте корректного решения указанных проблем.

Понятие тупиковой ситуации при выполнении параллельных вычислительных процессов

При организации параллельного выполнения нескольких вычислительных процессов одной из главных функций операционной системы является решение сложной задачи корректного распределения ресурсов между выполняющимися процессами и обеспечение последних средствами взаимной синхронизации и обмена данными.

При параллельном исполнении процессов могут возникать тупиковые ситуации, когда два или более процесса блокируют друг друга, вынуждая ожидать наступления события, связанного с освобождением ресурса. Самым простым является случай, когда каждый из двух процессов ожидает ресурс, занятый другим процессом. Из-за такого ожидания ни один из процессов не может продолжить исполнение и освободить в конечном итоге ресурс, необходимый другому процессу. Эта ситуация называется *тупиком*, дедлоком (dead lock[1]), или *клинчем*. Говорят, что в мультипрограммной системе процесс находится в состоянии тупика, если он ждет события, которое никогда не произойдет. Тупики чаще всего возникают из-за конкуренции несвязанных параллельных процессов за ресурсы вычислительной системы, но иногда к тупикам приводят и ошибки программирования взаимодействующих вычислений.

---

[1] Dead lock (англ.) — смертельное объятие.

При рассмотрении проблемы тупиков целесообразно понятие ресурсов системы обобщить и разделить их все на два класса:

- *повторно используемые* (Reusable Resource, RR), или *системные* (System Resource, SR), *ресурсы*;
- *потребляемые, или расходуемые, ресурсы* (Consumable Resource, CR).

Системные ресурсы (SR) есть конечное множество идентичных единиц некоторого вида ресурсов, обладающих следующими свойствами [54]:

- число единиц ресурса в системе неизменно;
- каждая единица ресурса либо доступна, либо выделена одному и только одному процессу (разделение отсутствует или не принимается во внимание, так как не оказывает влияния на распределение ресурсов, а значит, и на возникновение тупиковой ситуации);
- процесс может освободить единицу ресурса (сделать ее доступной), только если он ранее получил эту единицу, то есть никакой процесс не может оказывать влияние на ресурс, если этот ресурс ему не принадлежит.

Данное определение выделяет существенные для изучения проблемы тупика свойства системных ресурсов, к которым мы относим компоненты аппаратуры, такие как основная память, вспомогательная (внешняя) память, периферийные устройства и, возможно, процессоры, а также программное и информационное обеспечение, такое как файлы данных, таблицы и «разрешение войти в критическую секцию».

Расходуемые ресурсы (CR) отличаются от ресурсов типа SR в нескольких важных отношениях [17].

- Число доступных единиц некоторого ресурса типа CR изменяется по мере того, как выполняющимися процессами они расходуются (приобретаются) и освобождаются (производятся). В общем случае число единиц расходуемых ресурсов является потенциально неограниченным, поскольку некий процесс «производитель» может достаточно долго увеличивать число единиц ресурса, освобождая одну или более единиц, которые он «создал».
- Процесс «потребитель» уменьшает число единиц ресурса, сначала запрашивая и затем приобретая (потребляя) одну или более единиц. Единицы ресурса, которые приобретены, в общем случае не возвращаются ресурсу, а потребляются (расходуются). Эти свойства потребляемых ресурсов присущи многим синхронизирующим сигналам, сообщениям и данным, порождаемым как аппаратурой, так и программным обеспечением, и могут рассматриваться как ресурсы типа CR при изучении тупиков. В их число входят: прерывания от таймера и устройств ввода-вывода; сигналы синхронизации процессов; сообщения, содержащие запросы на различные виды обслуживания или данные, а также соответствующие ответы.

Для исследования параллельных процессов и, в частности, проблемы тупиков было разработано несколько моделей. Одной из них является *модель повторно используемых ресурсов Холта* [54]. Согласно этой модели система представляется как набор (множество) процессов и набор ресурсов, причем каждый из ресурсов состоит из

фиксированного числа единиц. Любой процесс может изменять состояние системы путем выдачи запроса на получение или освобождение единицы ресурса.

В графической форме процессы и ресурсы представляются квадратами и кружками соответственно. Каждый кружок содержит некоторое количество маркеров (фишек) в соответствии с существующим количеством единиц этого ресурса. Дуга, указывающая из «процесса» на «ресурс», означает запрос одной единицы этого ресурса. Дуга, указывающая из «ресурса» на «процесс», представляет выделение ресурса процессу. Поскольку каждая единица любого ресурса типа SR может быть выделена одновременно не более чем одному процессу, то число дуг, исходящих из ресурса к различным процессам, не может превышать общего числа единиц этого ресурса. Такая модель называется *графом повторно используемых ресурсов.*

Пример одного из состояний системы из двух процессов с ресурсами типа SR представлен на рис. 8.1.

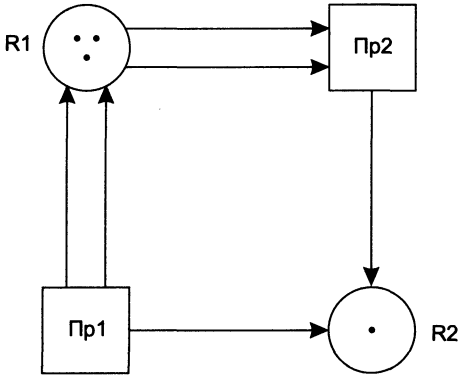


Рис. 8.1. Пример модели Холта

Пусть процесс Пр1 запрашивает две единицы ресурса R1 и одну единицу ресурса R2. Процессу Пр2 принадлежат две единицы ресурса R1, и ему нужна одна единица R2. Предположим, что процесс Пр1 получил запрошенную им единицу R2. Если принято правило, по которому процесс должен получить все запрошенные им ресурсы прежде, чем освободить хотя бы один из них, то удовлетворение запроса Пр1 приведет к тупиковой ситуации: Пр1 не сможет продолжиться до тех пор, пока Пр2 не освободит единицу ресурса R1, а процесс Пр2 не сможет продолжиться до тех пор, пока Пр1 не освободит единицу R2. Причиной этого тупика являются неупорядоченные попытки процессов войти в критическую секцию, связанную с выделением соответствующей единицы ресурса.

Примеры тупиковых ситуаций и причины их возникновения

Для понимания основных причин возникновения тупиков рассмотрим несколько простых характерных примеров.

Пример тупика на ресурсах типа CR

Пусть имеется три процесса Пр1, Пр2 и Пр3, которые вырабатывают сообщения М1, М2 и М3 соответственно. Эти сообщения представляют собой ресурсы типа CR. Пусть процесс Пр1 является потребителем сообщения М3, процесс Пр2 должен получить сообщение М1, а Пр3 ожидает сообщение М2 от процесса Пр2. Таким образом, каждый из этих трех процессов является и поставщиком, и потребителем одновременно, и вместе они образуют кольцевую систему (рис. 8.2) передачи сообщений через почтовые ящики (ПЯ).

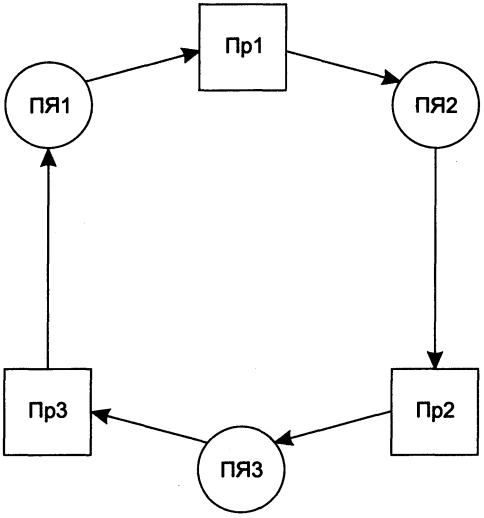


Рис. 8.2. Кольцевая схема взаимодействия процессов

Если связь с помощью этих сообщений со стороны каждого процесса устанавливается в порядке, представленном в листинге 8.1, то никаких проблем не возникает. Однако перестановка этих двух процедур в каждом из процессов вызывает тупик (листинг 8.2).

Листинг 8.1. Вариант псевдокода без тупиковой ситуации

```
Пр1:   ...
    ПОСЛАТЬ СООБЩЕНИЕ (Пр2, М1, ПЯ2);
    ЖДАТЬ СООБЩЕНИЕ (Пр3, М3, ПЯ1);
       ...

Пр2:   ...
    ПОСЛАТЬ СООБЩЕНИЕ (Пр3, М2, ПЯ3);
    ЖДАТЬ СООБЩЕНИЕ (Пр1, М1, ПЯ2);
       ...

Пр3:   ...
    ПОСЛАТЬ СООБЩЕНИЕ (Пр1, М3, ПЯ1);
    ЖДАТЬ СООБЩЕНИЕ (Пр2, М2, ПЯ3);
       ...
```

Листинг 8.2. Вариант псевдокода с тупиковой ситуацией

```
Пр1:   ...
       ЖДАТЬ СООБЩЕНИЕ (Пр3, М3, ПЯ1);
       ПОСЛАТЬ СООБЩЕНИЕ (Пр2, М1, ПЯ2);
          ...

Пр2:   ...
       ЖДАТЬ СООБЩЕНИЕ (Пр1, М1, ПЯ2);
       ПОСЛАТЬ СООБЩЕНИЕ (Пр3, М2, ПЯ3);
          ...

Пр3:   ...
       ЖДАТЬ СООБЩЕНИЕ (Пр2, М2, ПЯ3);
       ПОСЛАТЬ СООБЩЕНИЕ (Пр1, М3, ПЯ1);
          ...
```

В самом деле, во втором варианте ни один из процессов не сможет послать сообщение до тех пор, пока сам его не получит, а это событие никогда не произойдет, поскольку ни один процесс не может этого сделать.

Пример тупика на ресурсах типа CR и SR

Пусть некоторый процесс Пр1 должен обменяться сообщениями с процессом Пр2 и каждый из них запрашивает некоторый ресурс R, причем Пр1 требует три единицы этого ресурса для своей работы, а Пр2 — две единицы и только на время обработки сообщения. Всего же имеется только четыре единицы ресурса R. Запрос и освобождение ресурса можно реализовать через соответствующий монитор с процедурами REQUEST(R, N) — запрос N единиц ресурса R, и RELEASE(R, N) — освобождение (возврат) N единиц ресурса R. Обмен сообщениями будем осуществлять через почтовый ящик MB. Фрагменты программ Пр1 и Пр2 приведены в листинге 8.3.

Листинг 8.3. Пример тупика на ресурсах CR и SR

```
        ...
        ...
Пр1:    REQUEST ( R, 3 );
        ...
        ...
        SEND_MESSAGE ( Пр2, сообщение, MB );
        WAIT_ANSWER ( ответ, MB );
        ...
        ...
        RELEASE ( R, 3 );
        ...
        ...

-------------------------------------------------------

        ...
        ...
Пр2:    WAIT_MESSAGE ( Пр1, сообщение, MB );
        REQUEST ( R, 2 );
        ОБРАБОТКА СООБЩЕНИЯ;
        RELEASE ( R, 2 );
```

продолжение ⇩

Листинг 8.3 (*продолжение*)

```
SEND_ANSWER ( ответ, MB );
...
...
```

Увы, эти два процесса всегда будут попадать в состояние тупика. Действительно, процесс Пр2, выполняясь первым, сначала будет ожидать сообщения от процесса Пр1, после чего будет заблокирован при запросе ресурса R, часть которого окажется уже отданной процессу Пр1. Процесс Пр1, завладев частью ресурса R, будет заблокирован ожиданием ответа от Пр2, которого никогда не получит, так как для этого Пр2 нужно получить ресурс R, находящийся в распоряжении Пр1. Тупика можно избежать лишь при условии, что на время ожидания ответа от Пр2 процесс Пр1 отдаст хотя бы одну из единиц ресурса R, которыми он владеет. В данном примере, как и в предыдущем, причиной тупика являются ошибки программирования.

Пример тупика на ресурсах типа SR

Предположим, что существуют два процесса Пр1 и Пр2, разделяющих два ресурса типа SR: R1 и R2. Пусть взаимное исключение доступов к этим ресурсам реализуется с помощью семафоров S1 и S2 соответственно. Процессы Пр1 и Пр2 обращаются к ресурсам так, как показано на рис. 8.3 [17].

Процесс Пр 1		Процесс Пр 2	
	⋮		⋮
1:	P(S2);	(5):	P(S1);
	⋮		⋮
2:	P(S1);	(6):	P(S2);
	⋮		⋮
3:	V(S1);	(7):	V(S1);
	⋮		⋮
4:	V(S2);	(8):	V(S2);
	⋮		⋮

Рис. 8.3. Пример последовательности операторов для двух процессов, которые могут привести к тупиковой ситуации

Здесь несущественные детали (с точки зрения обращения к ресурсам) опущены. Считаем, что оба семафора первоначально установлены в единицу. Пространство возможных состояний приведено на рис. 8.4.

Горизонтальная ось задает выполнение процесса Пр1, вертикальная — процесса Пр2. Вертикальные линии, пронумерованные от 1 до 4, соответствуют операторам 1–4 процесса Пр1; аналогично горизонтальные линии, пронумерованные от 5 до 8, соответствуют операторам 5–8 программы Пр2. Точка на плоскости определяет состояние вычислений в некоторый момент времени. Так, точка A соответствует ситуации, при которой процесс Пр1 начал исполнение, но не достиг оператора 1, а процесс Пр2 выполнил оператор 6, но не дошел до оператора 7. По мере выпол-

нения точка будет двигаться горизонтально вправо, если исполняется процесс Пр1, и вертикально вверх, если исполняется процесс Пр2.

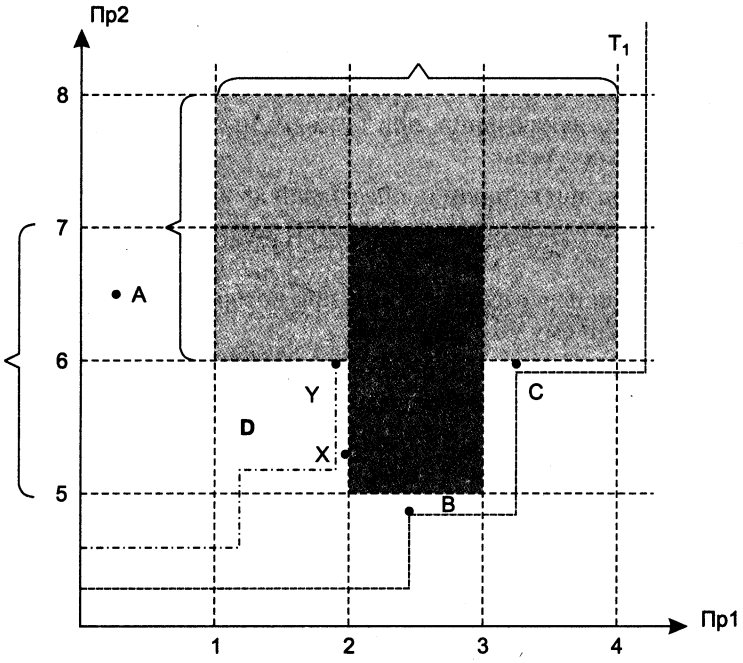


Рис. 8.4. Пространство состояний системы двух параллельных конкурирующих процессов

Интервалы исполнения, во время которых ресурсы R1 и R2 используются каждым процессом, показаны с помощью фигурных скобок. Линии 1–8 делят пространство вычислений на 25 областей, каждая из которых соответствует определенному состоянию в распределении ресурсов в процессе вычислений. Закрашенные серым цветом состояния являются недостижимыми из-за взаимного исключения процессов Пр1 и Пр2 при доступе к ресурсам R1 и R2.

Рассмотрим последовательность исполнения 1-2-5-3-6-4-7-8, представленную траекторией T1. Когда процесс Пр2 запрашивает ресурс R1 (оператор 5), ресурс недоступен (оператор выполнен, семафор закрыт). Поэтому процесс Пр2 заблокирован в точке B. Как только процесс Пр1 достигнет оператора 3, процесс Пр2 деблокируется по ресурсу R1. Аналогично в точке C процесс Пр2 будет заблокирован при попытке доступа к ресурсу R2 (оператор 6). Как только процесс Пр1 достигнет оператора 4, процесс Пр2 деблокируется по ресурсу R2.

Если же, например, выполняется последовательность 1-5-2-6, то процесс ПР1 заблокируется в точке X при выполнении оператора 2, а процесс Пр2 заблокируется в точке Y при выполнении оператора 6. При этом процесс ПР1 ждет, когда процесс Пр2 выполнит оператор 7, а Пр2 ждет, когда Пр1 выполнит оператор 4. Оба процесса будут находиться в тупике, ни Пр1, ни Пр2 не смогут закончить выполнение. При этом все ресурсы, которые получили оба процесса, становятся недоступными для других

процессов, что резко снижает возможности вычислительной системы по их обслуживанию. Отметим одно очень важное обстоятельство: тупик будет неизбежным, если вычисления зашли в прямоугольник D, являющийся опасным состоянием.

Исследования проблемы тупиков показали, что для возникновения тупиковой ситуации необходимо одновременное выполнение следующих четырех условий [17, 54]:

- условия взаимного исключения, при котором процессы осуществляют монопольный доступ к ресурсам;
- условия ожидания, при котором процесс, запросивший ресурс, ждет до тех пор, пока запрос не будет удовлетворен, при этом удерживая ранее полученные ресурсы;
- условия отсутствия перераспределения, при котором ресурсы нельзя отобрать у процесса, если они ему уже выделены;
- условия кругового ожидания, при котором существует замкнутая цепь процессов, каждый из которых ждет ресурс, удерживаемый его предшественником в цепи.

Проанализировав содержательный смысл этих четырех условий, легко убедиться, что все они выполняются в точке Y (см. рис. 8.4).

Формальные модели для изучения проблемы тупиковых ситуаций

Проблема борьбы с тупиками становится все более актуальной и сложной по мере развития и внедрения параллельных вычислительных систем и сетей. При проектировании таких систем разработчики стараются проанализировать возможные негативные ситуации, используя специальные модели и методы.

К настоящему времени разработано несколько десятков различных моделей, предназначенных для анализа и моделирования систем с параллельными асинхронными процессами, для которых возможность возникновения тупиковых ситуаций является очень серьезной проблемой. Изложение и сравнительный анализ этих моделей может составить большую монографию, поэтому здесь мы лишь кратко рассмотрим только три из них — сети Петри, модель пространства состояний и уже упомянутую нами модель Холта.

Сети Петри

Среди многих существующих методов описания и анализа параллельных систем уже более 35 лет значительное место занимают сетевые модели, восходящие к сетям специального вида, предложенным в 1962 году Карлом Петри для моделирования асинхронных информационных потоков в системах преобразования данных [36].

Взаимодействие событий в параллельных асинхронных дискретных системах имеет, как правило, сложную динамическую структуру. Эти взаимодействия описываются проще, если указывать не непосредственные связи между событиями, а те ситуации, при которых данное событие может реализоваться. При этом глобаль-

ные ситуации в системе формируются с помощью локальных операций, называемых условиями возникновения событий. Определенные сочетания условий допускают возникновение некоторого события (*предусловия события*), а реализация события изменяет некоторые условия (*постусловия события*), то есть события взаимодействуют с условиями, а условия — с событиями. Таким образом, предполагается, что для решения задач достаточно представить системы как структуры, образованные из элементов двух типов: событий и условий. Удобное обобщение этого, предложенное Петри, было развито А. Холтом, который назвал его *сетью Петри*.

В сетях Петри события и условия представлены абстрактными символами из двух непересекающихся алфавитов, называемых соответственно множеством переходов и множеством позиций. Имеется несколько формальных представлений сетей Петри:

- теоретико-множественное представление;
- графово-бихроматический (двудольный ориентированный) граф и, соответственно, графическое представление;
- матричное представление.

При использовании теоретико-множественного подхода к описанию сети Петри (поскольку эта модель представляет и структуру, и функционирование системы) она формально может быть определена как двойка вида $N = <S, M_0>$. Здесь S — это структура сети, которая представляется двудольным ориентированным мультиграфом $S = (V, U)$, где V — вершины этого графа, U — его дуги. M_0 — это начальное состояние сети Петри, которое также называется начальной маркировкой. Сеть Петри может функционировать и соответственно изменять свое состояние.

В силу того что граф S является двудольным, можно перейти к формальному описанию структуры сети Петри в виде тройки:

$$S = <P, T, Y>.$$

Здесь P — конечное множество позиций, $P = \{p_i\}$, $i = \overline{1,n}$; T — конечное множество переходов, $T = \{t_j\}$, $j = \overline{1,m}$; $T \cup P = V$, $T \cap P = \varnothing$, то есть T и P — это два типа вершин биграфа S; Y — конечное множество дуг, заданное отношениями между вершинами графа S:

$$Y \in (P \cdot T) \cup (T \cdot P).$$

Поскольку двудольный мультиграф S является ориентированным, то любой переход t_j, $j = \overline{1,m}$, соединяется с позициями $p_i \in P$ через входные и выходные дуги, которые задаются через функцию предшествования $B : (P \cdot T) \to \{0, 1, 2, ...\}$ и функцию следования $E : (T \cdot P) \to \{0, 1, 2, ...\}$, являющиеся отображениями из множества переходов в комплекты позиций [36] (синонимом термина «комплект» является понятие мультимножества). Эти функции определяют комплекты позиций $\{p_i\} \in \overline{P}$, связанных с переходом $t_j \in T$ через множество дуг $\{(p_i, t_j)_l\}$, где $1 \le \left|\{(p_i, t_j)_l : i, j = \text{const}\}\right| \le W$, и комплекты позиций $\{p_k\} \in \overline{P}$, связанных с переходом $t_j \in T$ через множество дуг $\{(t_j, p_k)_l\}$, где $1 \le \left|\{(t_j, p_k)_l : j, k = \text{const}\}\right| \le W$. Здесь W — мультичисло графа S; $\overline{P}$ — пространство комплектов, заданное на множестве P функциями E и B; $(p_i, t_j)_v$ — v-я дуга, выходящая из позиции p_i и входящая в пере-

ход t_j; $(t_j, p_k)_v$ — v-я дуга, выходящая из перехода t_j и входящая в позицию p_k. Таким образом, теперь структура S сети Петри N может быть представлена как четверка:

$$S = <P, T, B, E>.$$

Представим далее множество позиций P как объединение двух пересекающихся множеств: $P = I \cup O$; $I \cap O \neq \varnothing$. Здесь мы через I и O обозначили следующие множества:

$$I = \bigcup_{j=1}^{m} I(t_j); \; O = \bigcup_{j=1}^{m} O(t_j).$$

Здесь

$$I(t_j) = \{p_i : B(p_i, t_j) \geq 1, i = \overline{1,n}\}, \; j = \overline{1,m}; \; O(t_j) = \{p_k : E(t_j, p_k) \geq 1, k = \overline{1,n}\}, \; j = \overline{1,m};$$

где (p_i, t_j) — дуга с весом $w \leq W$, выходящая из вершины p_i и входящая в вершину t_j; (t_j, p_k) — дуга с весом $w \leq W$, выходящая из вершины t_j и входящая в вершину p_k, то есть $I(t_j)$ и $O(t_j)$ — комплекты входных и выходных позиций перехода t_j соответственно .

Элементы множества T обычно представляют собой те возможности (возможные ситуации, условия), при которых могут быть реализованы интересующие нас процессы (действия).

Начальная маркировка M_0 (как и текущая маркировка M, которая соответствует некоторому состоянию сети в текущий момент модельного времени) определяется одномерной матрицей (вектором), число компонентов которой равно числу позиций сети n, $n = |P|$, а значение i-го компонента ($1 \leq i \leq n$) есть натуральное число $m(p_i)$, которое определяет количество маркеров (меток) в позиции p_i:

$$M_0 = (m_0(p_1), m_0(p_2), \ldots, m_0(p_n));$$

$$M = (m(p_1), m(p_2), \ldots, m(p_n)).$$

Здесь $m_0(p_i)$, $m(p_i) \in Z$; Z — множество неотрицательных целых чисел. Ее же (маркировку M) можно также представлять как множество или комплект с той разницей, что отсутствие некоторого элемента в множестве будем обозначать специальным элементом — нулем. В этом случае запись вида $M_i = M_{i-1} - I(t)$ означает разность множеств и такое изменение маркировки, при котором на соответствующих местах вектора M_i будут уменьшенные значения.

Передвижение маркеров по сети осуществляется посредством срабатывания ее переходов. При срабатывании перехода изменяется маркировка в его входных и выходных позициях. Получается, что срабатывание возбужденного перехода, являющееся локальным актом, в целом ведет к изменению маркировки сети, то есть к изменению ее состояния. Таким образом, если в сети задана начальная маркировка M_0, при которой хотя бы один переход возбужден, то в сети начинается движение маркеров, отображающее смену состояний сети. Переход t_j может сработать, если

$$p_i \in I(t_j) : m(p_i) \geq \#(p_i, I(t_j)) - w.$$

Переход, для которого выполняется это условие, называется *возбужденным*. Здесь запись вида $\#(p_i, I(t_j))$ означает число появлений позиций p_i во входном комплекте перехода t_j; оно, естественно, равно весу w соответствующей дуги, если вместо мультиграфа рассматривать взвешенный граф. При срабатывании перехода t_j маркировка M_0 изменяется на маркировку M_1 следующим образом:

$$M_1 = M_0 - I(t_j) + O(t_j).$$

Иначе говоря:

$$\forall p_i \in P : m_1(p_i) = m_0(p_i) - \#(p_i, I(t_j)) + \#(p_i, O(t_j)).$$

Из последнего выражения видно, что количество маркеров, которое переход t_j изымает из своих входных позиций, может не равняться количеству маркеров, которое этот переход помещает в свои выходные позиции, так как совсем не факт, что число входных дуг перехода равняется числу его выходных дуг.

В графическом представлении сетей (оно наиболее наглядно и легко интерпретируемо) переходы изображаются вертикальными или горизонтальными планками (черточками), а позиции — кружками (см. далее). Условия-позиции и события-переходы связаны отношением непосредственной зависимости (непосредственной причинно-следственной связи), которое изображается с помощью направленных дуг, ведущих из позиций в переходы и из переходов в позиции. Позиции, из которых ведут дуги на данный переход, называются его входными позициями, а позиции, на которые ведут дуги из данного перехода, — выходными позициями.

Выполнение условия представляется разметкой соответствующей позиции, а именно помещением числа n в это место или изображением там n маркеров (фишек), где n — емкость условия ($n > 0$).

Говорят, что некоторый переход t_j для разметки M является *живым*, если для всех разметок M', достижимых из разметки M, существует последовательность срабатывания переходов, приводящая к маркировке M', при которой переход t_j может сработать. Сеть Петри называется *живой*, если живы все ее переходы; живучая разметка — это разметка, при которой каждый из ее переходов сможет запускаться бесконечное число раз. Когда достигнута такая разметка, при которой ни один из переходов не может быть запущен, говорят, что сеть Петри завершилась (достигнута желаемая конечная маркировка) или же зависла (то есть имеет место тупиковая ситуация).

Сети Петри очень удобны для описания процессов синхронизации и альтернатив. Например, семафор может быть представлен входной позицией, связанной с несколькими взаимоисключающими переходами (критическими секциями). Сети Петри позволяют моделировать асинхронность и недетерминизм параллельных независимых событий, параллелизм конвейерного типа, конфликтные взаимодействия между процессами. Говорят, что два перехода конфликтуют, если они взаимно исключают друг друга, то есть они не могут быть запущены оба одновременно.

Два перехода, готовые к срабатыванию, находятся в конфликте, если они связаны с общей входной позицией.

В качестве примера рассмотрим рис. 8.5.

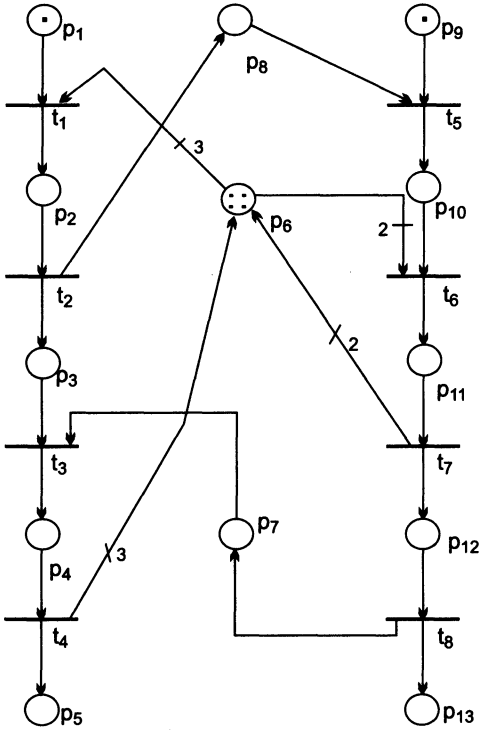


Рис. 8.5. Сеть Петри для системы двух взаимодействующих процессов

Эта сеть соответствует примеру тупиковой ситуации, которая возникает при взаимодействии процессов Пр1 и Пр2 во время передачи сообщений и потреблении ресурса R каждым из процессов (см. листинг 8.3). Начальная маркировка для сети, показанной на рис. 8.5, будет равна (1, 0, 0, 0, 0, 4, 0, 0, 1, 0, 0, 0, 0). Здесь позиция p_2 означает, что процесс Пр1 получил три единицы ресурса R. Дуга, соединяющая позицию p_6 (число маркеров в ней соответствует количеству доступных единиц ресурса R), имеет вес 3, и при срабатывании перехода t_1 процесс Пр1 получает затребованные три единицы ресурса. Переход t_2 представляет посылку процессом Пр1 сообщения для Пр2; переход t_5 — прием этого сообщения. Появление маркера в позиции p_7 означает, что процесс Пр2 обработал сообщение и послал ответ процессу Пр1. Срабатывание перехода t_4 представляет возврат в систему трех единиц ресурса, которыми владел процесс Пр1. Рассмотренная сеть не является живой, так как в ней всегда будут мертвы переходы t_3, t_4, t_6, t_7 и t_8.

Примеру тупиковой ситуации, возникающей при работе с ресурсами типа SR (см. рис. 8.3), соответствует сеть Петри, показанная на рис. 8.6.

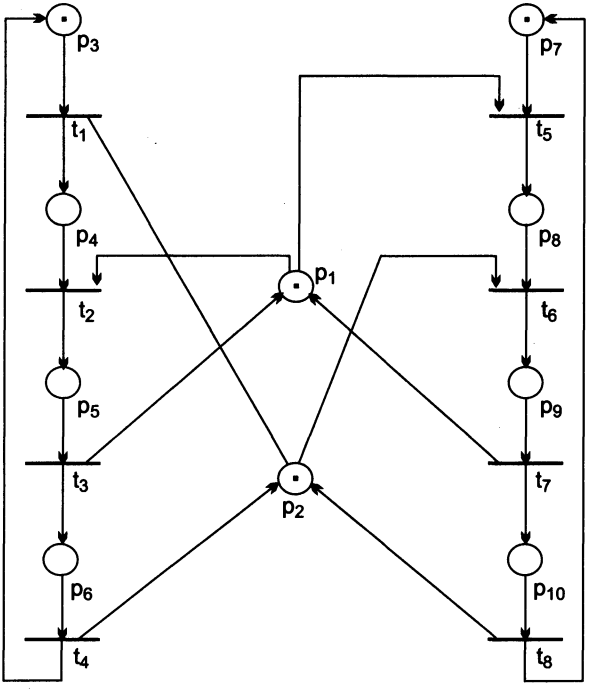


Рис. 8.6. Сеть Петри для тупиковой ситуации на ресурсах типа SR

В этой сети номера переходов соответствуют отмеченным номерам операторов, которые выполняют процессы Пр1 и Пр2, а позиции p_1 и p_2 — семафорам S1 и S2, над которыми выполняются операции P и V. Сеть на рис. 8.6 также не является живой, хотя для нее и существуют последовательности срабатывания переходов, не ведущие к тупиковой ситуации.

Сети Петри могут быть использованы для анализа системы на возможность возникновения в ней тупиковых ситуаций. При таком анализе исследуется пространство возможных состояний сети Петри, под которым понимается множество возможных маркировок сети. Для анализа сетей посредством матричных методов характерно множество проблем, поэтому в основном используется подход, основанный на построении редуцированного до дерева[1] графа возможных маркировок [21]. В таком дереве вершины графа — это состояния (маркировки) сети, а ветви дерева, помеченные соответствующими переходами сети, — это возможные изменения состояний сети, то есть срабатывания ее переходов.

Модель пространства состояний системы

Приведем еще одну формальную модель (она подробно рассмотрена в работе [54]). Эта модель очень проста, однако она позволяет сформулировать, что нам нужно делать, чтобы не попасть в тупиковое состояние.

[1] Напомним, что деревом в теории графов называют граф, не имеющий циклов.

Пусть состояние операционной системы сводится к состоянию различных ресурсов в системе (свободны они или выделены какому-нибудь процессу). Состояние системы изменяется процессами, когда они запрашивают, приобретают или освобождают ресурсы — это единственно возможные действия (точнее, мы принимаем во внимание только такие действия). Если процесс не блокирован в данном состоянии, он может изменить это состояние на новое. Однако так как в общем случае невозможно знать априори, какой путь может избрать произвольный процесс в своей программе (это неразрешимая проблема!), то новое состояние может быть любым из конечного числа возможных. Следовательно, процессы нами будут трактоваться как недетерминированные объекты. Введенные ограничения на известные понятия приводят нас к нескольким формальным определениям.

- Система есть пара $<\sigma, \pi>$, где

  σ — множество состояний системы $\{ S_1, S_2, S_3, ..., S_n \}$;

  π — множество процессов $\{ P_1, P_2, P_3, ..., P_k \}$.

- Процесс P_i есть частичная функция, отображающая состояние системы в непустые подмножества ее же состояний. Это обозначается так:

$$P_i: \sigma \rightarrow \{ \sigma \}.$$

Если процесс P_i определен на состоянии S, то область значений P_i обозначается как $P_i(S)$. Если $S_k \in P_i(S_e)$, то мы говорим, что P_i может изменить состояние S_e в состояние S_k посредством операции, и используем обозначение $S_e \xrightarrow{P_i} S_k$.

Наконец, запись $S_e \xrightarrow{*} S_w$ означает, что $S_e = S_w$, или $S_e \xrightarrow{P_i} S_w$ для некоторого i, или $S_e \xrightarrow{P_i} S_k$ для некоторых i и S_k, причем $S_k \xrightarrow{*} S_w$.

Другими словами, система может быть переведена посредством $n \geq 0$ операций с помощью $m \geq 0$ различных процессов из состояния S_e в состояние S_w.

Мы говорим, что процесс заблокирован в данном состоянии, если он не может изменить состояние, то есть в этом состоянии процесс не может ни требовать, ни получать, ни освобождать ресурсы. Поэтому справедливо следующее.

Процесс P_i заблокирован в состоянии S_e, если не существует ни одного состояния S_k, такого что $S_e \xrightarrow{P_i} S_k$, причем $P_i(S_e) = \varnothing$.

Далее, мы говорим, что процесс P_i находится в тупике в данном состоянии S_e, если он заблокирован в состоянии S_e и если вне зависимости от того, какие операции (изменения состояний) произойдут в будущем, процесс P_i остается заблокированным. Поэтому дадим еще одно определение.

Процесс P_i находится в тупике в состоянии S_e, если для всех состояний S_k, таких что $S_e \xrightarrow{*} S_k$, процесс P_i блокирован в состоянии S_k.

Приведем пример. Определим систему $<\sigma, \pi>$:

$$\sigma = \{ S_1, S_2, S_3, S_4 \}; \qquad \pi = \{ P_1, P_2 \};$$
$$P_1(S_1) = \{ S_2, S_3 \}; \qquad P_2(S_1) = \{ S_3 \};$$
$$P_1(S_2) = \varnothing; \qquad P_2(S_2) = \{ S_1, S_4 \};$$
$$P_1(S_3) = \{ S_4 \}; \qquad P_2(S_3) = \varnothing;$$
$$P_1(S_4) = \{ S_3 \}; \qquad P_2(S_4) = \varnothing.$$

Некоторые возможные последовательности изменений для этой системы таковы:

$$S_1 \xrightarrow{P_1} S_3; S_2 \xrightarrow{P_2} S_4; S_1 \xrightarrow{*} S_4.$$

Последовательность $S_1 \xrightarrow{*} S_4$ может быть реализована, например, следующим образом: $S_1 \xrightarrow{P_1} S_2; S_2 \xrightarrow{P_2} S_4$ или же $S_1 \xrightarrow{P_1} S_3; S_3 \xrightarrow{P_1} S_4$.

Заметим, что процесс P_2 находится в тупике как в состоянии S_3, так и в состоянии S_4; для последнего случая $S_2 \xrightarrow{P_2} S_4$ или $S_2 \xrightarrow{P_2} S_1$, и процесс P_1 не оказывается заблокированным ни в S_4, ни в S_1.

Диаграмма переходов этой системы изображена на рис. 8.7.

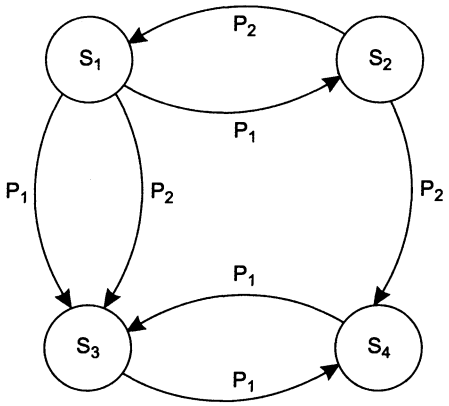


Рис. 8.7. Пример системы <σ, π> на четыре состояния

Состояние S называется тупиковым, если существует процесс P_i, находящийся в тупике в состоянии S.

Теперь мы можем сказать, что, по определению, тупик предотвращается при введении такого ограничения на систему, чтобы каждое ее возможное состояние не было тупиковым состоянием.

Введем еще одно определение.

Состояние S_i есть безопасное состояние, если для всех S_k, таких что $S_i \xrightarrow{*} S_k$, S_k не является тупиковым состоянием.

Рассмотрим еще один пример системы <σ, π>. Граф ее состояний приведен на рис. 8.8. Здесь состояния S_2 и S_3 являются безопасными; из них система никогда не сможет попасть в тупиковое состояние. Состояния S_1 и S_4 могут привести как к безопасным состояниям, так и к опасному состоянию S_5. Состояния S_6 и S_7 являются тупиковыми.

Наконец, в качестве еще одной простейшей системы вида <σ, π> приведем пример тупика с ресурсами типа SR, уже рассмотренный нами ранее и проиллюстрированный рис. 8.3. Для этого определим состояния процессов P_1 и P_2, которые используют ресурсы R_1 и R_2 (табл. 8.1).

Пусть состояние системы S_{ij} означает, что процесс P_1 находится в состоянии S_j, а процесс P_2 — в состоянии S_j. Возможные изменения в пространстве состояний

этой системы изображены на рис. 8.9. Вертикальными стрелками показаны возможные переходы из одного состояния в другое для процесса P_1, а горизонтальными — для процесса P_2. В данной системе имеется три опасных состояния: S_{22}, S_{23} и S_{32}. Попав в любое из них, мы неминуемо перейдем в тупиковое состояние S_{33}.

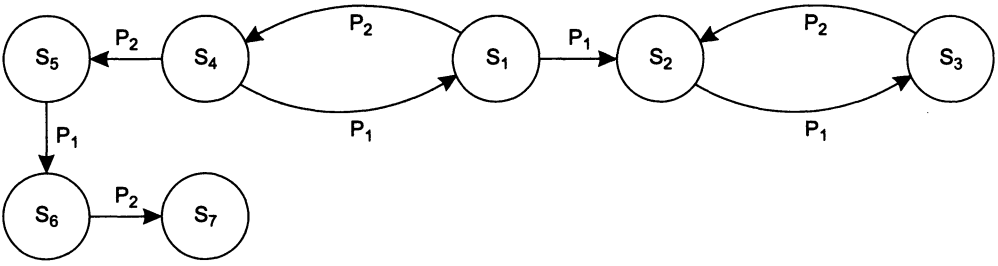


Рис. 8.8. Пример системы <σ, π> с безопасными, опасными и тупиковым состояниями

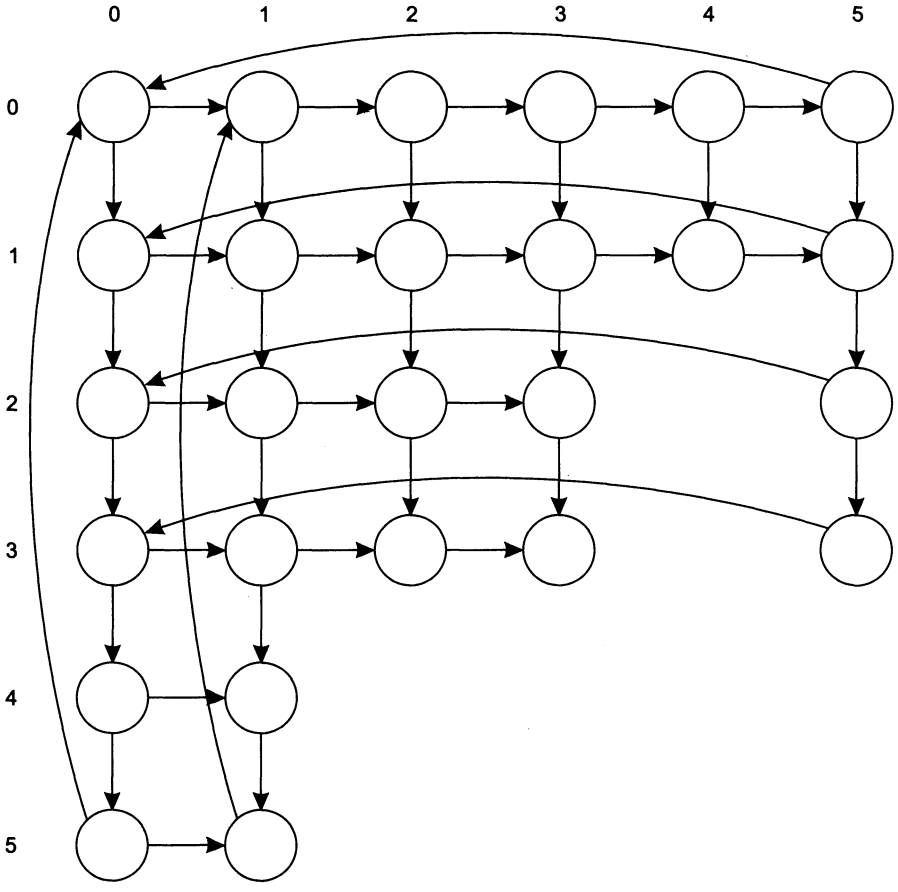


Рис. 8.9. Модель системы из двух процессов

Таблица 8.1. Состояния процессов P_1 и P_2 при использовании ресурсов R_1 и R_2

P_1	Описание	P_2	Описание
0	Не держит никаких ресурсов	0	Не держит никаких ресурсов
1	Запросил ресурс R_2, не держит никаких ресурсов	1	Запросил ресурс R_1, не держит никаких ресурсов
2	Держит ресурс R_2	2	Держит ресурс R_1
3	Запросил ресурс R_1, держит ресурс R_2	3	Запросил ресурс R_2, держит ресурс R_1
4	Держит ресурсы R_1 и R_2	4	Держит ресурсы R_1 и R_2
5	Держит ресурс R_2 после освобождения ресурса R_1	5	Держит ресурс R_2 после освобождения ресурса R_1

Теперь, когда мы знаем понятия надежного, опасного и безопасного состояний, можно рассмотреть методы борьбы с тупиками.

Методы борьбы с тупиками

Проблема тупиков является чрезвычайно серьезной и сложной. Разработано несколько подходов к разрешению этой проблемы, однако ни один из них нельзя считать панацеей. В некоторых случаях цена, которую приходится платить за то, чтобы освободить систему от тупиков, слишком высока. Кстати, именно по этой причине нам не так уж редко приходится сталкиваться с тупиковыми ситуациями. В других случаях, например в системах управления процессами реального времени, просто нет иного выбора, как идти на значительные затраты, поскольку возникновение тупика может привести к катастрофическим последствиям.

Проблема борьбы с тупиками становится все более актуальной и сложной по мере развития и внедрения параллельных вычислительных систем. При проектировании таких систем разработчики стараются проанализировать возможные тупиковые ситуации, используя специальные модели и методы. Борьба с тупиковыми ситуациями основывается на одной из трех стратегий:

- предотвращение тупика;
- обход тупика;
- распознавание тупика с последующим восстановлением.

Предотвращение тупиков

Предотвращение тупика основывается на предположении о чрезвычайно высокой его стоимости, поэтому лучше потратить дополнительные ресурсы системы, чтобы исключить вероятность его возникновения при любых обстоятельствах. Этот подход используется в наиболее ответственных системах, обычно в системах реального времени.

Предотвращение можно рассматривать как запрет существования опасных состояний. Поэтому подсистема распределения ресурсов, предотвращающая тупик, должна гарантировать, что ни одного из четырех условий, необходимых для его наступления, не возникнет.

- *Условие взаимного исключения* можно подавить путем разрешения неограниченного разделения ресурсов. Это удобно для повторно входимых программ и ряда драйверов, но совершенно неприемлемо для совместно используемых переменных в критических секциях.
- *Условие ожидания* можно подавить, предварительно выделяя ресурсы. При этом процесс может начать исполнение, только получив все необходимые ресурсы заранее. Следовательно, общее число затребованных параллельными процессами ресурсов должно быть не больше возможностей системы. Поэтому предварительное выделение может привести к снижению эффективности работы вычислительной системы в целом. Необходимо также отметить, что предварительное выделение ресурсов зачастую невозможно, так как реально необходимые ресурсы становятся известны процессу только в ходе исполнения.
- *Условие отсутствия перераспределения* можно исключить, позволяя операционной системе отнимать у процесса ресурсы. Для этого в операционной системе должен быть предусмотрен механизм запоминания состояния процесса с целью последующего восстановления хода вычислений. Перераспределение процессора реализуется достаточно легко, в то время как перераспределение устройств ввода-вывода крайне нежелательно.
- *Условие кругового ожидания* можно исключить, предотвращая образование цепи запросов. Это можно обеспечить с помощью принципа *иерархического выделения ресурсов*. Все ресурсы образуют некоторую иерархию. Процесс, затребовавший ресурс на одном уровне, может затем потребовать ресурсы только на более высоком уровне. Он может освободить ресурсы на данном уровне, только после освобождения всех ресурсов на всех более высоких уровнях. После того как процесс получил, а потом освободил ресурсы данного уровня, он может запросить ресурсы на том же самом уровне. Пусть имеются процессы Пр1 и Пр2, которые могут иметь доступ к ресурсам R1 и R2, причем R2 находится на более высоком уровне иерархии. Если процесс Пр1 захватил ресурс R1, то процесс Пр2 не может захватить ресурс R2, так как доступ к нему проходит через доступ к ресурсу R1, который уже захвачен процессом Пр1. Таким образом, создание замкнутой цепи исключается. Иерархическое выделение ресурсов часто не дает никакого выигрыша, если порядок использования ресурсов, определенный в описании процессов, отличается от порядка уровней иерархии. В этом случае ресурсы будут расходоваться крайне неэффективно.

В целом стратегия предотвращения тупиков — это очень дорогое решение проблемы тупиков, и эта стратегия используется нечасто.

Обход тупиков

Обход тупика можно интерпретировать как запрет входа в опасное состояние. Если ни одно из упомянутых четырех условий не исключено, то вход в опасное состояние можно предотвратить при наличии у системы информации о последовательности запросов, связанных с каждым параллельным процессом. Доказано [54], что если вычисления находятся в любом неопасном состоянии, то существует, по крайней мере, одна последовательность состояний, которая обходит опасное состоя-

ние. Следовательно, достаточно проверить, не приведет ли выделение затребованного ресурса сразу же к опасному состоянию. Если да, то запрос отклоняется, если нет, его можно выполнить. Определение того, является состояние опасным или нет, требует анализа последующих запросов от процессов.

Рассмотрим следующий пример. Пусть имеется система из трех вычислительных процессов, потребляющих некоторый ресурс R типа SR; который выделяется дискретными взаимозаменяемыми единицами, причем существует всего десять единиц этого ресурса. В табл. 8.2 приведены сведения о текущем распределении процессами этого ресурса R, об их текущих запросах на этот ресурс и о максимальных потребностях процессов в ресурсе R.

Таблица 8.2. Пример распределения ресурсов

Имя процесса	Выделено	Запрос	Максимальная потребность	Остаток потребностей
A	2	3	6	1
B	3	2	7	2
C	2	3	5	0

Последний столбец в таблице показывает нам, сколько еще единиц ресурса может затребовать каждый из процессов, если бы он получил ресурс на свой текущий запрос.

Если запрос процесса A будет удовлетворен первым, то он в принципе может запросить еще одну единицу ресурса R, и уже в этом случае мы получим тупиковую ситуацию, поскольку ни один из наших процессов не сможет продолжить свои вычисления. Следовательно, при выполнении запроса процесса A мы попадаем в ненадежное[1] состояние.

Если первым будет выполнен запрос процесса B, то у нас останется свободной еще одна единица ресурса R. Однако если процесс B запросит еще две, а не одну единицу ресурса R, а он может это сделать, то мы опять получим тупиковую ситуацию.

Если же мы сначала выполним запрос процесса C и выделим ему не две (как процессу B), а все три единицы ресурса R, то у нас не останется никакого резерва, но процесс C сможет благополучно завершиться и вернуть системе все свои ресурсы, поскольку на этом его потребности в ресурсах заканчиваются. Это приведет к тому, что свободное количество ресурса R станет равно пяти. После этого уже можно будет удовлетворить запрос либо процесса B, либо процесса A, но не обоих сразу.

Часто бывает так, что последовательность запросов, связанных с каждым процессом, заранее не известна. Но если заранее известен общий запрос на ресурсы каждого типа, то выделение ресурсов можно контролировать. В этом случае необходимо для каждого требования, в предположении, что оно удовлетворено, определить, существует ли среди общих запросов от всех процессов некоторая последователь-

[1] Термин «ненадежное состояние» не предполагает, что в данный момент существует или в какое-то время обязательно возникнет тупиковая ситуация. Он просто говорит о том, что в случае некоторой неблагоприятной последовательности событий система может зайти в тупик.

ность требований, которая может привести к опасному состоянию. Данный подход является примером контролируемого выделения ресурса.

Классическое решение этой задачи предложено Дейкстрой и известно как *алгоритм банкира* [53]. Алгоритм банкира напоминает процедуру принятия решения о том, может ли банк безопасно для себя дать взаймы денег. Принятие решения основывается на информации о потребностях клиента (нынешних и максимально возможных в принципе) и учете текущего баланса банка. Хотя этот алгоритм практически нигде не используется, рассмотрим его, так как он интересен с методической и академической точек зрения.

Пусть существует N процессов, для каждого из которых известно максимальное количество потребностей в некотором ресурсе R (обозначим эти потребности через Max(i)). Ресурсы выделяются не сразу все, а в соответствии с текущим запросом. Считается, что все ресурсы i-го процесса будут освобождены по его завершении. Количество полученных ресурсов для i-го процесса обозначим Получ(i). Остаток в потребностях i-го процесса на ресурс R обозначим через Остаток(i). Признак того, что процесс может не завершиться, — это значение false для переменной Заверш(i). Наконец, переменная Своб_рес будет означать количество свободных единиц ресурса R, а максимальное количество ресурсов в системе определено значением Всего_рес. Текст программы алгоритма банкира приведен в листинге 8.4.

Листинг 8.4. Алгоритм банкира Дейкстры

```
Begin
  Своб_рес := Всего_рес;
  For i := 1 to N do
    Begin
      Своб_рес := Своб_рес - Получ(i);
      Остаток(i) := Max(i) - Получ(i);
      Заверш(i) := false;  { процесс может не завершиться }
    End;
  flag := true;     { признак продолжения анализа }
  while flag do
    begin
      flag := false;
      for i := 1 to N do
        begin
          if ( not ( Заверш(i) )) and ( Остаток(i) <= Своб_рес )
            then begin
                Заверш(i) := true;
                Своб_рес := Своб_рес + Получ(i);
                Flag := true
                 End
        End
    End;
  If Своб_рес = Всего_рес
      then     Состояние системы безопасное,
                 и можно выдать ресурс
      else     Состояние небезопасное,
                 и выдавать ресурс нельзя
end.
```

Каждый раз, когда что-то может быть выделено из числа остающихся незанятыми ресурсов, предполагается, что соответствующий процесс работает, пока не окон-

чится, а затем его ресурсы освобождаются. Если в конце концов все ресурсы освобождаются, значит, все процессы могут завершиться и система находится в безопасном состоянии. Другими словами, согласно алгоритму банкира система удовлетворяет только те запросы, при которых ее состояние остается надежным. Новое состояние безопасно тогда и только тогда, когда каждый процесс может окончиться. Именно это условие и проверяется в алгоритме банкира. Запросы процессов, приводящие к переходу системы в ненадежное состояние, не выполняются и откладываются до момента, когда их можно будет выполнить.

Алгоритм банкира позволяет продолжать выполнение таких процессов, которым в случае системы с предотвращением тупиков пришлось бы ждать. Хотя алгоритм банкира относительно прост, его реализация может обойтись довольно дорого. Основным накладным расходом стратегии обхода тупика с помощью контролируемого выделения ресурса является время выполнения алгоритма, так как он выполняется при каждом запросе. Причем, алгоритм работает наиболее медленно, когда система близка к тупику. Необходимо отметить, что обход тупика неприменим при отсутствии информации о требованиях процессов на ресурсы.

Рассмотренный алгоритм примитивен, в нем учитывается только один вид ресурса, тогда как в реальных системах количество различных типов ресурсов бывает очень большим. Были опубликованы варианты этого алгоритма для большого числа различных типов системных ресурсов. Однако все равно алгоритм не получил распространения. Причин тому несколько.

- Алгоритм исходит из того, что количество распределяемых ресурсов в системе фиксировано. Иногда это не так, например вследствие неисправности отдельных устройств.
- Алгоритм требует, чтобы пользователи заранее указывали свои максимальные потребности в ресурсах. Это чрезвычайно трудно реализовать. Часть таких сведений, конечно, могла бы предоставлять система программирования, но все равно оставшуюся часть информации о потребностях в ресурсах должны давать пользователи. Однако чем более дружественными по отношению к пользователям становятся компьютеры, тем чаще встречаются пользователи, которые не имеют ни малейшего представления о том, какие ресурсы им потребуются.
- Алгоритм требует, чтобы число работающих процессов оставалось постоянным. Очевидно, что это требование также, в общем, нереалистично, особенно в мультитерминальных системах и в условиях, когда пользователь запускает несколько параллельных процессов.

Обнаружение тупика

Чтобы распознать тупиковое состояние, необходимо для каждого процесса определить, сможет ли он когда-либо снова развиваться, то есть изменять свои состояния. Так как нас интересует возможность развития процесса, а не сам процесс смены состояния, то достаточно рассмотреть только самые благоприятные изменения состояния.

Очевидно, что незаблокированный процесс (имеющий все необходимые ресурсы или только что получивший ресурс и поэтому пока незаблокированный) через некоторое время освобождает все свои ресурсы и затем благополучно завершается. Освобождение ранее занятых ресурсов может «разбудить» некоторые ранее заблокированные процессы, которые, в свою очередь, развиваясь, смогут освободить другие ранее занятые ресурсы. Это может продолжаться до тех пор, пока либо не останется незаблокированных процессов, либо какое-то количество процессов все же останется заблокированными. В последнем случае (если существуют заблокированные процессы при завершении указанной последовательности действий) начальное состояние S является состоянием тупика, а оставшиеся процессы находятся в тупике в состоянии S. В противном случае S не есть состояние тупика.

Обнаружение тупика посредством редукции графа повторно используемых ресурсов

Наиболее благоприятные условия для незаблокированного процесса P_i могут быть представлены *редукцией* (сокращением) графа повторно используемых ресурсов (см. описание модели Холта ранее в разделе «Понятие тупиковой ситуации при выполнении параллельных вычислительных процессов»). Редукция графа может быть описана следующим образом.

- Граф повторно используемых ресурсов сокращается процессом P_i, который не является ни заблокированной, ни изолированной вершиной, путем удаления всех ребер, входящих в вершину P_i и выходящих из P_i. Эта процедура является эквивалентной приобретению процессом P_i неких ресурсов, на которые он ранее выдавал запросы, а затем освобождению всех его ресурсов. Тогда P_i становится изолированной вершиной.
- Граф повторно используемых ресурсов несокращаем (не редуцируется), если он не может быть сокращен ни одним процессом.
- Граф ресурсов типа SR является полностью сокращаемым, если существует последовательность сокращений, которая удаляет все дуги графа.

Лемма: для ресурсов типа SR порядок сокращений дуг графа повторно используемых ресурсов несущественен; все последовательности ведут к одному и тому же несокращаемому графу.

Допустим, что лемма неверна. Тогда должно существовать некоторое состояние S, которое сокращается до некоторого несокращаемого состояния S_1 с помощью последовательности seq_1 и до несокращаемого состояния S_2 с помощью последовательности seq_2 так, что $S_1 \neq S_2$ (то есть все процессы в состояниях S_1 и S_2 или заблокированы, или изолированы).

Если сделать такое предположение, то мы приходим к противоречию, которое устраняется только при условии, что $S_1 = S_2$. Действительно, предположим, что последовательность seq_1 состоит из упорядоченного списка процессов (P_1, P_2, ..., P_k). Тогда последовательность seq_1 должна содержать процесс P, который не содержится в последовательности seq_2. В противном случае $S_1 = S_2$, потому что редукция графа только удаляет дуги, уже существующие в состоянии S, а если последовательности seq_1 и seq_2 содержат одно и то же множество процессов (пусть и в раз-

личном порядке), то должно быть удалено одно и то же множество дуг. И доказательство по индукции покажет, что $P \neq P_i$, $(i = 1, 2, ..., k)$ приводит к нашему противоречию.

- Имеет место соотношение $P \neq P_1$, так как вершина S может быть редуцирована процессом P_1, а состояние S_2 должно, следовательно, также содержать процесс P_1.
- Пусть $P \neq P_i$, $(i = 1, 2, ..., j)$. Однако поскольку после редукции процессами P_i, $(i = 1, 2, ..., j)$ возможно еще сокращение графа процессом P_{j+1}, это же самое должно быть справедливо и для последовательности seq_2 независимо от порядка следования процессов. То же самое множество ребер графа удаляется с помощью процесса P_i. Таким образом, $P \neq P_{j+1}$.

Следовательно, $P \neq P_i$ для $i = 1, 2, ..., k$ и P не может существовать, а это противоречит нашему предположению, что $S_1 \neq S_2$. Следовательно, $S_1 = S_2$.

Теорема о тупике: Состояние S есть состояние тупика тогда и только тогда, когда граф повторно используемых ресурсов в состоянии S не является полностью сокращаемым.

- Для доказательства предположим, что состояние S есть состояние тупика, и процесс P_i находится в тупике в S. Тогда для всех S_j, таких что $S \xrightarrow{*} S_j$ процесс P_i заблокирован в состоянии S_j (по определению). Так как сокращения графа идентичны для серии операций процессов, то конечное несокращаемое состояние в последовательности сокращений должно оставить процесс P_i блокированным. Следовательно, граф не является полностью сокращаемым.
- Предположим теперь, что состояние S не является полностью сокращаемым. Тогда существует процесс P_i, который остается заблокированным при всех возможных последовательностях операций редукции в соответствии с леммой (см. выше). Так как любая последовательность операций редукции графа повторно используемых ресурсов, оканчивающаяся несокращаемым состоянием, гарантирует, что все ресурсы типа SR, которые могут когда-либо стать доступными, в действительности освобождены, то процесс P_i навсегда заблокирован и, следовательно, находится в тупике.

Первое следствие: процесс P_i не находится в тупике тогда и только тогда, когда серия сокращений приводит к состоянию, в котором P_i не заблокирован.

Второе следствие: если S есть состояние тупика (по ресурсам типа SR), то по крайней мере два процесса находятся в тупике в S.

Из теоремы о тупике непосредственно следует и алгоритм обнаружения тупиков. Нужно просто попытаться сократить граф по возможности эффективным способом; если граф полностью не сокращается, то начальное состояние было состоянием тупика для тех процессов, вершины которых остались в несокращенном графе. Рассмотренная нами лемма позволяет предложить алгоритмы обнаружения тупика. Например, можно представить систему посредством графа повторно используемых ресурсов и попробовать выполнить его редукцию, причем делать это следует, стараясь упорядочивать сокращения удобным образом.

Граф повторно используемых ресурсов может быть представлен матрицами или списками. В обоих случаях экономия памяти может быть достигнута за счет взве-

шенных ориентированных мультиграфов (слиянием определенных дуг получения или дуг запроса между конкретным ресурсом и данным процессом в одну дугу с соответствующим весом, определяющим количество единиц ресурса).

Рассмотрим вариант с матричным представлением. Поскольку граф двудольный, он представляется двумя матрицами размером n × m. Одна матрица — *матрица распределения* $D = \|d_{ij}\|$, в которой элемент d_{ij} отражает количество единиц R_j ресурса, выделенного процессу P_i, то есть $d_{ij} = |(R_j, P_i)|$, где (R_j, P_i) — это дуга между вершинами R_j и P_i, ведущая из R_j в P_i. Вторая матрица — *матрица запросов* $N = \|n_{ij}\|$, где $n_{ij} = |(P_i, R_j)|$.

В случае использования связанных списков для отображения той же структуры можно построить две группы списков. Ресурсы, выделенные некоторому процессу P_i, связаны с P_i указателями:

$$P_i \longrightarrow (R_x, d_x) \longrightarrow (R_y, d_y) \longrightarrow \ldots \longrightarrow (R_z, d_z).$$

Здесь R_j — вершина, представляющая ресурс, d_j — вес дуги, то есть $d_j = |(R_j, P_i)|$.

Подобным образом и ресурсы, запрошенные процессом P_i, связаны вместе.

Аналогичные списки создаются и для ресурсов (списки распределенных и запрошенных ресурсов):

$$R_i \longrightarrow (P_u, n_u) \longrightarrow (P_v, n_v) \longrightarrow \ldots \longrightarrow (P_w, n_w).$$

Здесь $n_j = |(P_j, R_i)|$.

Для обоих представлений удобно также иметь одномерный массив доступных единиц ресурсов $(r_1, r_2, \ldots, r_m)$, где r_i обозначает число доступных (нераспределенных единиц) ресурса R_i, то есть $r_i = |R_i| - \Sigma|(R_i, P_k)|$.

Простой метод прямого обнаружения тупика заключается в просмотре по порядку списка (или матрицы) запросов, причем, где возможно, производятся сокращения дуг графа до тех пор, пока нельзя будет сделать более ни одного сокращения. При этом самая плохая ситуация возникает, когда процессы упорядочены в некоторой последовательности $P_1, P_2, \ldots, P_n$, а единственно возможным порядком сокращений является обратная последовательность, то есть $P_n, P_{n-1}, \ldots, P_2, P_1$, а также в случае, когда процесс запрашивает все m ресурсов. Тогда число проверок процессов равно:

$$n + (n - 1) + \ldots + 1 = n \cdot (n + 1)/2.$$

Таким образом, время выполнения такого алгоритма в наихудшем случае пропорционально $m \times n^2$, поскольку каждая проверка требует испытания m ресурсов.

Более эффективный алгоритм может быть получен за счет хранения некоторой дополнительной информации о запросах. Для каждой вершины процесса P_i определяется так называемый *счетчик ожиданий* w_i, отображающий количество ресурсов (не число единиц ресурса), которые в какое-то время вызывают блокировку процесса. Кроме того, можно сохранять для каждого ресурса запросы, упорядоченные по размеру (числу единиц ресурса). Тогда следующий алгоритм сокращений, записанный на псевдокоде, имеет максимальное время выполнения, пропорциональное m × n.

```
For all P ∈ L do
Begin  for all Rj э |(Rj,P)| > 0 do
  Begin  rj := rj + |(Rj,P)|;
      For all Pi э 0 < |(Pi,Rj)| <= rj do
        Begin  wi := wi - 1;
            If wi = 0 then L := L U {Pi}
        End
  End
End
Deadlock := Not (L = {P1, P2, …, Pn});
```

Здесь L — это текущий список процессов, которые могут выполнять редукцию графа. Можно сказать, что $L = \{P_i \mid w_i = 0\}$. Программа выбирает процесс P из списка L, процесс P сокращает граф, увеличивая число доступных единиц r_j всех ресурсов R_j, распределенных процессу P, обновляет счетчик ожидания w_i каждого процесса P_i, который сможет удовлетворить свой запрос на частный ресурс R_j, и добавляет P_i к L, если счетчик ожидания становится нулевым.

Методы обнаружения тупика по наличию замкнутой цепочки запросов

Структура графа обеспечивает простое необходимое (но не достаточное) условие для тупика. Для любого графа G = <X, E> и вершины x I X пусть P(x) обозначает множество вершин, достижимых из вершины x, то есть

$$P(x) = \{ y \mid (x, y) \text{ I } E \} \text{ U } \{ z \mid (y, z) \text{ I } E \text{ \& } y \text{ I } P(x) \}.$$

Можно сказать, что в ориентированном графе *потомством вершины x*, обозначенным как P(x), называется множество всех вершин, в которые ведут пути из x.

Тогда если существует некоторая вершина x I X: x I P(x), то в графе G имеется цикл.

Первая теорема: цикл в графе повторно используемых ресурсов является необходимым условием тупика.

Для доказательства этой теоремы (которое мы опустим[1]) можно воспользоваться следующим свойством ориентированных графов: если ориентированный граф не содержит цикла, то существует линейное упорядочение вершин такое, что если существует путь от вершины i к вершине j, то i появляется перед j в этом упорядочении.

Вторая теорема: если S не является состоянием тупика и $S \xrightarrow{P_i} S_T$, где S_T есть состояние тупика, в том и только в том случае, когда операция процесса P_i есть запрос и P_i находится в тупике в S_T.

Это следует понимать таким образом, что тупик может быть вызван только при запросе, который не удовлетворен немедленно. Учитывая эту теорему, можно сделать вывод, что проверка на тупиковое состояние может быть выполнена более эффективно, если она проводится непрерывно, то есть по мере развития процессов. Тогда надо применять редукцию графа только после запроса от некоторого

[1] При желании его можно найти в [54].

процесса P_i и на любой стадии необходимо сначала попытаться сократить граф с помощью процесса P_i. Если процесс P_i смог провести сокращение графа, то никакие дальнейшие сокращения не являются необходимыми.

Ограничения, накладываемые на распределители, на число ресурсов, запрошенных одновременно, и на количество единиц ресурсов, приводят к более простым условиям для тупика.

Пучок, или *узел*, в ориентированном графе G = <X, E> — это подмножество вершин Z I X, таких что ∀ x I Z, P(x) = Z, то есть потомством каждой вершины из Z является само множество Z. Каждая вершина в узле достижима из каждой другой вершины этого узла, и узел есть максимальное подмножество с этим свойством. Поясним сказанное рис. 8.10.

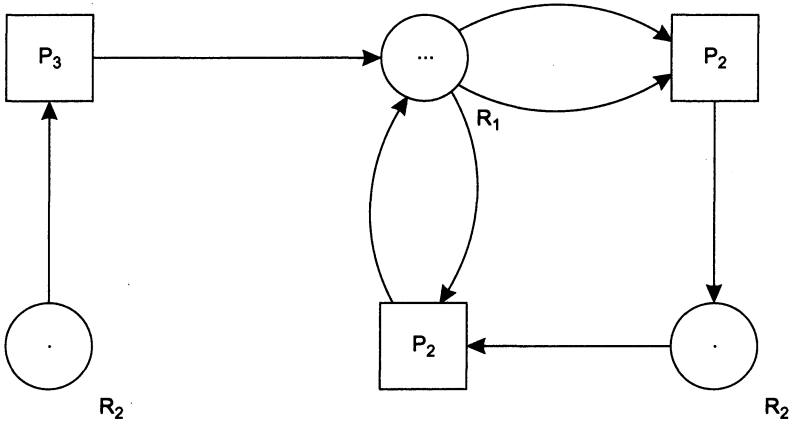


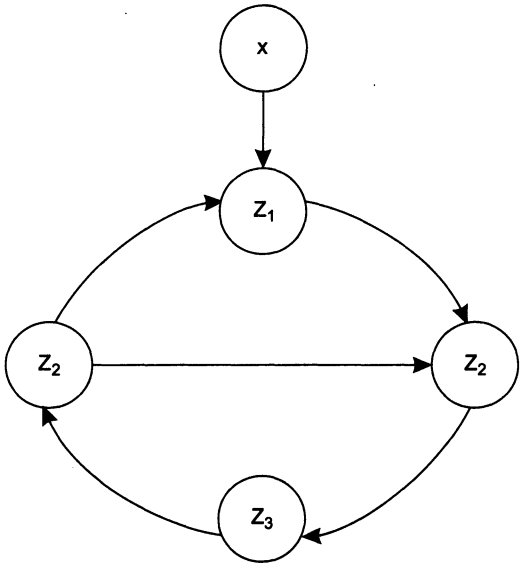
Рис. 8.10. Пример узла в модели Холта

Следует заметить, что наличие цикла — это необходимое, но не достаточное условие для узла. Так, на рис. 8.11 изображены два подграфа. Подграф *а* представляет собой пучок (узел), тогда как подграф *б* представляет собой цикл, но узлом не является. В узел должны входить дуги, но они не должны из него выходить.

Если состояние системы таково, что удовлетворены все запросы, которые могут быть удовлетворены, то существует простое достаточное условие существования тупика. Эта ситуация возникает, если распределители ресурсов не откладывают запросы, которые могут быть удовлетворены, а выполняют их по возможности немедленно (большинство распределителей следуют именно этой дисциплине).

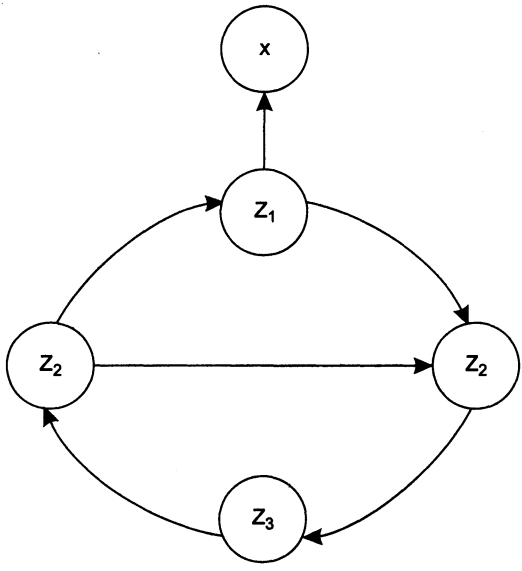
Состояние называется *фиксированным*, если каждый процесс, выдавший запрос, заблокирован.

Третья теорема: если состояние системы фиксированное (все процессы, имеющие запросы, удовлетворены), то наличие узла в соответствующем графе повторно используемых ресурсов является достаточным условием тупика.



Узел $Z = \{Z_1, Z_2, Z_3, Z_4\}$

а



Цикл $Z = \{Z_1, Z_2, Z_3, Z_4\}$, но не узел

б

Рис. 8.11. Узел и цикл в ориентированном графе

Для доказательства предположим, что граф содержит узел Z. Тогда все процессы в этом узле должны быть заблокированы только по ресурсам, принадлежащим Z, так как никакие ребра не могут выходить из Z по определению. Аналогично, по той же самой причине все распределенные ресурсы узла Z принадлежат процессам из Z. Наконец, все процессы в Z должны быть заблокированы согласно условию фиксированности и определению узла. Следовательно, все процессы в узле Z должны быть в тупике.

Допустим, что каждый ресурс имеет единичную емкость (по одной единице ресурса), то есть $|R_i| = 1$, (i = 1, 2, ..., m). При этом ограничении наличие цикла также становится достаточным условием тупика.

Четвертая теорема: граф повторно используемых ресурсов с единичной емкостью указывает на состояние тупика тогда и только тогда, когда он содержит цикл.

Необходимость цикла доказывает первая теорема. Для доказательства достаточности допустим, что граф содержит цикл, и рассмотрим только лишь процессы и ресурсы, принадлежащие циклу. Так как каждая вершина-процесс должна иметь входящее и исходящее ребра, она должна выдать запрос на некоторый ресурс, принадлежащий циклу, и должна удерживать некоторый ресурс, принадлежащий тому же циклу. Аналогично, каждый ресурс единичной емкости в цикле должен быть захвачен некоторым процессом. Следовательно, каждый процесс в цикле блокируется на ресурсе, который может быть освобожден только некоторым процессом из этого цикла; поэтому процессы в цикле находятся в тупике.

Чтобы обнаружить тупик в случае ресурса единичной емкости, мы должны просто проверить граф повторно используемых ресурсов на наличие циклов.

Алгоритм обнаружения тупика по наличию замкнутой цепочки запросов

Итак, распознавание тупика может быть основано на анализе модели распределения ресурсов. Один из алгоритмов, основанный на методе обнаружения замкнутой цепи запросов, был разработан сотрудниками фирмы IBM и применялся в одной из ОС этой компании. Он использует информацию о состоянии системы, содержащуюся в двух таблицах: таблице текущего распределения (назначения) ресурсов RATBL и таблице заблокированных процессов PWTBL (для каждого вида ресурса может быть свой список заблокированных процессов). При каждом запросе на получение или освобождение ресурсов содержимое этих таблиц модифицируется, а запрос анализируется в соответствии со следующим алгоритмом [22].

1. Начало алгоритма. Приходит запрос от процесса с номером J на занятый ресурс с номером I.
2. Поместить номер ресурса I в таблицу PWTBL в строке с номером процесса J.
3. Использовать I в качестве смещения в таблице RATBL, чтобы найти номер процесса K, который владеет ресурсом.
4. Использовать K в качестве смещения в таблице PWTBL.
5. Проверить, ждет ли процесс с номером K освобождения какого-либо ресурса с номером I'. Если нет, то перейти к шагу 6, в противном случае — к шагу 7.

6. Перевести процесс с номером J в состояние ожидания и выйти из алгоритма.
7. Использовать I' в качестве смещения в таблице RATBL, чтобы найти номер K' блокирующего его процесса.
8. Проверить, что K' = J. Если нет, перейти к шагу 9, в противном случае — к шагу 11.
9. Проверить, вся ли таблица PWTBL просмотрена. Если да, то перейти к шагу 6, в противном случае — к шагу 10.
10. Присвоить K := K' и перейти к шагу 4.
11. Сделать вывод о наличии тупика с последующим восстановлением.
12. Конец алгоритма.

Для иллюстрации описанного алгоритма распознавания тупика рассмотрим следующую последовательность событий.

1. Процесс P1 занимает ресурс R1.
2. Процесс P2 занимает ресурс R3.
3. Процесс P3 занимает ресурс R2.
4. Процесс P2 занимает ресурс R4.
5. Процесс P1 занимает ресурс R5.

   В результате таблица распределения ресурсов (RATBL) принимает такой вид, как показано в табл. 8.3.

Таблица 8.3. Таблица распределения ресурсов

Ресурсы	Процессы
R1	P1
R2	P3
R3	P2
R4	P2
R5	P1

6. Пусть процесс P1 пытается занять ресурс R3, поэтому в соответствии с описанным алгоритмом J = 1, I = 3, K = 2. Процесс K не ждет никакого ресурса, поэтому процесс P1 блокируется по ресурсу R3.
7. Далее, пусть процесс P2 пытается занять ресурс R2: J = 3, I = 2, K = 3. Процесс с номером K=3 не ждет никакого ресурса, поэтому процесс P2 блокируется по ресурсу R2.
8. И наконец, пусть процесс P3 пытается обратиться к ресурсу R5: J = 3, I = 5, K = 1, I' = 3, K' = 2, K' <> J, поэтому берем K = 2, I' = 2, K' = 3.

В этом случае K' = J, то есть тупик определен. Таблица заблокированных процессов (PWTBL) теперь будет выглядеть так, как показано в табл. 8.4.

Равенство J = K' означает, что существует замкнутая цепь взаимоисключающих и ожидающих процессов, то есть выполняются все четыре условия существования тупика.

Таблица 8.4. Таблица заблокированных ресурсов

Процесс	Ресурс
P1	R3
P2	R2
P3	R5

Для описанного нами примера можно построить модель Холта, как показано на рис. 8.12. Здесь пронумерованы дуги запросов, которые процессы последовательно генерировали в соответствии с нашим примером. Из рисунка сразу видно, что в результате такой последовательности запросов образовалась замкнутая цепочка: (8, 5, 6, 2, 7, 3), что и говорит о существовании тупика.

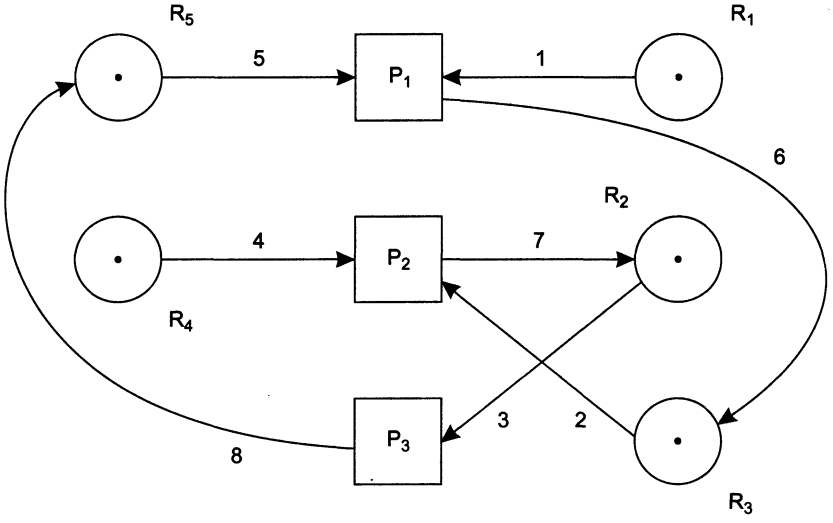


Рис. 8.12. Граф распределения ресурсов

Распознавание тупика требует дальнейшего восстановления вычислений.

Восстановление можно интерпретировать как запрет постоянного пребывания в опасном состоянии. Существуют следующие методы восстановления:

- принудительный перезапуск системы, характеризующийся потерей информации обо всех процессах, существовавших до перезапуска;
- принудительное завершение процессов, находящихся в тупике;
- принудительное последовательное завершение процессов, находящихся в тупике, с последующим вызовом алгоритма распознавания после каждого завершения до исчезновения тупика;
- перезапуск процессов, находящихся в тупике, с некоторой контрольной точки, то есть из состояния, предшествовавшего запросу на ресурс;
- перераспределение ресурсов с последующим последовательным перезапуском процессов, находящихся в тупике.

Основные издержки восстановления составляют потери времени на повторные вычисления, которые могут оказаться весьма существенными. К сожалению, в ряде случаев восстановление может стать невозможным, например исходные данные, поступившие с каких-либо датчиков, могут измениться, тогда предыдущие значения будут безвозвратно потеряны.

Контрольные вопросы и задачи

1. Что такое тупиковое состояние? Приведите несколько примеров возникновения тупиковой ситуации.
2. Что является причиной возникновения тупиков на ресурсах типа SR? Перечислите условия, при которых возникает тупик.
3. Приведите пример графа повторно используемых ресурсов. Что позволяет сделать эта модель Холта?
4. Приведите пример теоретико-множественного описания сети Петри.
5. Что такое маркировка сети Петри? Что представляет собой пространство возможных состояний сети Петри?
6. Приведите пример графического представления сети Петри.
7. Что следует предпринять для реализации стратегии предотвращения тупиковых ситуаций? Какие реальные проблемы при этом возникают?
8. Что представляет собой «обход тупика»? Приведите алгоритм банкира Дейкстры. Почему на практике невозможно воспользоваться алгоритмом банкира для борьбы с тупиковыми ситуациями?
9. Что такое «опасное состояние»? Приведите пример опасного состояния на модели состояний системы.
10. Опишите метод обнаружения тупика посредством редукции графа повторно используемых ресурсов.
11. Опишите алгоритм обнаружения тупика по наличию замкнутой цепочки запросов.

Глава 9. Архитектура операционных систем

Как комплекс системных управляющих и обрабатывающих программ (см. главу 1), операционная система представляет собой очень сложный конгломерат взаимосвязанных программных модулей и структур данных, которые должны обеспечивать надежное и эффективное выполнение вычислений. Большинство потенциальных возможностей операционной системы, ее технические и потребительские параметры — все это во многом определяется архитектурой системы — ее структурой и основными принципами построения.

В главе 1 мы упомянули несколько наиболее распространенных классификаций. Очевидно, что системы, ориентированные на диалог, должны иметь иные стратегию обслуживания и дисциплину диспетчеризации, чем системы пакетной обработки. Диалоговое взаимодействие предполагает реализацию развитой интерфейсной подсистемы, обеспечивающей взаимодействие пользователя с компьютером. Это отличие сказывается и на особенностях построения систем. Очевидно, что для диалоговых операционных систем необходимо предусмотреть множество механизмов, которые позволят пользователям эффективно управлять своими вычислениями.

Аналогично, и системы, реализующие мультизадачный режим работы, отличаются по своему строению от однозадачных систем. Если система допускает работу нескольких пользователей, то желательно иметь достаточно развитую подсистему информационной безопасности. А это, в свою очередь, налагает определенные требования и на идеологию построения операционной системы, и на выбор конкретных механизмов, помогающих реализовать защиту информационных ресурсов и ввести ограничения на доступ к другим видам ресурсов. Поскольку операционные системы помимо функций организации вычислений и организации интерфейса пользователя предоставляют интерфейсы для взаимодействия программ с операционной системой, мы в этой главе рассмотрим и интерфейсы прикладного программирования.

Основные принципы построения операционных систем

Среди множества принципов построения операционных систем перечислим несколько наиболее важных: принцип модульности, принцип виртуализации, принципы мобильности (переносимости) и совместимости, принцип открытости, принцип генерации операционной системы из программных компонентов и некоторые другие.

Принцип модульности

Операционная система строится из множества программных модулей. Под *модулем* в общем случае понимают функционально законченный элемент системы, выполненный в соответствии с принятыми межмодульными интерфейсами. По своему определению модуль предполагает легкий способ его замены другим при наличии заданных интерфейсов. Способы обособления составных частей операционной системы в отдельные модули могут быть существенно разными, но чаще всего разделение происходит именно по функциональному признаку. В значительной степени разделение системы на модули определяется используемым методом проектирования системы (снизу вверх или наоборот).

Особо важное значение при построении операционных систем имеют *привилегированные*, *повторно входимые* и *реентерабельные* модули, ибо они позволяют более эффективно использовать ресурсы вычислительной системы. Как мы уже знаем (см. главу 1), свойство реентерабельности может быть достигнуто различными способами, но чаще всего используются механизмы динамического выделения памяти под переменные для нового вычислительного процесса (задачи). В некоторых системах реентерабельность программы получают автоматически. Этого можно достичь благодаря неизменяемости кодовых частей программ при исполнении, а также автоматическому распределению регистров, автоматическому отделению кодовых частей программ от данных и помещению последних в системную область памяти, которая распределяется по запросам от выполняющихся задач. Естественно, что для этого необходима соответствующая аппаратная поддержка. В других случаях это достигается программистами за счет использования специальных системных модулей.

Принцип модульности отражает технологические и эксплуатационные свойства системы. Наибольший эффект от его использования достижим в случае, когда принцип распространен одновременно на операционную систему, прикладные программы и аппаратуру. Принцип модульности является одним из основных в UNIX-системах.

Во всех операционных системах можно выделить некоторую часть наиболее важных управляющих модулей, которые должны постоянно находиться в оперативной памяти для более скорой реакции системы на возникающие события и более эффективной организации вычислительных процессов. Эти модули вместе с некоторыми системными структурами данных, необходимыми для функционирования операционной системы, образуют так называемое *ядро операционной системы*, так как это действительно ее самая главная, центральная часть, основа системы.

При формировании состава ядра требуется удовлетворить двум противоречивым требованиям. В состав ядра должны войти наиболее часто используемые системные модули. Количество модулей должно быть таким, чтобы объем памяти, занимаемый ядром, был не слишком большим. В его состав, как правило, входят модули по управлению системой прерываний, средства по переводу программ из состояния счета в состояние ожидания, готовности и обратно, средства по распределению основных ресурсов, таких как оперативная память и процессор. В главе 1 мы уже упоминали, что операционные системы могут быть *микроядерными* и *макроядерными* (*монолитными*). В микроядерных операционных системах само ядро очень компактно, а остальные модули вызываются из ядра как сервисные. При этом сервисные модули могут размещаться и в оперативной памяти. В противоположность микроядерным в макроядерных операционных системах главная супервизорная часть включает в себя большое количество модулей. Более подробно о микроядерных и макроядерных операционных системах см. далее.

Помимо программных модулей, входящих в состав ядра и постоянно располагающихся в оперативной памяти, может быть много других системных программных модулей, которые получают название транзитных. Транзитные программные модули загружаются в оперативную память только при необходимости и в случае отсутствия свободного пространства могут быть замещены другими транзитными модулями. В качестве синонима термина «транзитный» можно использовать термин «диск-резидентный».

Принцип особого режима работы

Ядро операционной системы и низкоуровневые драйверы, управляющие работой каналов и устройств ввода-вывода, должны работать в специальном режиме работы процессора. Это необходимо по нескольким причинам. Во-первых, введение специального режима работы процессора, в котором должен исполняться только код операционной системы, позволяет существенно повысить надежность выполнения вычислений. Это касается выполнения как управляющих функций самой операционной системы, так и прикладных задач пользователей. Категорически нельзя допускать, чтобы какая-нибудь прикладная программа могла вмешиваться (преднамеренно или в связи с появлением ошибок вычислений) в вычисления, связанные с супервизорной частью операционной системы. Во-вторых, ряд функций должен выполняться исключительно централизованно, под управлением операционной системы. К этим функциям мы, прежде всего, должны отнести функции, связанные с управлением процессами ввода-вывода данных. Вспомните основные принципы организации ввода-вывода (см. главу 5): *все операции ввода-вывода данных объявляются привилегированными*. Это легче всего сделать, если процессор может работать, как минимум, в двух режимах: привилегированном (режим супервизора) и пользовательском. В первом режиме процессор может выполнять все команды, тогда как в пользовательском набор разрешенных команд ограничен. Естественно, что помимо запрета на выполнение команд ввода-вывода в пользовательском режиме работы процессор не должен позволять обращаться к своим специальным системным регистрам — эти регистры должны быть доступны только

в привилегированном режиме, то есть исключительно супервизорному коду самой операционной системы. Попытка выполнить запрещенную команду или обратиться к запрещенному регистру должна вызывать прерывание (исключение), и центральный процессор должен быть предоставлен супервизорной части операционной системы для управления выполняющимися вычислениями.

Поскольку любая программа требует операций ввода-вывода, прикладные программы для выполнения этих (и некоторых других) операций обращаются к супервизорной части операционной системы (модуль супервизора иногда называют *супервизором задач*) с соответствующим *запросом*. При этом процессор должен переключиться в привилегированный режим работы. Чтобы программы не могли произвольным образом обращаться к супервизорному коду, который работает в привилегированном режиме, им предоставляется возможность обращаться к нему в строгом соответствии с принятыми правилами. Каждый запрос имеет свой идентификатор и должен сопровождаться соответствующим количеством параметров, уточняющих запрашиваемую у операционной системы функцию (операцию). Поэтому супервизор задач при получении запроса сначала его тщательно проверяет. Если запрос корректный и программа имеет право с ним обращаться, то запрос на выполнение операции, как правило, передается соответствующему модулю операционной системы. Множество запросов к операционной системе образует соответствующий системный *интерфейс прикладного программирования* (Application Program Interface, API).

Принцип виртуализации

В наше время уже не требуется пояснять значение слова «виртуальный», ибо о виртуальных мирах, о виртуальной реальности знают даже дети. Принцип виртуализации нынче используется практически в любой операционной системе. Виртуализация ресурсов позволяет не только организовать разделение тех ресурсов между вычислительными процессами, которые не должны разделяться. Виртуализация позволяет абстрагироваться от конкретных ресурсов, максимально обобщить их свойства и работать с некоторой абстракцией, вобравшей в себя наиболее значимые особенности. Этот принцип позволяет представить структуру системы в виде определенного набора планировщиков процессов и распределителей ресурсов (мониторов) и использовать единую централизованную схему распределения ресурсов.

Следует заметить, что сама операционная система существенно изменяет наши представления о компьютере. Она виртуализирует его, добавляя ему функциональности, удобства управления, предоставляя средства организации параллельных вычислений и т. д. Именно благодаря операционной системе мы воспринимаем компьютер совершенно иначе, чем без нее.

Наиболее законченным и естественным проявлением концепции виртуальности является понятие *виртуальной машины*. По сути, любая операционная система, являясь средством распределения ресурсов и организуя по определенным правилам управление процессами, скрывает от пользователя и его приложений реальные аппаратные и иные ресурсы, заменяя их некоторой абстракцией. В результате

пользователи видят и используют виртуальную машину как некое устройство, способное воспринимать их программы, написанные на определенном языке программирования, выполнять их и выдавать результаты на виртуальные устройства, которые связаны с реально существующими в данной вычислительной системе. При таком языковом представлении пользователя совершенно не интересует реальная конфигурация вычислительной системы, способы эффективного использования ее компонентов и подсистем. Он мыслит и работает с машиной в терминах используемого им языка.

Чаще виртуальная машина, предоставляемая пользователю, воспроизводит архитектуру реальной машины, но архитектурные элементы в таком представлении выступают с новыми или улучшенными характеристиками, часто упрощающими работу с системой. Характеристики могут быть произвольными, но чаще всего пользователи желают иметь собственную «идеализированную» по архитектурным характеристикам машину в следующем составе.

- Единообразная по логике работы память (виртуальная) достаточного для выполнения приложений объема. Организация работы с информацией в такой памяти производится в терминах работы с сегментами данных на уровне выбранного пользователем языка программирования.
- Произвольное количество процессоров (виртуальных), способных работать параллельно и взаимодействовать во время работы. Способы управления процессорами, в том числе синхронизация и информационные взаимодействия, реализованы и доступны пользователям с уровня используемого языка в терминах управления процессами.
- Произвольное количество внешних устройств (виртуальных), способных работать с памятью виртуальной машины параллельно или последовательно, асинхронно или синхронно по отношению к работе того или иного виртуального процессора, которые инициируют работу этих устройств. Информация, передаваемая или хранимая на виртуальных устройствах, не ограничена допустимыми размерами. Доступ к такой информации осуществляется на основе либо последовательного, либо прямого способа доступа в терминах соответствующей системы управления файлами. Предусмотрено расширение информационных структур данных, хранимых на виртуальных устройствах.

Степень приближения к «идеальной» виртуальной машине может быть большей или меньшей в каждом конкретном случае. Чем больше виртуальная машина, реализуемая средствами операционной системы на базе конкретной аппаратуры компьютера, приближена к «идеальной» по характеристикам машине и, следовательно, чем больше ее архитектурно-логические характеристики отличны от реально существующих, тем больше степень ее виртуальности.

Одним из важнейших результатов принципа виртуализации является возможность организации выполнения в операционной системе приложений, разработанных для другой операционной системы, имеющей совсем другой интерфейс прикладного программирования. Другими словами, речь идет об организации нескольких операционных сред, о чем мы уже говорили в главе 1. Реализация этого принципа позволяет операционной системе иметь очень сильное преимущество перед другими

операционными системами, не имеющими такой возможности. Примером реализации принципа виртуализации может служить VDM-машина (Virtual DOS Machine) — защищенная подсистема, предоставляющая полную среду типа MS DOS и консоль для выполнения DOS-приложений. Как правило, параллельно может выполняться практически произвольное число DOS-приложений, каждое в своей VDM-машине. Такие VDM-машины имеются и в операционных системах Windows[1] компании Microsoft, в OS/2, в Linux.

Одним из аспектов общего принципа виртуализации является независимость программ от внешних устройств, хотя иногда эту особенность выделяют особенно и называют принципом. Она заключается в том, что связь программ с конкретными устройствами производится не в процессе создания программы, а в период планирования ее исполнения. В результате перекомпиляция при работе программы с новым устройством, на котором располагаются данные, не требуется. Этот принцип позволяет одинаково осуществлять операции управления внешними устройствами независимо от их конкретных физических характеристик. Например, программе, содержащей операции обработки последовательного набора данных, безразлично, на каком носителе эти данные будут располагаться. Смена носителя и данных, размещаемых на них (при неизменности структурных характеристик данных), не привнесет каких-либо изменений в программу, если в системе реализован принцип независимости программ от внешних устройств. Независимость программ от внешних устройств реализуется в подавляющем большинстве операционных систем общего применения. Ярким примером такого подхода являются операционные системы с общим названием UNIX. Реализована такая независимость и в большинстве современных операционных систем для персональных компьютеров.

Например, в системах Windows все аппаратные ресурсы полностью виртуализированы, и прямой доступ к ним со стороны прикладных (и системных обрабатывающих) программ однозначно запрещен. В системах Windows NT/2000/XP даже были введены понятия *HAL* (Hardware Abstraction Layer — уровень абстрагирования аппаратуры) и *HEL* (Hardware Emulation Layer — уровень эмуляции аппаратуры), и этот шаг очень помогает в реализации идей переносимости (мобильности) операционной системы.

Принцип мобильности

Мобильность, или переносимость, означает возможность и легкость переноса операционной системы на другую аппаратную платформу. Мобильная операционная система обычно разрабатывается с помощью специального языка высокого уровня, предназначенного для создания системного программного обеспечения. Такой язык помимо поддержки высокоуровневых операторов, типов данных и модульных конструкций должен позволять непосредственно использовать аппаратные возможности и особенности процессора. Кроме этого, такой язык должен быть широко распространенным и реализованным в виде систем программирования,

[1] Не все операционные системы компании Microsoft, в названии которых слово Windows является основным, поддерживают VDM-машины. В частности, такой возможности нет в системе Windows ME.

которые либо уже имеются на целевой платформе, либо позволяют получать программные коды для целевого компьютера. Другими словами, этот язык системного программирования должен быть достаточно распространенным и технологичным. Одним из таких языков является язык C. В последние годы язык C++ также стал использоваться для этих целей, поскольку идеи объектно-ориентированного программирования оказались плодотворными не только для прикладного, но и для системного программирования. Большинство современных операционных систем были созданы именно как объектно-ориентированные.

Обеспечить переносимость операционной системы достаточно сложно. Дело в том, что архитектуры разных процессоров могут очень сильно различаться. У них может быть разное количество рабочих регистров, причем часть регистров может оказаться контекстно-зависимыми, как это имеет место в процессорах с архитектурой ia32. Различия могут быть и в реализации адресации. Более того, для операционной системы важной является не только архитектура центрального процессора, но и архитектура компьютера в целом, ибо важнейшую роль играет подсистема ввода-вывода, а она строится на дополнительных (по отношению к центральному процессору) аппаратных средствах. В таких условиях сделать эффективным код операционной системы при условии создания его на языке типа C/C++ невозможно. Поэтому часть программных модулей, которые более всего зависят от аппаратных особенностей процессора, от типов поддерживаемых данных, способов адресации, системы команд и других важнейших моментов, разрабатывается на языке ассемблера. Очевидно, что модули, написанные на языке ассемблера, при переносе операционной системы на процессор с иной архитектурой должны быть написаны заново. Зато остальная (большая) часть кода операционной системы может быть просто перекомпилирована под целевой процессор. Именно по этому принципу в свое время была создана операционная система UNIX. Относительная легкость переноса этой системы на другие компьютеры позволила сделать ее одной из самых распространенных. Для обеспечения мобильности был даже создан стандарт на интерфейс прикладного программирования, названный POSIX (Portable Operating System Interface for Computer Environments — интерфейс прикладного программирования для переносимых операционных систем).

К сожалению, на самом деле далеко не все операционные системы семейства UNIX допускают относительно простую переносимость созданного для них программного обеспечения, хотя сами они и поддерживают такую переносимость. Основная причина тому — отход от единого стандарта API — POSIX. Очевидно, что платой за универсальность, прежде всего, является потеря производительности при выполнении операций ввода-вывода и вычислений, связанных с этими операциями. Поэтому ряд разработчиков шли и до сих пор идут на отказ от принципа мобильности, поскольку не всегда следование этому принципу экономически оправдано.

Если при разработке операционной системы сразу не следовать принципу мобильности, то в последующем очень трудно обеспечить перенос на другую платформу как самой операционной системы, так и программного обеспечения, созданного для нее. Например, компания IBM потратила долгие годы на перенос своей операционной системы OS/2, созданной для персональных компьютеров с процессора-

ми архитектуры ia32, на платформу PowerPC. Но даже если изначально в спецификации на операционную систему заложить требование легкой переносимости, это не значит, что его в последующем будет просто реализовать. Подтверждением тому является тот же проект OS/2-Windows NT. Как известно, проект Windows NT обеспечивал работу этой операционной системы на процессорах с архитектурой ia32, MIPS, Alpha (DEC), PowerPC. Однако в последующем трудности с реализацией этого принципа привели к тому, что нынешние версии операционных систем класса Windows NT (Windows 2000/XP) уже создаются только для процессоров с архитектурой ia32 и не поддерживают MIPS, Alpha и PowerPC.

Принцип совместимости

Одним из аспектов совместимости является способность операционной системы выполнять программы, написанные для других систем или для более ранних версий данной операционной системы, а также для другой аппаратной платформы.

Необходимо разделять вопросы двоичной совместимости и совместимости на уровне исходных текстов приложений. Двоичная совместимость достигается в том случае, когда можно взять исполняемую программу и запустить ее на выполнение на другой операционной системе. Для этого необходимы: совместимость на уровне команд процессора, совместимость на уровне системных вызовов и даже на уровне библиотечных вызовов, если они являются динамически связываемыми.

Совместимость на уровне исходных текстов требует наличия соответствующего транслятора в составе системного программного обеспечения, а также совместимости на уровне библиотек и системных вызовов. При этом необходима перекомпиляция имеющихся исходных текстов в новый выполняемый модуль.

Гораздо сложнее достичь двоичной совместимости между процессорами, основанными на разных архитектурах. Для того чтобы один компьютер выполнял программы другого (например, программу для персонального компьютера типа IBM PC хочется выполнять на компьютере типа Mac от фирмы Apple), этот компьютер должен работать с машинными командами, которые ему изначально непонятны. Например, процессор типа PowerPC на Mac должен исполнять двоичный код, предназначенный для процессора i80x86. Процессор 80x86 имеет свои собственные дешифратор команд, регистры и внутреннюю архитектуру. Процессор PowerPC имеет другую архитектуру, он не понимает непосредственно двоичный код 80x86, поэтому должен выбрать каждую команду, декодировать ее, чтобы определить, для чего она предназначена, а затем выполнить эквивалентную подпрограмму, написанную для PowerPC. К тому же у PowerPC нет в точности таких же регистров, флагов и внутреннего арифметико-логического устройства, как в 80x86, поэтому он должен эмулировать все эти элементы с использованием своих регистров или памяти. И он должен тщательно воспроизводить результаты каждой команды, что требует специально написанных подпрограмм для PowerPC, гарантирующих, что состояние эмулируемых регистров и флагов после выполнения каждой команды будет в точности таким же, как и на реальном процессоре 80x86. Выходом в таких случаях является использование так называемых прикладных сред, или эмуляторов. Учитывая, что основную часть программы, как правило, составляют *вызовы библио-*

течных функций, прикладная среда имитирует библиотечные функции целиком, используя заранее написанную библиотеку функций аналогичного назначения, а остальные команды эмулирует каждую по отдельности.

Одним из средств обеспечения совместимости программных и пользовательских интерфейсов является соответствие стандартам POSIX. Эти стандарты позволяют создавать программы в стиле UNIX, которые впоследствии могут легко переноситься из одной системы в другую.

Принцип генерируемости

Согласно принципу генерируемости исходное представление центральной системной управляющей части операционной системы (ее ядра и основных компонентов, которые должны постоянно находиться в оперативной памяти) должно обеспечивать возможность настройки, исходя из конкретной конфигурации конкретного вычислительного комплекса и круга решаемых задач. Под генерацией операционной системы понимается ее сборка (компоновка) из отдельных программных модулей. В результате генерации получают скомпонованные двоичные коды операционной системы и построенные системные таблицы, отражающие конкретную конфигурацию компьютера. Эта процедура проводится редко перед достаточно протяженным периодом эксплуатации операционной системы. Процесс генерации осуществляется с помощью специальной программы-генератора и соответствующего входного языка для этой программы, позволяющего описывать программные возможности системы и конфигурацию машины. В результате генерации получается полная версия операционной системы. Сгенерированная версия операционной системы представляет собой совокупность системных наборов модулей и данных.

Упомянутый раньше принцип модульности положительно проявляется при генерации операционной системы. Он существенно упрощает ее настройку на требуемую конфигурацию вычислительной системы. В наши дни при использовании персональных компьютеров с принципом генерируемости операционной системы можно столкнуться разве что при работе с Linux. В этой UNIX-системе имеется возможность не только использовать какое-либо готовое ядро операционной системы, но и самому сгенерировать (скомпилировать) такое ядро, которое будет оптимальным для данного конкретного персонального компьютера и решаемых на нем задач. Кроме генерации ядра в Linux имеется возможность указать и набор подгружаемых драйверов и служб, то есть часть функций может реализовываться модулями, непосредственно входящими в ядро системы, а часть — модулями, имеющими статус подгружаемых, транзитных.

В остальных современных распространенных операционных системах, в том числе и для персональных компьютеров, конфигурирование системы под соответствующий состав оборудования осуществляется на этапе установки, причем в большинстве случаев не представляется возможным серьезно вмешаться в этот процесс. В дальнейшем, при эксплуатации компьютера, можно изменить состав драйверов, служб, отдельных параметров и режимов работы. Как правило, внесение подобных изменений может быть осуществлено посредством редактирования конфигу-

рационного файла или реестра. Например, мы можем отключить ненужное устройство, заменить для какого-нибудь устройства драйвер, отключить или добавить ту или иную службу. Более того, для большей гибкости часто вводится механизм поддержки нескольких конфигураций. Например, такие популярные системы, как Windows 98 и Windows NT/2000/XP, предоставляют возможность создавать до девяти конфигураций. При загрузке операционной системы пользователю предоставляется возможность выбрать одну из имеющихся конфигураций. Таким образом, имея всего одну операционную систему, за счет нескольких различающихся конфигураций пользователь может получить несколько виртуальных систем, различающихся составом установленного (работающего) оборудования, драйверов и служб, и на выбор запускать одну из этих систем.

Принцип открытости

Открытая операционная система доступна для анализа как пользователям, так и системным специалистам, обслуживающим вычислительную систему. Наращиваемая (модифицируемая, развиваемая) операционная система позволяет не только использовать возможности генерации, но и вводить в ее состав новые модули, совершенствовать существующие и т. д. Другими словами, необходимо, чтобы можно было легко внести дополнения и изменения, если это потребуется, не нарушая целостности системы. Прекрасные возможности для расширения предоставляет подход к структурированию операционной системы по типу клиент-сервер с использованием микроядерной технологии. В соответствии с этим подходом операционная система строится как совокупность привилегированной управляющей программы и набора непривилегированных служб — «серверов». Основная часть операционной системы может оставаться неизменной, в то время как добавляются новые службы или изменяются старые.

Этот принцип иногда трактуют как расширяемость системы.

К открытым операционным системам прежде всего следует отнести UNIX-системы и, естественно, системы Linux.

Принцип обеспечения безопасности вычислений

Обеспечение безопасности при выполнении вычислений является желаемым свойством для любой многопользовательской системы. Правила безопасности определяют такие свойства, как защита ресурсов одного пользователя от других и установление квот по ресурсам для предотвращения захвата одним пользователем всех системных ресурсов (таких как память).

Обеспечение защиты информации от несанкционированного доступа является обязательной функцией многих операционных систем. Для решения этой проблемы чаще всего используется механизм учетных записей. Он предполагает проведение *аутентификации* пользователя при его регистрации на компьютере и последующую *авторизацию*, которая определяет уровень полномочий (прав) пользователя (об аутентификации и авторизации пользователей см. главу 1). Каждая учетная запись может входить в одну или несколько групп. Встроенные группы, как правило, определяют

права пользователей, тогда как создаваемые администратором группы (их называют группами безопасности) используются для определения разрешений в доступе пользователей к тем или иным ресурсам. Имеющиеся учетные записи хранятся в специальной базе данных, которая бывает доступна только для самой системы. Для этого файл базы данных с учетными записями открывается системой в монопольном режиме, и доступ к нему со стороны любого пользователя становится невозможным. Делается это для того, чтобы нельзя было получить базу данных с учетными записями. Если получить файл с учетными записями, то посредством его анализа можно было бы узнать пароль пользователя, по которому осуществляется аутентификация.

Во многих современных операционных системах гарантируется степень безопасности данных, соответствующая уровню C2 в системе стандартов США. Основы стандартов в области безопасности были заложены в документе «Критерии оценки надежных компьютерных систем». Этот документ, изданный в США в 1983 году. Национальным центром компьютерной безопасности (National Computer Security Center), часто называют Оранжевой книгой.

В соответствии с требованиями Оранжевой книги безопасной считается система, которая «посредством специальных механизмов защиты контролирует доступ к информации таким образом, что только имеющие соответствующие полномочия лица или процессы, выполняющиеся от их имени, могут получить доступ на чтение, запись, создание или удаление информации».

Иерархия уровней безопасности, приведенная в Оранжевой книге, помечает низший уровень безопасности как D, а высший — как A.

В класс D попадают системы, оценка которых выявила их несоответствие требованиям всех других классов.

Основными свойствами, характерными для систем класса C, являются наличие подсистемы учета событий, связанных с безопасностью, и избирательный контроль доступа. Класс (уровень) C делится на два подуровня: уровень C1 обеспечивает защиту данных от ошибок пользователей, но не от действий злоумышленников. На более строгом уровне C2 должны присутствовать:

- средства секретного входа, обеспечивающие идентификацию пользователей путем ввода уникального имени и пароля перед тем, как им будет разрешен доступ к системе;
- избирательный контроль доступа, требуемый на этом уровне, позволяет владельцу ресурса определить, кто имеет доступ к ресурсу и что он может с ним делать (владелец делает это путем предоставления разрешений доступа пользователю или группе пользователей);
- средства аудита (auditing) обеспечивают обнаружение и запись важных событий, связанных с безопасностью, или любых попыток создать системные ресурсы, получить доступ к ним или удалить их;
- защита памяти заключается в том, что память перед ее повторным использованием должна инициализироваться.

На этом уровне система не защищена от ошибок пользователя, но поведение его может быть проконтролировано по записям в журнале, оставленным средствами аудита.

Системы уровня B основаны на помеченных данных и распределении пользователей по категориям, то есть реализуют мандатный контроль доступа. Каждому пользователю присваивается рейтинг защиты, и он может получать доступ к данным только в соответствии с этим рейтингом. Этот уровень в отличие от уровня C защищает систему от ошибочного поведения пользователя.

Уровень A является самым высоким уровнем безопасности, он требует в дополнение ко всем требованиям уровня B выполнения формального математически обоснованного доказательства соответствия системы требованиям безопасности.

Различные коммерческие структуры (например, банки) особо выделяют необходимость учетной службы, аналогичной той, что предлагают государственные рекомендации C2. Любая деятельность, связанная с безопасностью, может быть отслежена и тем самым учтена. Это как раз то, что требует стандарт для систем класса C2 и что обычно нужно банкам. Однако коммерческие пользователи, как правило, не хотят расплачиваться производительностью за повышенный уровень безопасности. Уровень безопасности A занимает своими управляющими механизмами до 90 % процессорного времени, что, безусловно, в большинстве случаев неприемлемо. Более безопасные системы не только снижают эффективность, но и существенно ограничивают число доступных прикладных пакетов, которые соответствующим образом могут выполняться в подобной системе. Например, для операционной системы Solaris (версия UNIX) есть несколько тысяч приложений, а для ее аналога уровня B — только около ста.

Микроядерные операционные системы

В микроядерных операционных системах мы можем выделить центральный компактный модуль, относящийся к супервизорной части системы. Этот модуль имеет очень небольшие размеры и выполняет относительно небольшое количество управляющих функций, но позволяет передать управление на другие управляющие модули, которые и выполнят затребованную функцию. Микроядро — это минимальная главная (стержневая) часть операционной системы, служащая основой модульных и переносимых расширений. Микроядро само является модулем системного программного обеспечения, работающим в наиболее приоритетном состоянии компьютера и поддерживающим связи с остальной частью операционной системы, которая рассматривается как набор серверных приложений (служб).

В 90-е годы XX века было весьма распространенным убеждение, что большинство операционных систем следующих поколений будут строиться как микроядерные. Однако практика показывает, что это не совсем так. Разработчики желают иметь компактное микроядро, но при этом включить в него как можно больше функций, исполняемых непосредственно этим программным модулем. Ибо выполнение затребованной функции другим модулем, вызываемым из микроядра, приводит и к дополнительным задержкам, и к дополнительным сложностям. Более того, имеется масса разных мнений по поводу того, как следует организовывать службы операционной системы по отношению к микроядру; как проектировать драйверы устройств, чтобы добиться наибольшей эффективности, но сохранить функции

драйверов максимально независимыми от аппаратуры; следует ли выполнять операции, не относящиеся к ядру, в пространстве ядра или в пространстве пользователя; стоит ли сохранять программы имеющихся подсистем (например, UNIX) или лучше отбросить все и начать с нуля.

Основная идея, заложенная в технологию микроядра заключается в том, чтобы создать необходимую среду верхнего уровня иерархии, из которой можно легко получить доступ ко всем функциональным возможностям уровня аппаратного обеспечения. При этом микроядро является стартовой точкой для создания всех остальных модулей системы. Все эти остальные модули, реализующие необходимые системе функции, вызываются из микроядра и выполняют сервисную роль. При этом они получают статус обычного процесса или задачи. Можно сказать, что микроядерная архитектура соответствует технологии клиент-сервер. Именно эта технология позволяет в большей мере и с меньшими трудозатратами реализовать перечисленные выше принципы проектирования операционных систем.

Важнейшая задача разработки микроядра заключается в выборе базовых примитивов, которые должны находиться в микроядре для обеспечения необходимого и достаточного сервиса. В микроядре содержится и исполняется минимальное количество кода, необходимое для реализации основных системных вызовов. В число этих вызовов входят передача сообщений и организация другого общения между внешними по отношению к микроядру процессами, поддержка управления прерываниями, а также ряд других весьма немногочисленных функций. Остальные системные функции, характерные для «обычных» (не микроядерных) операционных систем, обеспечиваются как модульные дополнения-процессы, взаимодействующие главным образом между собой и осуществляющие взаимодействие посредством передачи сообщений.

Для большинства микроядерных операционных систем основой для такой архитектуры выступает технология микроядра Mach. Эта операционная система была создана в университете Карнеги Меллон, и многие разработчики брали с нее пример.

Исполняемые микроядром функции ограничены в целях сокращения его размеров и максимизации количества кода, работающего как прикладная программа. Микроядро включает только те функции, которые требуются в целях определения набора абстрактных сред обработки для прикладных программ и организации совместной работы приложений. В результате микроядро обеспечивает только пять различных типов сервисов:

- управление виртуальной памятью;
- поддержка заданий и потоков;
- взаимодействие между процессами (Inter-Process Communication, IPC);
- управление поддержкой ввода-вывода и прерываниями;
- сервисы хоста (host)[1] и процессора.

[1] Хост — главный компьютер. Нынче этим термином обозначают любой компьютер, имеющий IP-адрес.

Другие подсистемы и функции операционной системы, такие как файловые системы, поддержка внешних устройств и традиционные программные интерфейсы, оформляются как системные сервисы либо получают статус обычных обрабатывающих задач. Эти программы работают как приложения на микроядре.

С применением концепции нескольких потоков выполнения на одно задание микроядро создает прикладную среду, обеспечивающую использование мультипроцессоров; при этом совсем не обязательно, чтобы машина была мультипроцессорной: на однопроцессорной машине различные потоки просто выполняются в разное время. Вся поддержка, требуемая для мультипроцессорных машин, сконцентрирована в сравнительно малом и простом микроядре.

Благодаря своим небольшим размерам и способности поддерживать остальные службы в виде обычных процессов, выполняющихся вместе с прикладными программами, сами микроядра проще, чем ядра монолитных или модульных операционных систем. С микроядром супервизорная часть операционной системы разбивается на модульные части, которые могут быть сконфигурированы целым рядом способов, позволяя строить большие системы добавлением частей к меньшим. Например, каждый аппаратно-независимый нейтральный сервис логически отделен и может быть сконфигурирован различными способами. Микроядра также облегчают поддержку мультипроцессоров созданием стандартной программной среды, которая может использовать несколько процессоров, если они есть, однако если их нет, работает на одном. Специализированный код для мультипроцессоров ограничен самим микроядром. Более того, сети из общающихся между собой микроядер могут быть использованы для операционной системной поддержки возникающего класса массивно параллельных машин.

В некоторых случаях использование микроядерного подхода на практике сталкивается с определенными сложностями, что проявляется в некотором замедлении скорости выполнения системных вызовов при передаче сообщений через микроядро по сравнению с классическим подходом. С другой стороны, можно констатировать и обратное. Поскольку микроядра малы и в значительной степени оптимизированы, при соблюдении ряда условий они позволяют обеспечить характеристики реального времени, требующиеся для управления устройствами и для высокоскоростных коммуникаций. Наконец, хорошо структурированные микроядра обеспечивают изолирующий слой для аппаратных различий, которые не маскируются применением языков программирования высокого уровня. Таким образом, они упрощают перенесение кода и увеличивают уровень его повторного использования.

Наиболее ярким представителем микроядерных операционных систем является операционная система реального времени QNX. Микроядро QNX поддерживает только планирование и диспетчеризацию процессов, взаимодействие процессов, обработку прерываний и сетевые службы нижнего уровня (подробнее об ОС QNX см. в главе 10). Это микроядро обеспечивает всего лишь пару десятков системных вызовов, но благодаря этому оно может быть целиком размещено во внутреннем кэше даже таких процессоров, как Intel 486. Как известно, разные версии этой операционной системы имели и разные объемы ядер — от 8 до 46 Кбайт.

Чтобы построить минимальную систему QNX, требуется добавить к микроядру менеджер процессов, который создает процессы и управляет ими и памятью процессов. Чтобы операционная система QNX была применима не только во встроенных и бездисковых системах, нужно добавить файловую систему и менеджер устройств. Эти менеджеры исполняются вне пространства ядра, так что ядро остается небольшим.

Макроядерные операционные системы

В *макроядерных*, или *монолитных*, операционных системах ядро, состоящее из множества управляющих модулей и структур данных, не разделено на центральную часть и периферийные (по отношению к этой центральной части) модули. Ядро получается монолитным, неделимым. В этом смысле макроядерные операционные системы являются прямой противоположностью микроядерным. Можно согласиться с тем, как трактуется архитектура монолитных операционных систем в работах [29, 30]. В монолитной операционной системе, несмотря на ее возможную сильную структуризацию, очень трудно удалить один из уровней многоуровневой модульной структуры. Добавление новых функций и изменение существующих для монолитных операционных систем требует очень хорошего знания всей архитектуры операционной системы и чрезвычайно больших усилий.

Очень плодотворным оказался подход, основанный на модели *клиент-сервер*. Эта модель предполагает наличие программного компонента — потребителя какого-либо сервиса, или *клиента*, и программного компонента — поставщика этого сервиса, или *сервера*. Взаимодействие между клиентом и сервером стандартизируется, так что сервер может обслуживать клиентов, реализованных различными способами и, возможно, разными разработчиками. При этом главным требованием является то, чтобы использовался единообразный интерфейс. Инициатором обмена обычно является клиент, который посылает запрос на обслуживание серверу, находящемуся в состоянии ожидания запроса. Один и тот же программный компонент может быть клиентом по отношению к одному виду услуг и сервером для другого вида услуг. Модель клиент-сервер является скорее удобным концептуальным средством ясного представления функций того или иного программного элемента в той или иной ситуации, нежели технологией. Эта модель успешно применяется не только при построении операционных систем, но и на всех уровнях программного обеспечения, и имеет в некоторых случаях более узкий, специфический смысл, сохраняя, естественно, при этом все свои общие черты. Микроядерные операционные системы в полной мере используют модель клиент-сервер.

При поддержке монолитных операционных систем возникает ряд проблем, связанных с тем, что все компоненты макроядра работают в едином адресном пространстве. Во-первых, это опасность возникновения конфликта между различными частями ядра, во-вторых, сложность подключения к ядру новых драйверов. Преимущество микроядерной архитектуры перед макроядерной заключается в том, что каждый компонент системы представляет собой самостоятельный процесс, запуск или остановка которого не отражается на работоспособности остальных процессов.

Микроядерные операционные системы нынче разрабатываются чаще монолитных. Однако следует заметить, что использование технологии клиент-сервер — это еще не гарантия того, что операционная система станет микроядерной. В качестве подтверждения этому можно привести пример с операционными системами класса Windows NT, которые построены на идеологии клиент-сервер, но которые тем не менее трудно назвать микроядерными. Их «микроядро» имеет уже достаточно большой размер, приставка «микро» здесь вызывает улыбку. Хотя по своей архитектуре супервизорная часть этих систем без каких-либо условностей может быть отнесена к системам, построенным на базе модели клиент-сервер. Причем для последних версий операционных систем с общим названием NT (New Technology) еще более заметным является отход от микроядерной архитектуры, но сохранение принципа клиент-сервер во взаимодействиях между модулями управляющей (супервизорной) части. Для того чтобы согласиться с таким высказыванием, достаточно сравнить операционную систему QNX и операционные системы Windows NT/2000/XP.

Требования к операционным системам реального времени

Как известно, система реального времени (СРВ) должна давать отклик на любые непредсказуемые внешние воздействия в течение предсказуемого интервала времени. Для этого должны выполняться следующие требования.

- *Ограничение времени отклика.* После наступления события реакция на него гарантированно должна последовать до предустановленного крайнего срока. Отсутствие такого ограничения рассматривается как серьезный недостаток программного обеспечения.
- *Одновременность обработки.* Даже если наступает более одного события одновременно, все временные ограничения для всех событий должны быть выдержаны. Это означает, что системе реального времени должен быть присущ параллелизм, что достигается использованием нескольких процессоров и/или многозадачного подхода.

Примерами систем реального времени являются системы управления атомными электростанциями или какими-нибудь технологическими процессами, системы медицинского мониторинга, системы управления вооружением, системы космической навигации, системы разведки, системы управления лабораторными экспериментами, системы управления автомобильными двигателями, робототехника, телеметрические системы управления, системы антиблокировки тормозов, системы сигнализации — список в принципе бесконечен.

Иногда можно услышать из разговоров специалистов, что различают системы «мягкого» и «жесткого» реального времени. Различие между жесткой и мягкой СРВ зависит от требований к системе — система считается жесткой, если «нарушение временных ограничений недопустимо», и мягкой, если «нарушение временных ограничений нежелательно». В недалеком прошлом велось множество дискуссий о точном смысле терминов «жесткая» и «мягкая» СРВ. Можно даже показать, что

мягкая СРВ не является СРВ вовсе, ибо основное требование о соблюдении временных ограничений не выполнено. В действительности термин СРВ часто неправомерно применяют по отношению к быстрым системам.

Часто путают понятия СРВ и ОСРВ (операционная система реального времени), а также неправильно используют атрибуты «мягкая» и «жесткая», когда говорят, что та или иная ОСРВ мягкая или жесткая. Нет мягких или жестких операционных систем реального времени. ОСРВ может только служить основой для построения мягкой или жесткой СРВ. Сама по себе ОСРВ не препятствует тому, что ваша СРВ будет мягкой. Например, пусть вы решили создать СРВ, которая должна работать через Ethernet по протоколу TCP/IP. Такая система не может быть жесткой СРВ, поскольку сама сеть Ethernet в принципе непредсказуема вследствие использования случайного метода доступа к среде передачи данных, в отличие, например, от сети IBM Token Ring или ARC Net, в которых используются детерминированные методы доступа.

Итак, перечислим основные требования к ОСРВ.

Мультипрограммность и мультизадачность

Требование 1. Операционная система должна быть мультипрограммной и мультизадачной (*многопоточной* — multi-threaded), а также активно использовать прерывания для диспетчеризации. Как указывалось выше, ОСРВ должна быть предсказуемой. Это означает не то, что ОСРВ должна быть быстрой, а то, что максимальное время выполнения того или иного действия должно быть известно заранее и соответствовать требованиям приложения. Так, например, система Windows 3.11 даже на Pentium IV с частотой более 3000 МГц не может функционировать в качестве ОСРВ, ибо одно приложение может практически монопольно захватить центральный процессор и заблокировать систему для остальных вычислений.

В соответствии с первым требованием операционная система должна быть многопоточной на принципе абсолютного приоритета (прерываемой). То есть планировщик должен иметь возможность прервать любой поток выполнения и предоставить ресурс тому потоку, которому он более необходим. Операционная система (и аппаратура) должна также обеспечивать прерывания на уровне обработки прерываний.

Приоритеты задач

Требование 2. Должно существовать понятие приоритета потока (задачи). Проблема в том, чтобы определить, какой задаче ресурс требуется более всего. В идеальной ситуации ОСРВ отдает ресурс потоку или драйверу с ближайшим крайним сроком, что называется управлением временным ограничением (DeadLine DrivenOS). Чтобы реализовать это временное ограничение, операционная система должна знать, сколько времени требуется каждому из выполняющихся потоков для завершения. Операционных систем, построенных по этому принципу, практически нет, так как он слишком сложен для реализации. Поэтому разработчики операционных систем принимают иную точку зрения: вводится понятие уровня приоритета для задачи,

и временные ограничения сводятся к приоритетам. Так как умозрительные решения чреваты ошибками, показатели СРВ при этом снижаются. Чтобы более эффективно осуществить указанное преобразование ограничений, проектировщик может воспользоваться теорией расписаний или имитационным моделированием, хотя и это может оказаться бесполезным. Тем не менее, так как на сегодняшний день не имеется иного решения, без понятия приоритета потока не обойтись.

Наследование приоритетов

Требование 3. Должна существовать система наследования приоритетов. На самом деле именно этот механизм синхронизации и тот факт, что различные потоки выполнения используют одно и то же пространство памяти, отличают их от процессов. Как мы уже знаем, процессы не разделяют одно и то же пространство памяти. Так, например, старые версии UNIX не являются многопоточными. «Старая» UNIX — многозадачная ОС, где задачами являются процессы (а не потоки), которые сообщаются через каналы связи (pipes) и разделяемую память. Оба этих механизма используют файловую систему, а ее поведение непредсказуемо.

Комбинация приоритетов потоков и разделение ресурсов между ними приводит к другому явлению — классической проблеме инверсии приоритетов. Это можно проиллюстрировать на примере, где есть как минимум три потока. Когда поток низшего приоритета захватил ресурс, разделяемый с потоком высшего приоритета, и начал выполняться поток среднего приоритета, выполнение потока высшего приоритета будет приостановлено, пока не освободится ресурс и не отработает поток среднего приоритета. В этой ситуации время, необходимое для завершения потока высшего приоритета, зависит от нижних уровней приоритетов — это и есть инверсия приоритетов. Ясно, что в такой ситуации трудно выдержать ограничение на время исполнения.

Чтобы устранить такие инверсии, ОСРВ должна допускать наследование приоритета, то есть повышение уровня приоритета потока до уровня потока, который его вызывает. Наследование означает, что блокирующий ресурс поток наследует приоритет потока, который он блокирует (разумеется, это справедливо лишь в том случае, если блокируемый поток имеет более высокий приоритет).

Иногда можно услышать утверждение, что в грамотно спроектированной системе такая проблема не возникает. В случае сложных систем с этим нельзя согласиться. Единственный способ решения этой проблемы состоит в увеличении приоритета потока «вручную» прежде, чем ресурс окажется заблокированным. Разумеется, это возможно в случае, когда два потока разных приоритетов претендуют на один ресурс. В общем случае решения не существует.

Сихронизация процессов и задач

Требование 4. Операционная система должна обеспечивать мощные, надежные и удобные механизмы синхронизации задач. Так как задачи разделяют данные (ресурсы) и должны сообщаться друг с другом, представляется логичным существование механизмов блокирования и коммуникации. То есть необходимы механиз-

мы, гарантированно предоставляющие возможность оперативно обменяться сообщениями и синхросигналами между параллельно выполняющимися задачами и процессами. Эти системные механизмы должны быть всегда доступны процессам, требующим реального времени. Следовательно, системные ресурсы для их функционирования должны быть распределены заранее.

Предсказуемость

Требование 5. Поведение операционной системы должно быть известно и достаточно точно прогнозируемо. Времена выполнения системных вызовов и временные характеристики поведения системы в различных обстоятельствах должны быть известны разработчику. Поэтому создатель ОСРВ должен приводить следующие характеристики:

- латентную задержку прерывания, то есть время от момента прерывания до момента запуска задачи: она должна быть предсказуема и согласована с требованиями приложения (эта величина зависит от числа одновременно «висящих» прерываний);
- максимальное время выполнения каждого системного вызова (оно должно быть предсказуемо и не должно зависеть от числа объектов в системе);
- максимальное время маскирования прерываний драйверами и супервизорными модулями операционной системы.

Интерфейсы операционных систем

Напомним, что операционная система всегда выступает как интерфейс между аппаратурой компьютера и пользователем с его задачами. Под интерфейсами операционных систем здесь и далее следует понимать специальные интерфейсы системного и прикладного программирования (API), предназначенные для выполнения перечисленных ниже задач.

- Управление процессами, которое включает в себя следующий набор основных функций:
  - запуск, приостанов и снятие задачи с выполнения;
  - задание или изменение приоритета задачи;
  - взаимодействие задач между собой (механизмы сигналов, семафорные примитивы, очереди, конвейеры, почтовые ящики);
  - вызов удаленных процедур (Remote Procedure Call, RPC).
- Управление памятью:
  - запрос на выделение блока памяти;
  - освобождение памяти;
  - изменение параметров блока памяти (например, память может быть заблокирована процессом либо предоставлена в общий доступ);
  - отображение файлов на память (имеется не во всех системах).

- Управление вводом-выводом:
  - запрос на управление виртуальными устройствами (напомним, что управление вводом-выводом является привилегированной функцией самой операционной системы, и никакая из пользовательских задач не должна иметь возможности непосредственно управлять устройствами);
  - файловые операции (запросы к системе управления файлами на создание, изменение и удаление данных, организованных в файлы).

Здесь мы перечислили основные наборы функций, которые выполняются операционной системой по соответствующим запросам от задач. Что касается интерфейса пользователя с операционной системой, то он реализуется с помощью специальных программных модулей, которые принимают его команды на соответствующем языке (возможно, с использованием графического интерфейса) и транслируют их в обычные вызовы в соответствии с основным интерфейсом системы. Обычно эти модули называют интерпретатором команд. Так, например, функции такого интерпретатора в MS DOS выполняет модуль `COMMAND.COM`. Получив от пользователя команду, такой модуль после лексического и синтаксического анализа либо сам выполняет действие, либо, что случается чаще, обращается к другим модулям операционной системы, используя механизм API. Надо заметить, что в последние годы большую популярность получили *графические интерфейсы* (Graphical User Interface, GUI), в которых задействованы соответствующие манипуляторы типа мышь или трекбол (track-ball)[1]. Указание курсором на объект и щелчок или двойной щелчок на соответствующей кнопке мыши приводит к каким-либо действиям — запуску программы, ассоциированной с объектом, выбору и/или активизации меню и т. д. Можно сказать, что такая интерфейсная подсистема транслирует «команды» пользователя в обращения к операционной системе.

Поясним также, что управление GUI является частным случаем задачи управления вводом-выводом и не относится к функциям ядра операционной системы, хотя в ряде случаев разработчики операционной системы относят функции GUI к основному системному интерфейсу API.

Следует отметить, что имеются два основных подхода к управлению задачами. Так, в одних системах порождаемая задача наследует все ресурсы задачи-родителя, тогда как в других системах существуют равноправные отношения, и при порождении нового процесса ресурсы для него запрашиваются у операционной системы.

Обращения к операционной системе в соответствии с имеющимся интерфейсом API могут осуществляться как посредством вызова подпрограммы с передачей ей необходимых параметров, так и через механизм программных прерываний. Выбор метода реализации вызовов функций API должен определяться архитектурой платформы.

---

[1] Трекбол — специальный шарик, который в переносных компьютерах (NoteBook) размещается рядом с клавиатурой, прокручивается пальцами и служит для перемещения указателя мыши. В настоящее время гораздо чаще используют устройство, чувствительное к касанию (touchpad). С помощью такого устройства пользователь управляет указателем мыши, перемещая палец по специальной поверхности.

Так, например, в операционной системе MS DOS, которая разрабатывалась для однозадачного режима (поскольку процессор i80x86 не поддерживал мультипрограммирование), использовался механизм программных прерываний. При этом основной набор функций API был доступен через точку входа обработчика int 21h.

В более сложных системах имеется не одна точка входа, а множество — по количеству функций API. Так, в большинстве операционных систем используется метод вызова подпрограмм. В этом случае вызов сначала передается в модуль API, например в библиотеку времени выполнения (Run Time Library, RTL)[1], который перенаправляет его соответствующим обработчикам программных прерываний, входящим в состав операционной системы. Использование механизма прерываний вызвано, главным образом, тем, что при этом процессор переводится в режим супервизора.

Интерфейс прикладного программирования

Прежде всего, необходимо однозначно разделить общий термин *API* (Application Program Interface — интерфейс прикладного программирования) на следующие направления:

- API как интерфейс высокого уровня, принадлежащий к библиотекам RTL;
- API прикладных и системных программ, входящих в поставку операционной системы;
- прочие интерфейсы API.

Интерфейс прикладного программирования, как это и следует из названия, предназначен для использования прикладными программами системных ресурсов компьютера и реализуемых операционной системой разнообразных системных функций. API описывает совокупность функций и процедур, принадлежащих ядру или надстройкам операционной системы.

Итак, API — это набор функций, предоставляемых системой программирования разработчику прикладной программы и ориентированных на организацию взаимодействия результирующей прикладной программы с целевой вычислительной системой. Целевая вычислительная система представляет собой совокупность программных и аппаратных средств, в окружении которых выполняется результирующая программа. Сама результирующая программа порождается системой программирования на основании кода исходной программы, созданного разработчиком, а также объектных модулей и библиотек, входящих в состав системы программирования.

В принципе API используется не только прикладными, но и системными программами как в составе операционной системы, так и в составе системы программирования. Но дальше речь пойдет только о функциях API с точки рения разработчика

[1] RTL включает в себя те стандартные подпрограммы, которые система программирования подставляет на этапе компиляции. В общем случае это не только модули системы программирования, но и модули самой операционной системы.

прикладной программы. Для системной программы существуют некоторые дополнительные ограничения на возможные реализации API.

Функции API позволяют разработчику строить результирующую прикладную программу так, чтобы использовать средства целевой вычислительной системы для выполнения типовых операций. При этом разработчик программы избавлен от необходимости создавать исходный код для выполнения этих операций.

Программный интерфейс API включает в себя не только сами функции, но и соглашения об их использовании, которые регламентируются операционной системой, архитектурой целевой вычислительной системы и системой программирования.

Существует несколько вариантов реализации API:

- реализация на уровне модулей операционной системы;
- реализация на уровне системы программирования;
- реализация на уровне внешней библиотеки процедур и функций.

Система программирования в каждом из этих вариантов предоставляет разработчику средства для подключения функций API к исходному коду программы и организации их вызовов. Объектный код функций API подключается к результирующей программе компоновщиком при необходимости.

Возможности API можно оценивать со следующих позиций:

- эффективности выполнения функций API — эффективность включает в себя скорость выполнения функций и объем вычислительных ресурсов, необходимых для их выполнения;
- широты предоставляемых возможностей;
- зависимости прикладной программы от архитектуры целевой вычислительной системы.

В идеале хотелось бы иметь набор функций API, выполняющихся с наивысшей эффективностью, предоставляющих пользователю все возможности современных операционных систем и имеющих минимальную зависимость от архитектуры вычислительной системы (еще лучше — лишенных такой зависимости).

Добиться наивысшей эффективности выполнения функций API практически трудно по тем же причинам, по которым невозможно добиться наивысшей эффективности выполнения для любой результирующей программы. Поэтому об эффективности интерфейса API можно говорить только в сравнении его характеристик с другими интерфейсами API.

Что касается двух других показателей, то в принципе нет никаких технических ограничений на их реализацию. Однако существуют организационные проблемы и узкие корпоративные интересы, тормозящие создание такого рода библиотек.

Реализация функций API на уровне модулей операционной системы

При реализации функций API на уровне модулей операционной системы операционная система ответственна за выполнение функций API. Объектный код, вы-

полняющий функции, либо непосредственно входит в состав операционной системы (или даже ядра операционной системы), либо находится в составе динамически загружаемых библиотек, поставляемых вместе с системой. Система программирования ответственна только за то, чтобы организовать интерфейс для вызова этого кода.

В таком варианте результирующая программа обращается непосредственно к операционной системе. Поэтому достигается наибольшая эффективность выполнения функций API по сравнению со всеми другими вариантами реализации API.

Недостатком организации API по такой схеме является практически полное отсутствие переносимости не только кода результирующей программы, но и кода исходной программы. Программа, созданная для одной архитектуры вычислительной системы, не сможет исполняться на вычислительной системе другой архитектуры даже после того, как ее объектный код полностью перестроен. Чаще всего система программирования просто не сможет выполнить перестроение исходного кода для новой архитектуры вычислительной системы, поскольку многие функции API, ориентированные на определенную операционную систему, в новой архитектуре могут просто отсутствовать.

Таким образом, в данной схеме для переноса прикладной программы с одной целевой вычислительной системы на другую потребуется изменение исходного кода программы.

Переносимости можно было бы добиться, если унифицировать функции API в различных операционных системах. С учетом корпоративных интересов производителей операционных систем и иного системного программного обеспечения такое направление их развития представляется практически невозможным. В лучшем случае при приложении определенных организационных усилий удается добиться стандартизации смыслового (семантического) наполнения основных функций API, но не их программного интерфейса.

Хорошо известным примером API такого рода может служить набор функций, предоставляемых пользователю со стороны операционных систем типа Microsoft Windows — WinAPI (Windows API). Надо сказать, что даже внутри этого корпоративного интерфейса API существует определенная несогласованность, которая несколько ограничивает переносимость программ между различными операционными системами типа Windows. Еще одним примером такого интерфейса API можно считать набор сервисных функций простейших однопрограммных операционных систем типа MS DOS, реализованный в виде набора подпрограмм обслуживания программных прерываний.

Реализация функций API на уровне системы программирования

При реализации функций API на уровне системы программирования эти функции предоставляются пользователю в виде библиотеки функций соответствующего языка программирования. Обычно речь идет о библиотеке времени выполнения (RTL). Система программирования предоставляет пользователю библиотеку

функций и обеспечивает подключение к результирующей программе объектного кода, ответственного за выполнение этих функций.

Очевидно, что эффективность вызова функций API в таком варианте будет несколько ниже, чем при непосредственном обращении к функциям операционной системы. Так происходит, поскольку для выполнения многих функций API библиотека RTL языка программирования должна все равно выполнять обращения к функциям операционной системы. Наличие всех необходимых вызовов и обращений к функциям операционной системы в объектном коде RTL обеспечивает система программирования.

Однако переносимость исходного кода программы в таком варианте оказывается самой высокой, поскольку синтаксис и семантика всех функций строго регламентированы в стандарте соответствующего языка программирования. Они зависят от языка и не зависят от архитектуры целевой вычислительной системы. Поэтому для выполнения прикладной программы на новой архитектуре вычислительной системы достаточно заново построить код результирующей программы с помощью соответствующей системы программирования.

Единообразное выполнение функций языка обеспечивается системой программирования. При ориентации на различные архитектуры целевой вычислительной системы в системе программирования могут потребоваться различные комбинации вызовов функций операционной системы для выполнения одних и тех же функций исходного языка. Это должно быть учтено в коде RTL. В общем случае для каждой архитектуры целевой вычислительной системы потребуется свой код RTL языка программирования. Выбор того или иного объектного кода RTL для подключения к результирующей программе система программирования обеспечивает автоматически.

Например, рассмотрим функции динамического выделения памяти в языках C и Паскаль. В C это функции malloc, realloc и free (функции new и delete в C++), в Паскале — функции new и dispose. Если использовать эти функции в исходном тексте программы, то с точки зрения исходной программы они будут действовать одинаковым образом в зависимости только от семантики исходного кода программы. При этом для разработчика исходной программы не имеет значения, на какую архитектуру ориентирована его программа. Это имеет значение для системы программирования, которая для каждой из этих функций должна подключить к результирующей программе объектный код библиотеки. Этот код будет выполнять обращение к соответствующим функциям операционной системы. Не исключено даже, что для однотипных по смыслу функций в разных языках (например, malloc в C и new в Паскале выполняют схожие по смыслу действия) этот код будет выполнять схожие обращения к операционной системе. Однако для различных вариантов операционной системы этот код будет различен даже при использовании одного и того же исходного языка.

Проблема главным образом заключается в том, что большинство языков программирования предоставляют пользователю не очень широкий набор стандартизованных функций. Поэтому разработчик исходного кода существенно ограничен в выборе доступных функций API. Как правило, набора стандартных функций ока-

зывается недостаточно для создания полноценной прикладной программы. Тогда разработчик может обратиться к функциям других библиотек, имеющихся в составе системы программирования. В этом случае нет гарантии, что функции, включенные в состав данной системы программирования, но не входящие в стандарт языка программирования, будут доступны в другой системе программирования, особенно если та ориентирована на другую архитектуру целевой вычислительной системы. Такая ситуация уже ближе к третьему варианту реализации API.

Например, те же функции malloc, realloc и free в языке C фактически не входят в стандарт языка. Они входят в состав стандартной библиотеки, которая «де-факто» входит во все системы программирования, построенные на основе языка C. Общепринятые стандарты существуют для многих часто используемых функций языка. Если же взять более специфические функции, такие как функции порождения новых процессов, то для них ни в C, ни в Паскале не окажется общепринятого стандарта.

Реализация функций API с помощью внешних библиотек

При реализации функций API с помощью внешних библиотек эти функции предоставляются пользователю в виде библиотеки процедур и функций, созданной сторонним разработчиком.

Система программирования ответственна только за то, чтобы подключить объектный код библиотеки к результирующей программе. Причем внешняя библиотека может быть и динамически загружаемой (загружаемой во время выполнения программы).

С точки зрения эффективности выполнения этот метод реализации API имеет самые низкие результаты, поскольку внешняя библиотека обращается как к функциям операционной системы, так и к функциям RTL языка программирования. Только при очень высоком качестве внешней библиотеки ее эффективность сравнима с эффективностью библиотеки RTL.

Если говорить о переносимости исходного кода, то здесь требование только одно — используемая внешняя библиотека должна быть доступна в любой из архитектур вычислительных систем, на которые ориентирована прикладная программа. Тогда удается достигнуть переносимости. Это возможно, если внешняя библиотека удовлетворяет какому-то принятому стандарту, а система программирования поддерживает этот стандарт.

Например, библиотеки, удовлетворяющие стандарту POSIX (см. следующий раздел), доступны в большинстве систем программирования для языка C. И если прикладная программа использует только библиотеки этого стандарта, то ее исходный код будет переносимым. Еще одним примером является широко известная библиотека графического интерфейса XLib, поддерживающая стандарт графической среды X-Window.

Для большинства специфических библиотек отдельных разработчиков это не так. Если пользователь использует какую-то библиотеку, то она ориентирована на ог-

раниченный набор доступных архитектур целевой вычислительной системы. Примерами могут служить библиотеки MFC (Microsoft Foundation Classes) от Microsoft и VCL (Visual Controls Library) от Borland, жестко ориентированные на архитектуру операционных систем семейства Windows.

Тем не менее многие фирмы-разработчики сейчас стремятся создавать библиотеки, которые бы обеспечивали переносимость исходного кода приложений между различными архитектурами целевой вычислительной системы. Пока еще такие библиотеки не получили широкого распространения, но имеется несколько попыток их реализации, например библиотека CLX от Borland ориентирована на архитектуру операционных систем семейств Linux и Windows.

В целом развитие функций API идет в направлении попытки создать библиотеки API, обеспечивающие широкую переносимость исходного кода. Однако с учетом корпоративных интересов различных производителей и сложившейся ситуации на рынке в ближайшее время вряд ли удастся достичь значительных успехов в этом направлении. Разработка широко применимого стандарта API пока еще остается делом будущего.

Поэтому разработчики системных программ вынуждены оставаться в довольно узких рамках ограничений стандартных библиотек языков программирования.

Что касается прикладных программ, то гораздо большую перспективу для них предоставляют технологии, связанные с разработками в рамках архитектуры клиент-сервер или трехзвенной архитектуры создания приложений. В этом направлении ведущие производители операционных систем, СУБД и систем программирования скорее достигнут соглашений, чем в направлении стандартизации API.

Итак, нами были рассмотрены основные принципы, цели и подходы к реализации системных интерфейсов API. Отметим еще один очень важный момент: желательно, чтобы интерфейс прикладного программирования не зависел от системы программирования. Конечно, были одно время персональные компьютеры, у которых базовой системой программирования выступал интерпретатор с языка Basic, но это скорее исключение. Обычно интерфейс API не зависит от системы программирования и может вызываться из любой системы программирования, хотя и с использованием соответствующих правил построения вызывающих последовательностей. В то же время, в ряде случаев система программирования может сама генерировать обращения к API. Например, мы можем написать в программе вызов функции по запросу 256 байт памяти:

```
unsigned char * ptr = malloc (256);
```

Система программирования с языка C сгенерирует целую последовательность обращений. Из кода пользовательской программы будет осуществлен вызов библиотечной функции **malloc**, код которой расположен в RTL языка C. Библиотека времени выполнения, в данном случае, реализует вызов **malloc** уже как вызов системной функции **HeapAlloc** API:

```
LPVOID HeapAlloc(
    HANDLE  hHeap,  // handle to the private heap block - указатель на блок
    DWORD  dwFlags, // heap allocation control flags - свойства блока
```

```
    DWORD  dwBytes  // number of bytes to allocate - размер блока
);
```

Параметры выделяемого блока памяти в таком случае задаются системой программирования, и пользователь лишен возможности задавать их напрямую. С другой стороны, если это необходимо, функции API можно вызывать прямо в тексте программы:

```
unsigned char * ptr = (LPVOID) HeapAlloc( GetProcessHeap(), 0, 256);
```

В этом случае программирование вызова немного усложняется, но получаемый конечный результат будет, как правило, короче и, что самое важное, работать будет эффективнее. Следует отметить, что далеко не все возможности API доступны через обращения к функциям системы программирования. Непосредственное обращение к API позволяет пользователю обращаться к системным ресурсам более эффективным способом. Однако это требует знания функций API, количество которых нередко достигает нескольких сотен.

Как правило, функции API не стандартизированы. В каждом конкретном случае набор вызовов API определяется, прежде всего, архитектурой операционной системы и ее назначением. В то же время, принимаются попытки стандартизировать некоторый базовый набор функций, поскольку это существенно облегчило бы перенос приложений с одной операционной системы на другую. Таким примером может служить очень известный и, пожалуй, один из самых распространенных стандарт POSIX. В этом стандарте перечислен большой набор функций, их параметров и возвращаемых значений. Стандартизированными, согласно POSIX, являются не только обращения к API, но и файловая система, организация доступа к внешним устройствам, набор системных команд[1]. Использование в приложениях этого стандарта позволяет в дальнейшем легко переносить такие программы с одной операционной системы в другую путем простейшей перекомпиляции исходного текста.

Частным случаем попытки стандартизации API является внутренний корпоративный стандарт компании Microsoft, известный как WinAPI. Он включает в себя следующие реализации: Win16, Win32s, Win32, WinCE. С точки зрения WinAPI (в силу ряда идеологических причин графический, то есть «оконный», интерфейс пользователя обязателен) базовой задачей является окно. Таким образом, стандарт WinAPI изначально ориентирован на работу в графической среде. Однако базовые понятия дополнены традиционными функциями, в том числе частично поддерживается стандарт POSIX.

Интерфейс POSIX

POSIX (Portable Operating System Interface for Computer Environments — независимый от платформы системный интерфейс для компьютерного окружения) — это стандарт IEEE (Institute of Electrical and Electronics Engineers — институт инженеров по электротехнике и радиоэлектронике), описывающий системные интер-

[1] В данном контексте под системными командами следует понимать некий набор программ, позволяющих управлять вычислительными процессами, например pstat, kill, dir и др.

фейсы для открытых операционных систем, в том числе оболочки, утилиты и инструментарии. Помимо этого, согласно POSIX, стандартизированными являются задачи обеспечения безопасности, задачи реального времени, процессы администрирования, сетевые функции и обработка транзакций. Стандарт базируется на UNIX-системах, но допускает реализацию и в других операционных системах.

Интерфейс POSIX начинался как попытка пропаганды институтом IEEE идей переносимости приложений в UNIX-средах путем разработки абстрактного независимого от платформы стандарта. Однако POSIX не ограничивается только UNIX-системами; существуют различные реализации этого стандарта в системах, которые соответствуют требованиям, предъявляемым стандартом IEEE Standard 1003.1-1990 (POSIX.1). Например, известная ОС реального времени QNX соответствует спецификациям этого стандарта, что облегчает перенос приложений в эту систему, но UNIX-системой не является ни в каком виде, ибо ее архитектура использует абсолютно иные принципы.

Этот стандарт подробно описывает систему виртуальной памяти (Virtual Memory System, VMS), многозадачность (MultiProcess Executing, MPE) и технологию переноса операционных систем (CTOS). Таким образом, на самом деле POSIX представляет собой множество стандартов POSIX.1–POSIX.12. В табл. 9.1 перечислены основные направления, описываемые данными стандартами. Следует также особо отметить, что в POSIX.1 основным языком описания системных функций API предполагается язык C.

Таблица 9.1. Семейство стандартов POSIX

Стандарт	Стандарт ISO	Краткое описание
POSIX.0	Нет	Введение в стандарт открытых систем. Данный документ не является стандартом в чистом виде, а представляет собой рекомендации и краткий обзор технологий
POSIX.1	Да	Системный интерфейс API (язык C)
POSIX.2	Нет	Оболочки и утилиты (одобренные IEEE)
POSIX.3	Нет	Тестирование и верификация
POSIX.4	Нет	Задачи реального времени и потоки выполнения
POSIX.5	Да	Использование языка ADA применительно к стандарту POSIX.1
POSIX.6	Нет	Системная безопасность
POSIX.7	Нет	Администрирование системы
POSIX.8	Нет	Сети, «прозрачный» доступ к файлам, абстрактные сетевые интерфейсы, не зависящие от физических протоколов, вызовы RPC, связь системы с приложениями, зависящими от протокола
POSIX.9	Да	Использование языка Fortran, применительно к стандарту POSIX.1
POSIX.10	Нет	Super-computing Application Environment Profile (AEP)
POSIX.11	Нет	Обработка транзакций AEP
POSIX.12	Нет	Графический интерфейс пользователя (GUI)

Таким образом, программы, написанные с соблюдением данных стандартов, будут одинаково выполняться на всех POSIX-совместимых системах. Однако стандарты отчасти носят всего лишь рекомендательный характер. Часть стандартов описана очень строго, тогда как другая часть только поверхностно раскрывает основные требования. Нередко программные системы заявляются как POSIX-совместимые, хотя таковыми их назвать нельзя. Причины кроются в формальном подходе к реализации интерфейса POSIX в различных операционных системах. На рис. 9.1 изображена типовая схема реализации строго соответствующего POSIX приложения.

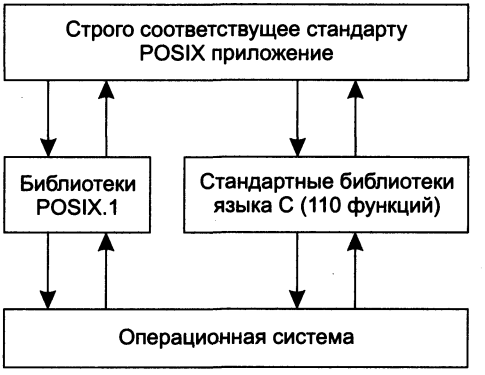


Рис. 9.1. Схема реализации приложения, строго соответствующего стандарту POSIX

Из рисунка видно, что для взаимодействия с операционной системой программа использует только библиотеки POSIX.1 и стандартную библиотеку RTL языка C, в которой возможно использование только 110 различных функций, также описанных стандартом POSIX.1.

К сожалению, достаточно часто с целью увеличения производительности той или иной подсистемы либо для введения фирменных технологий, которые ограничивают область применения приложения соответствующей операционной средой, при программировании используются другие функции, не отвечающие стандарту POSIX.

Реализации стандарта POSIX на уровне операционной системы различны. Если UNIX-системы в своем абсолютном большинстве изначально соответствуют спецификациям IEEE Standard 1003.1-1990, то WinAPI не является POSIX-совместимым. Однако для его поддержки в операционной системе Windows NT введен специальный модуль API для поддержки стандарта POSIX, работающий на уровне привилегий пользовательских процессов. Данный модуль обеспечивает преобразование и передачу вызовов из пользовательской программы к ядру системы и обратно, работая с ядром через WinAPI. Прочие приложения, написанные с использованием WinAPI, могут передавать информацию POSIX приложениям через стандартные механизмы потоков ввода-вывода stdin и stdout [57].

Примеры программирования для разных интерфейсов API

Для наглядной демонстрации принципиальных различий интерфейсов API наиболее популярных современных операционных систем для персональных компьютеров рассмотрим простейший пример, в котором необходимо подсчитать количество пробелов в текстовых файлах, имена которых должны указываться в командной строке. Рассмотрим два варианта программы: для Windows (с использованием WinAPI) и для Linux (POSIX API).

Поскольку нас интересует работа с параллельными задачами, пусть при выполнении программы для каждого из перечисленных в командной строке файлов создается свой процесс или поток выполнения (задача), который параллельно с другими процессами (потоками) производит работу по подсчету пробелов в «своем» файле. Результатом работы программы будет являться список файлов с подсчитанным количеством пробелов для каждого.

Следует обратить особое внимание на то, что приведенные ниже реализации программ решения данной задачи не являются единственно возможными. В обеих рассматриваемых операционных системах существуют разные методы работы с файловой системой и управления процессами. В данном случае рассматривается только один, но наиболее характерный для соответствующего интерфейса API вариант.

Для того чтобы было удобнее сравнивать эту (листинг 9.1) и следующую (листинг 9.2) программы, а также учитывая, что задача не требует для своего решения оконного интерфейса, в тексте использованы только те вызовы API, которые не затрагивают графический интерфейс. Конечно, нынче редко какое приложение не использует возможностей GUI, но зато в нашем случае сразу можно увидеть разницу в организации параллельной работы запускаемых вычислений.

Листинг 9.1. Текст программы для Windows (WinAPI)

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

// Название: processFile
// Описание: исполняемый код потока
// Входные параметры: lpFileName - имя файла для обработки
// Выходные параметры: нет
DWORD processFile(LPVOID lpFileName ) {
    HANDLE handle; // описатель файла
    DWORD numRead, total = 0;
    char buf;

    // запрос к ОС на открытие файла (только для чтения)
    handle = CreateFile( (LPCTSTR)lpFileName, GENERIC_READ,
      FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

    // цикл чтения до конца файла
    do {
```

продолжение ⇩

Листинг 9.1 (*продолжение*)

```
            // чтение одного символа из файла
            ReadFile( handle, (LPVOID) &buf, 1, &numRead, NULL);
            if (buf == 0x20) total++;
    } while ( numRead > 0);

    fprintf( stderr, "(ThreadID: %Lu), File %s, spaces = %d\n",
      GetCurrentThreadId(), lpFileName, total);

    // закрытие файла
    CloseHandle( handle);

    return(0);
}

// Название: main
// Описание: главная программа
// Входные параметры: список имен файлов для обработки
// Выходные параметры: нет
int main(int argc, char *argv[]) {
int i;
DWORD pid;
HANDLE hThrd[255];    // массив ссылок на потоки

    // для всех файлов, перечисленных в командной строке
    for (i = 0; i< (argc-1); i++) {
        // запуск потока - обработка одного файла
        hThrd[i] = CreateThread( NULL, 0x4000,
          (LPTHREAD_START_ROUTINE) processFile,
          (LPVOID) argv[i+1], 0, &pid);
        fprintf( stdout, "processFile started (HND=%d)\n", hThrd[i]);
    }

    // ожидание окончания выполнения всех запущенных потоков
    WaitForMultipleObjects( argc-1, hThrd, true, INFINITE);

    return(0);
}
```

Обратите внимание, что основная программа запускает потоки и ждет окончания их выполнения. Другими словами, мы имеем всего один вычислительный процесс, но используем мультизадачные возможности операционной системы.

Листинг 9.2. Текст программы для Linux (POSIX API)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <wait.h>
#include <fcntl.h>
#include <stdio.h>

// Название: processFile
// Описание: обработка файла, подсчет кол-ва пробелов
// Входные параметры: fileName - имя файла для обработки
// Выходные параметры: кол-во пробелов в файле
int processFile( char *fileName) {
    int handle, numRead, total = 0;
```

```
    char buf;

    // запрос к ОС на открытие файла (только для чтения)
    handle = open( fileName, O_RDONLY);

    // цикл чтения до конца файла
    do {
        // чтение одного символа из файла
        numRead = read( handle, &buf, 1);
        if (buf == 0x20) total++;
    } while (numRead > 0);

       // закрытие файла
    close( handle);
    return( total);
}

// Название: main
// Описание: главная программа
// Входные параметры: список имен файлов для обработки
// Выходные параметры: нет
int main(int argc, char *argv[]) {
int i, pid, status;

    // для всех файлов, перечисленных в командной строке
    for (i = 1; i< argc; i++) {
        // запускаем дочерний процесс
        pid = fork();
        if (pid == 0) {
            // если выполняется дочерний процесс
            // вызов функции счета количества пробелов в файле
            printf( "(PID: %d), File %s, spaces = %d\n",
              getpid(), argv[ i], processFile( argv[ i]));
            // выход из процесса
            exit();
        }
        // если выполняется родительский процесс
        else
        printf( "processFile started (pid=%d)\n", pid);
    }

    // ожидание окончания выполнения всех запущенных процессов
    if (pid != 0) while (wait(&status)>0);
    return;
}
```

Из листинга 9.2 видно, что здесь все вычисления имеют статус процессов, а не потоков выполнения. Надо заметить, что многие современные версии UNIX поддерживают механизм потоков, поскольку потоки в ряде случаев позволяют повысить эффективность вычислений и упрощают их создание, но в рассматриваемом интерфейсе потоков нет.

В заключение можно заметить, что очень трудно сравнивать интерфейсы API. При их разработке создатели, как правило, стараются реализовать функционально полный набор основных функций, используя которые можно решать разные задачи, правда, порой, различными способами. Один набор системных функций хорош для

одного набора задач, другой — для иного набора задач. Тем более что, фактически, сейчас мы имеем существенно ограниченное множество интерфейсов API из-за того, что имеет место доминирование наиболее популярных операционных систем и на их распространении в большей степени сказалась правильная маркетинговая политика их создателей, а не достоинства и недостатки самих этих систем и их интерфейсов.

Контрольные вопросы и задачи

1. Что вы понимаете под архитектурой операционной системы?
2. Перечислите и поясните основные принципы построения операционных систем.
3. Для чего операционные системы используют несколько режимов работы процессора? Чем отличается супервизорный режим работы процессора от пользовательского? Как часто процессор переводится в супервизорный режим?
4. Объясните принцип виртуализации. Имеется ли связь между принципом виртуализации и принципом совместимости? Если имеется, то поясните, в чем она заключается?
5. Что такое ядро операционной системы? Расскажите об основных моментах, характерных для микроядерных ОС. Какие основные функции должно выполнять микроядро ОС?
6. Перечислите основные требования, предъявляемые к операционным системам в плане обеспечения информационной безопасности.
7. Перечислите основные требования, предъявляемые к операционным системам реального времени.
8. Какие задачи возлагаются на интерфейс прикладного программирования (API)?
9. Какими могут быть варианты реализации API? В чем заключаются достоинства и недостатки каждого варианта?
10. Что такое библиотека времени выполнения (RTL)?
11. Что такое POSIX? Какими преимуществами обладают программы, созданные с использованием только стандартных функций, предусмотренных POSIX?

Глава 10. Краткий обзор современных операционных систем

Теперь, после знакомства с основными понятиями, относящимися к операционным системам, и изучения конкретных механизмов, реализующих известные методы организации вычислительных процессов, вкратце рассмотрим архитектурные особенности современных операционных систем для персональных компьютеров типа IBM PC.

Прежде всего, отметим тот общеизвестный факт, что наиболее популярными являются операционные системы семейства Windows компании Microsoft. Это и Windows 95/98/ME, и Windows NT/2000, и новое поколение Windows XP/2003 — этим операционным системам посвящена отдельная глава (см. главу 11). Здесь же мы рассмотрим операционные системы, не относящиеся к продуктам Microsoft, — это UNIX-подобные операционные системы Linux и FreeBSD, а также системы QNX и OS/2. При изучении известных всему миру систем с общим названием Linux и системы FreeBSD, по которым сейчас появляется немало монографий и учебников, упор будет сделан именно на основных архитектурных особенностях семейства UNIX, в абсолютном своем большинстве относящихся ко всем UNIX-системам. Система QNX была выбрана потому, что является наиболее известной и удачной операционной системой реального времени. Операционную систему OS/2 мы рассмотрим последней. Хотя сейчас эта система уже практически всеми забыта[1], она была одной из первых полноценных и надежных мультипрограммных и мультизадачных операционных систем для персональных компьютеров, в которой поддерживалось несколько операционных сред.

---

[1] В настоящее время ее используют те организации, которые в свое время создали под нее свои приложения, вложив немалые средства. И поскольку система по-прежнему в основном неплохо выполняет свои функции, эти организации не спешат вкладывать деньги для переноса своих задач на новые платформы.

Семейство операционных систем UNIX

UNIX является исключительно удачным примером реализации простой мультипрограммной и многопользовательской операционной системы. В свое время она проектировалась как инструментальная система для разработки программного обеспечения. Своей уникальностью система UNIX обязана во многом тому обстоятельству, что была, по сути, создана всего двумя разработчиками[1], которые делали ее исключительно для себя и первое время использовали на мини-ЭВМ с очень скромными вычислительными ресурсами. Первая версия этой системы занимала всего около 12 Кбайт и могла работать на компьютерах с очень небольшим объемом оперативной памяти. Поскольку при создании второй версии UNIX разработчики отказались от языка ассемблера и специально придумали язык высокого уровня, на котором можно было бы писать не только системные, но и прикладные программы (речь идет о языке C), то и сама система UNIX, и приложения, выполняющиеся в ней, стали легко переносимыми (мобильными). Компилятор с языка C для всех оттранслированных программ дает реентерабельный и разделяемый код, что позволяет эффективно использовать имеющиеся в системе ресурсы.

Общая характеристика и особенности архитектуры

Первой целью при разработке этой системы было стремление сохранить простоту и обойтись минимальным количеством функций. Все реальные сложности оставлялись пользовательским программам.

Второй целью была общность. Одни и те же методы и механизмы должны были использоваться во многих случаях:

- обращение к файлам, устройствам ввода-вывода и буферам межпроцессных сообщений выполняются с помощью одних и тех же примитивов;
- одни и те же механизмы именования, присвоения альтернативных имен и защиты от несанкционированного доступа применяются и к файлам с данными, и к каталогам, и к устройствам;
- одни и те же механизмы работают в отношении программно и аппаратно инициируемых прерываний.

Третья цель заключалась в том, чтобы сложные задачи можно было решать, комбинируя существующие небольшие программы, а не разрабатывая их заново.

Наконец, четвертая цель состояла в создании мультитерминальной операционной системы с эффективными механизмами разделения не только процессорного времени, но и всех остальных ресурсов. В мультитерминальной операционной системе на одно из первых мест по значимости выходят вопросы защиты одних вычис-

[1] Создателями системы UNIX считаются Кен Томпсон и Деннис Ритчи. В своей операционной системе Томпсон и Ритчи учли опыт работы над проектом сложной мультизадачной операционной системы с разделением времени, которая имела название MULTICS (MULTiplexed Information and Computing System). Название новой системы UNIX произошло от аббревиатуры UNICS (Uniplexed Information and Computing System).

лительных процессов от вмешательства других вычислительных процессов. Причем для реализации третьей цели необходимо было создать механизмы полноценного обмена данными между программными модулями, из которых предполагалось составлять конечные программы.

Операционная система UNIX обладает простым, но очень мощным командным языком и независимой от устройств файловой системой. Важным, хотя и простым с позиций реализации такой возможности, является тот факт, что система UNIX предоставляет пользователям средства направления выхода одной программы непосредственно на вход другой. В результате достигается четвертая цель — большие программные системы можно создавать путем композиции имеющихся небольших программ, а не путем написания новых, что в большинстве случаев упрощает задачу. UNIX-системы существуют уже 30 лет, и к настоящему времени имеется чрезвычайно большой набор легко переносимых из системы в систему отлично отлаженных и проверенных временем приложений.

В число системных и прикладных программ, поставляемых с UNIX-системами, входят редакторы текстов, программируемые интерпретаторы командного языка, компиляторы с нескольких популярных языков программирования, включая C, C++, ассемблер, PERL, FORTRAN и многие другие, компоновщики (редакторы межпрограммных связей), отладчики, многочисленные библиотеки системных и пользовательских программ, средства сортировки и ведения баз данных, многочисленные административные и обслуживающие программы. Для абсолютного большинства всех этих программ имеется документация, в том числе исходные тексты программ (как правило, хорошо комментированные). Кроме того, описания и документация по большей части доступны пользователям в интерактивном режиме. Используется иерархическая файловая система с полной защитой, работа со съемными томами, обеспечивается независимость от устройств.

Центральной частью UNIX-систем является ядро (kernel). Оно состоит из большого количества модулей и с точки зрения архитектуры считается монолитным. Однако в ядре всегда можно выделить три основные подсистемы: управления процессами, управления файлами, управления операциями ввода-вывода между центральной частью и периферийными устройствами. Подсистема управления процессами организует выполнение и диспетчеризацию процессов, их синхронизацию и разнообразное межпроцессное взаимодействие. Важнейшая функция подсистемы управления процессами — это распределение оперативной памяти и (для современных систем) организация виртуальной памяти. Подсистема управления файлами тесно связана и с подсистемой управления процессами, и с драйверами. Ядро может быть перекомпилировано с учетом конкретного состава устройств компьютера и решаемых задач. Не все драйверы могут быть включены в состав ядра, часть из них может вызываться из ядра. Более того, очень большое количество системных функций выполняется системными программными модулями, не входящими непосредственно в ядро, но вызываемых из ядра. Основные системные функции, которые должно выполнять ядро совместно с остальными системными модулями, строго стандартизированы. За счет этого во многом достигается переносимость кода между разными версиями UNIX и абсолютно различным аппаратным обеспечением.

Основные понятия

Одним из достоинств ОС UNIX является то, что система базируется на небольшом числе понятий; рассмотрим их вкратце. Здесь необходимо отметить, что настоящая книга не претендует на полноценное изложение основ работы и детальное описание архитектуры системы UNIX (или Linux). На эту тему имеется достаточное количество специальной литературы, например отличная монография [39] или такие замечательные книги, как [23, 43]. Тем не менее, исходя из имеющегося опыта преподавания предметов, относящихся к операционным системам и системному программному обеспечению, считаю полезным изложить здесь минимальный набор основных понятий, который часто помогает студентам «погрузиться в мир UNIX», отличающийся от привычного всем окружения Windows.

Виртуальная машина

Система UNIX многопользовательская. Каждому пользователю после регистрации (входа в систему) предоставляется виртуальный компьютер, в котором есть все необходимые ресурсы: процессор (процессорное время выделяется на основе круговой, или карусельной, диспетчеризации и с использованием динамических приоритетов, что позволяет обеспечить равенство в обслуживании), оперативная память, устройства, файлы. Текущее состояние такого виртуального компьютера, предоставляемого пользователю, называется *образом*. Можно сказать, что процесс — это выполнение образа. Образ процесса состоит:

- из образа памяти;
- значений общих регистров процессора;
- состояния открытых файлов;
- текущего каталога файлов;
- другой информации.

Образ процесса во время выполнения процесса размещается в основной памяти. В старых версиях UNIX образ можно было «сбросить» на диск, если какому-либо более приоритетному процессу требовалось место в основной памяти. Напомним, что такое замещение процессов называется свопингом (swapping). В современных реализациях, поддерживающих, как правило, страничный механизм виртуальной памяти, прежде всего выгружаются неиспользуемые страницы, а не целиком образ. В частности, в системах Linux свопинг образов не применяется, но создается специальный[1] раздел на магнитном диске для файла подкачки (swap-file), где размещаются виртуальные страницы выполняющихся процессов, для которых не хватает места в оперативной памяти. Таким образом, замещаются не процессы, а их отдельные страницы.

Образ памяти делится на три логических сегмента:

- сегмент реентерабельных процедур (начинается с нулевого адреса в виртуальном адресном пространстве процесса);

[1] Сигнатура этого раздела обозначается как 082h.

- сегмент данных (располагается следом за сегментом процедур и может расти в сторону больших адресов);
- сегмент стека (начинается со старшего адреса и растет в сторону младших адресов по мере занесения в него информации при вызовах подпрограмм и при прерываниях).

В современных версиях UNIX-систем все виртуальное адресное пространство каждого образа отображается на реальную физическую память компьютера. Используется страничный механизм организации виртуальной памяти. И следует различать замещение процессов и подкачку страниц, хотя в обоих случаях используется термин swapping.

Пользователь

Мы уже отмечали, что с самого начала операционная система UNIX замышлялась как интерактивная многопользовательская система. Другими словами, UNIX предназначена для мультитерминальной работы. Чтобы начать работать, пользователь должен «войти» в систему, введя со свободного терминала свое учетное, или входное, имя (account name, или login) и пароль (password). Человек, зарегистрированный в учетных файлах системы и, следовательно, имеющий учетное имя, называется зарегистрированным пользователем системы. Регистрацию новых пользователей обычно выполняет администратор системы. Пользователь не может изменить свое учетное имя, но может установить и/или изменить свой пароль. Пароли хранятся в отдельном файле в закодированном виде.

Ядро операционной системы UNIX идентифицирует каждого пользователя по его идентификатору (User Identifier, UID), уникальному целому значению, присваиваемому пользователю при регистрации в системе. Кроме того, каждый пользователь относится к некоторой группе пользователей, которая также идентифицируется некоторым целым значением (Group Identifier, GID). Значения UID и GID для каждого зарегистрированного пользователя сохраняются в учетных файлах системы и приписываются процессу, в котором выполняется командный интерпретатор, запущенный при входе пользователя в систему. Эти значения наследуются каждым новым процессом, запущенным от имени данного пользователя, и используются ядром системы для контроля правомочности доступа к файлам, выполнения программ и т. д.

Все пользователи операционной системы UNIX явно или неявно работают с файлами. Файловая система операционной системы UNIX имеет древовидную структуру [39]. Промежуточными узлами дерева являются каталоги со ссылками на другие каталоги или файлы, а листья дерева соответствуют файлам или пустым каталогам. Каждому зарегистрированному пользователю соответствует некоторый каталог файловой системы, который называется *домашним* (home) каталогом пользователя. При входе в систему пользователь получает неограниченный доступ к своему домашнему каталогу и всем каталогам и файлам, содержащимся в нем. Пользователь может создавать, удалять и модифицировать каталоги и файлы, содержащиеся в домашнем каталоге. Потенциально возможен доступ и ко всем другим файлам, однако он может быть ограничен, если пользователь не имеет достаточных привилегий.

Суперпользователь

Очевидно, что администратор системы, который тоже является зарегистрированным пользователем, чтобы управлять всей системой, должен обладать существенно большими, чем обычные пользователи, привилегиями. В операционных системах UNIX эта задача решается путем выделения единственного нулевого значения UID. Пользователь с таким значением UID называется *суперпользователем* (superuser) и обозначается словом *root* (корень). Он имеет неограниченные права на доступ к любому файлу и на выполнение любой программы. Кроме того, такой пользователь имеет возможность полного контроля над системой. Он может остановить ее и даже разрушить. По этой причине не рекомендуется работать под этой учетной записью. Администратор должен создать себе обычную учетную запись простого пользователя, а для выполнения действий, связанных с административными полномочиями, рекомендуется использовать команду su. Команда su запрашивает у пользователя пароль суперпользователя, и, если он указан правильно, операционная система переводит сеанс пользователя в режим работы суперпользователя. После выполнения необходимых действий, требующих привилегий суперпользователя, следует выполнить команду exit, которая и вернет администратору статус простого пользователя.

Еще одним важным отличием суперпользователя от обычного пользователя операционной системы UNIX является то, что на суперпользователя не распространяются ограничения на используемые ресурсы. Для обычных пользователей устанавливаются такие ограничения, как максимальный размер файла, максимальное число сегментов разделяемой памяти, максимально допустимое пространство на диске и т. д. Суперпользователь может изменять эти ограничения для других пользователей, но на него они не действуют.

Интерфейс пользователя

Традиционный способ взаимодействия пользователя с системой UNIX основывается на командных языках. После входа пользователя в систему для него запускается один из командных интерпретаторов (в зависимости от параметров, сохраняемых в файле /etc/passwd). Обычно в системе поддерживается несколько командных интерпретаторов с похожими, но различающимися своими возможностями командными языками. Общее название для любого командного интерпретатора ОС UNIX — *оболочка* (shell), поскольку любой интерпретатор представляет внешнее окружение ядра системы. По умолчанию в системах Linux командным интерпретатором является bash. В принципе он может быть заменен другим, но практически никто этого не делает.

Вызванный командный интерпретатор выдает приглашение на ввод пользователем командной строки, которая может содержать простую команду, конвейер команд или последовательность команд. После выполнения очередной командной строки и выдачи на экран терминала или в файл соответствующих результатов интерпретатор команд снова выдает приглашение на ввод командной строки, и так до тех пор, пока пользователь не завершит свой сеанс работы и не выйдет из системы.

Командные языки, используемые в UNIX, достаточно просты, чтобы новые пользователи могли быстро начать работать, и достаточно мощны, чтобы можно было использовать их для написания сложных программ. Последняя возможность опирается на механизм *командных файлов* (shell scripts), которые могут содержать произвольные последовательности командных строк. При указании имени командного файла вместо очередной команды интерпретатор читает файл строка за строкой и последовательно интерпретирует команды.

Поскольку в настоящее время все большее распространение получают графические интерфейсы, в операционных системах семейства UNIX стали все чаще работать в X-Window. X-Window — это графический интерфейс, позволяющий пользователям взаимодействовать со своими вычислениями и с системой в графическом режиме. В отличие от систем Windows компании Microsoft, графический интерфейс для UNIX-систем не является основным, в системе можно работать и без него. Прежде всего, графический режим разрабатывался для приложений, предназначенных для работы с графикой. Однако в последние годы его стали применять гораздо чаще, особенно в системах Linux, которые начинают использовать не только как серверные операционные системы, но и как системы для персональных компьютеров.

Графический интерфейс в UNIX-системах основан на модели клиент-сервер. Серверная часть X-Window — это аппаратно-зависимая система ввода-вывода, которая непосредственно взаимодействует с приложением и видеоподсистемой, клавиатурой и мышью. При этом серверная часть должна работать на компьютере, производящем вычисления. Взаимодействие с пользователем осуществляется через клиентскую часть, которая обеспечивает вывод данных на дисплей и прием их с устройств ввода. Клиентская часть должна быть на том компьютере, за которым работает пользователь. Таким образом, можно работать в графическом режиме, сидя за одним компьютером, в то время как собственно вычисления могут происходить и на другом компьютере.

Один из клиентов X-Window — это оконный менеджер (также называемый диспетчером окон). Он управляет размещением окон на экране, определяет их вид и характер управляющих элементов. То есть именно он и предоставляет пользователю графический интерфейс (GUI), тогда как X-Window — это его основа.

В системах Linux наиболее популярными менеджерами графического интерфейса являются KDE и GNOME. Для запуска X-Window в системах семейства UNIX (и Linux) используется команда `startx`.

Команды и командный интерпретатор

Как уже упоминалось, оболочкой (shell) в UNIX-системе называют механизм взаимодействия между пользователями и системой. По сути дела, это интерпретатор команд, который считывает набираемые пользователем строки и запускает указанные в командах программы, которые и выполняют запрошенные системные функции и операции. Полный командный язык, интерпретируемый оболочкой, богат возможностями и достаточно сложен, однако большинство команд просты в использовании, и запомнить их не составляет труда.

Командная строка состоит из имени команды (а именно имени выполняемого файла), за которым следует список аргументов, разделенных пробелами. Оболочка разбивает командную строку на компоненты. Указанный в команде файл загружается, и ему обеспечивается доступ к заданным в команде аргументам.

Любой командный язык оболочки фактически состоит из трех частей:

- служебных конструкций, позволяющих манипулировать текстовыми строками и строить сложные команды на основе простых команд;
- встроенных команд, выполняемых непосредственно интерпретатором командного языка;
- команд, представляемых отдельными выполняемыми файлами.

В свою очередь, набор команд последнего вида включает стандартные команды (системные утилиты, такие как vi, cc и т. д.) и команды, созданные пользователями системы. Для того чтобы выполняемый файл, разработанный пользователем ОС UNIX, можно было запускать как команду оболочки, достаточно определить в одном из исходных файлов функцию с именем main (имя main должно быть глобальным, то есть перед ним не должно указываться ключевое слово static). Если употребить в качестве имени команды имя такого выполняемого файла, командный интерпретатор создаст новый процесс и запустит в нем указанную выполняемую программу, начиная с вызова функции main.

Тело функции main, вообще говоря, может быть произвольным (для интерпретатора существенно только наличие входной точки в программу с именем main), но для того чтобы создать команду, которой можно задавать параметры, придерживаются некоторых стандартных правил. В этом случае каждая функция main должна определяться с двумя параметрами — argc и argv. После вызова команды параметру argc будет соответствовать число символьных строк, указанных в качестве аргументов вызова команды, а argv — массив указателей на переменные, содержащие эти строки. При этом имя самой команды составляет первую строку аргументов (то есть после вызова значение argc всегда больше или равно 1). Код функции main должен проанализировать допустимость заданного значения argc и соответствующим образом обработать заданные текстовые строки.

Например, следующий текст на языке C может быть использован для создания команды, которая выводит на экран текстовую строку, заданную в качестве ее аргумента:

```
#include <stdio.h>
main (argc, argv)
int argc;
char *argv[];
{
  if (argc != 2)
{ printf("usage: %s your-text\n", argv[0]);
  exit;
}
  printf("%s\n", argv[1]);
}
```

Процессы

Процесс в системах UNIX — это процесс в классическом понимании этого термина, то есть это программа, выполняемая в собственном виртуальном адресном пространстве. Когда пользователь входит в систему, автоматически создается процесс, в котором выполняется программа командного интерпретатора. Если командному интерпретатору встречается команда, соответствующая выполняемому файлу, то он создает новый процесс и запускает в нем соответствующую программу, начиная с функции main. Эта запущенная программа, в свою очередь, может создать процесс и запустить в нем другую программу (та тоже должна содержать функцию main) и т. д.

Для образования нового процесса и запуска в нем программы используются два системных вызова API — fork() и exec(имя_выполняемого_файла). Системный вызов fork() приводит к созданию нового адресного пространства, состояние которого абсолютно идентично состоянию адресного пространства основного процесса (то есть в нем содержатся те же программы и данные). Для дочернего процесса заводятся копии всех сегментов данных.

Другими словами, сразу после выполнения системного вызова fork() основной (родительский) и порожденный процессы являются абсолютными близнецами; управление в том и другом находится в точке, непосредственно следующей за вызовом fork(). Чтобы программа могла разобраться, в каком процессе (основном или порожденном) она теперь работает, функция fork() возвращает разные значения: 0 в порожденном процессе и целое положительное число в основном процессе. Этим целым положительным числом является уже упоминавшийся идентификатор процесса (PID). Таким образом, родительский процесс будет знать идентификатор своего дочернего процесса и может при необходимости управлять им.

Теперь, если мы хотим запустить новую программу в порожденном процессе, нужно обратиться к системному вызову exec, указав в качестве аргументов вызова имя файла, содержащего новую выполняемую программу, и, возможно, одну или несколько текстовых строк, которые будут переданы в качестве аргументов функции main новой программы. Выполнение системного вызова exec приводит к тому, что в адресное пространство порожденного процесса загружается новая выполняемая программа и запускается с адреса, соответствующего входу в функцию main. Другими словами, это приводит к замене текущего программного сегмента и текущего сегмента данных, которые были унаследованы при выполнении вызова fork, соответствующими сегментами, заданными в файле. Прежние сегменты теряются. Это эффективный метод смены выполняемой процессом программы, но не самого процесса. Файлы, уже открытые до вызова примитива exec, остаются открытыми после его выполнения.

В следующем примере пользовательская программа, вызываемая как команда оболочки, выполняет в отдельном процессе стандартную команду ls оболочки, которая выдает на экран содержимое текущего каталога.

```
main()
{if(fork()==0) wait(0);       /* родительский процесс */
  else execl("ls", "ls", 0); /* порожденный процесс */
}
```

Таким образом, с практической точки зрения процесс в UNIX является объектом, создаваемым в результате выполнения функции fork(). Каждый процесс за исключением начального (нулевого) порождается в результате вызова другим процессом функции fork(). Каждый процесс имеет одного родителя, но может породить много процессов. Начальный (нулевой) процесс является особенным процессом, который создается в результате загрузки системы. После порождения нового процесса с идентификатором 1 нулевой процесс становится процессом подкачки и реализует механизм виртуальной памяти. Процесс с идентификатором 1, известный под именем init, является предком любого другого процесса в системе и связан с каждым процессом особым образом.

Функционирование

Теперь, когда мы познакомились с основными понятиями, рассмотрим наиболее характерные моменты функционирования UNIX-системы.

Выполнение процессов

Процесс может выполняться в одном из двух состояний, а именно *пользовательском* и *системном*. В пользовательском состоянии процесс выполняет пользовательскую программу и имеет доступ к пользовательскому сегменту данных. В системном состоянии процесс выполняет программы ядра и имеет доступ к системному сегменту данных.

Когда пользовательскому процессу требуется выполнить системную функцию, он делает *системный вызов*. Фактически происходит вызов ядра системы как подпрограммы. С момента системного вызова процесс считается системным. Таким образом, пользовательский и системный процессы являются двумя фазами одного и того же процесса, но они никогда не пересекаются между собой. Каждая фаза пользуется своим собственным стеком. Стек задачи содержит аргументы, локальные переменные и другую информацию относительно функций, выполняемых в режиме задачи. Диспетчерский процесс не имеет пользовательской фазы.

В UNIX-системах организуется *разделение времени* (time sharing), то есть каждому процессу выделяется квант времени. Либо процесс завершается сам до истечения отведенного ему кванта времени, либо он приостанавливается по истечении кванта и продолжает свое исполнение при очередном получении нового кванта времени. Механизм диспетчеризации характеризуется достаточно справедливым распределением процессорного времени между всеми процессами. Пользовательским процессам приписываются приоритеты в зависимости от получаемого ими процессорного времени. Процессам, которые получили много процессорного времени, назначают более низкие приоритеты, в то время как процессам, которые получили лишь немного процессорного времени, наоборот, повышают приоритет. Вспомните рассмотренные ранее механизмы динамических приоритетов. Такой метод диспетчеризации обеспечивает хорошее время реакции для всех пользователей системы. Все системные процессы имеют более высокие приоритеты по сравнению с пользовательскими и поэтому всегда обслуживаются в первую очередь.

Подсистема ввода-вывода

Функции ввода-вывода в UNIX задаются в основном с помощью пяти системных вызовов: **open**, **close**, **read**, **write** и **seek**.

Открыть файл можно следующей командой:

```
file_descriptor = open (file_name, mode)
```

Здесь **mode** — режим открытия файла (чтение, запись или то и другое); **file_descriptor** — дескриптор файла, служит для последующих ссылок на данный файл; **file_name** — имя открываемого файла.

Чтение и запись осуществляются командами следующего вида:

```
after_reading_bytes = read (file_descriptor, buffer, bytes)
after_writing_bytes = write (file_descriptor, buffer, bytes)
```

Здесь **bytes** — количество байтов, которые должны быть прочитаны или записаны; **after_reading_bytes** и **after_writing_bytes** — реально прочитанное и записанное количество байтов соответственно.

При чтении возможны три ситуации, в каждой из которых чтение происходит последовательно:

- если это первое чтение из файла, то оно осуществляется последовательно с самого начала файла;
- если операции чтения предшествовала другая операция чтения из этого файла, то текущая операция предоставит данные, непосредственно следующие за предыдущими;
- если предшествовала операция поиска **seek** (см. далее), то чтение осуществляется последовательно от точки смещения, указанной в операции **seek**.

Это же справедливо и по отношению к операции записи в файл. Обратите внимание, что все эти вызовы относятся к последовательному доступу и эффект прямой адресации достигается с помощью команды **seek**, смещающей текущую позицию файла:

```
Seek (file_descriptor,displacement,displacement_type)
```

Здесь параметр **displacement_type** (тип смещения) определяет, является смещение абсолютным или относительным, а также задано оно числом байтов или числом блоков по 512 байт.

Важно заметить, что команда **seek** исполняется для магнитных дисков так же, как и для магнитных лент, которые нынче уже практически не используются, но во времена появления и становления UNIX-систем были часто используемым устройством.

Чтобы закрыть файл, достаточно выполнить следующую команду:

```
close (file_descriptor)
```

Еще три примитива — **gtty**, **stty**, **stat** — позволяют получать и задавать информацию о файлах и терминалах.

Те же самые команды ввода-вывода применяются и к физическим устройствам. В UNIX-системах физические устройства представлены специальными файлами в единой структуре файловой системы. Это означает, что пользователь не может

написать зависящую от устройств программу, если только эта зависимость не отражена в самом потоке передаваемых данных. Стандартные файлы ввода и вывода, приписываемые пользовательскому терминалу, открывать обычным путем не требуется. Терминал открывается автоматически по команде входа в систему login.

Система ввода-вывода UNIX в отличие от большинства других систем ориентирована на работу скорее с потоком данных, а не с записями. Здесь *поток данных* (stream)[1] — это последовательность байтов, заканчивающаяся разделителем (то есть символом конца потока). Понятие потока данных позволяет проще добиться независимости от устройств и унификации файлов с физическими устройствами и конвейерами. Тем самым пользователь получает гибкость в работе с группами данных, но на него ложатся и дополнительные заботы, поскольку ему приходится писать программы управления данными. Пользователь может при необходимости относительно легко самостоятельно реализовать работу с записями. Чтобы работать с записями фиксированной длины, достаточно просто задавать постоянную длину во всех командах чтения и записи. Для нахождения позиции нужной записи при фиксированной длине записей нужно умножить длину записи на номер записи и выполнить команду seek. Работу с записями переменной длины можно организовать, если разместить в начале каждой записи поле фиксированного размера, содержащее значение длины записи.

Перенаправление ввода-вывода

Механизм перенаправления ввода-вывода является одним из наиболее элегантных, мощных и одновременно простых механизмов UNIX. Цель, которая ставилась при разработке этого механизма, состоит в следующем. Поскольку UNIX — это интерактивная система, которая создавалась в конце 60-х — начале 70-х годов, то обычно программы считывали текстовые строки с алфавитно-цифрового терминала и выводили результирующие текстовые строки на экран терминала. Для того чтобы обеспечить большую гибкость при использовании таких программ, желательно было иметь возможность вводить в них данные непосредственно из файлов или с выхода других программ и выводить их данные в файл или на вход других программ.

Реализация этого механизма основывается на следующих свойствах операционных систем семейства UNIX. Во-первых, любой ввод-вывод трактуется как ввод из некоторого файла и вывод в некоторый файл. Клавиатура и экран терминала тоже интерпретируются как файлы (первый можно только читать, а во второй можно только писать). Во-вторых, доступ к любому файлу производится через его дескриптор (положительное целое число). Фиксируются три значения дескрипторов файлов. Файл с дескриптором 1 называется файлом стандартного ввода (stdin), файл с дескриптором 2 — файлом стандартного вывода (stdout), и файл с дескриптором 3 — файлом стандартного вывода диагностических сообщений (stderr). В-третьих, программа, запущенная в некотором процессе, «наследует» от породившего процесса все дескрипторы открытых файлов.

[1] Не путать с потоком выполнения, или тредом (thread).

В головном процессе интерпретатора командного языка файлом стандартного ввода является клавиатура терминала пользователя, а файлами стандартного вывода и вывода диагностических сообщений — экран терминала. Однако при запуске любой команды можно сообщить интерпретатору (средствами соответствующего командного языка), какой файл или выход какой программы должен служить файлом стандартного ввода для запускаемой программы, а также какой файл или вход какой программы должен служить для запускаемой программы файлом стандартного вывода или файлом вывода диагностических сообщений. Тогда интерпретатор перед выполнением системного вызова exec открывает указанные файлы, подменяя смысл дескрипторов 1, 2 и 3.

То же самое может проделать и любая другая программа, запускающая третью программу в специально созданном процессе. Следовательно, все, что требуется для нормального функционирования механизма перенаправления ввода-вывода, — это придерживаться при программировании соглашения об использовании дескрипторов stdin, stdout и stderr. Это не очень трудно, поскольку в наиболее распространенных функциях библиотеки ввода-вывода printf, scanf и error вообще не требуется указывать дескриптор файла. Функция printf неявно использует дескриптор stdout, функция scanf — дескриптор stdin, функция error — дескриптор stderr.

Файловая система

Файл в системе UNIX представляет собой множество символов с произвольным доступом. В файле могут содержаться любые данные, помещенные туда пользователем, и файл не имеет никакой иной структуры, кроме той, какую создаст в нем пользователь.

Структура файловой системы

Здесь мы вкратце рассмотрим одну из первых реализаций файловой системы, поскольку основные ее идеи сохраняются до сих пор.

Информация на дисках размещается блоками. В первой версии файловой системы размер блока был равен 512 байт. Во многих современных файловых системах, разработанных для конкретной версии UNIX-клона, размер блока больше. Это позволяет повысить быстродействие файловых операций. Например, в системе FFS (Fast File System — быстродействующая файловая система) размер блока равен 8192 байт.

В рассматриваемой версии файловой системы раздел диска разбивается на следующие области (рис. 10.1):

- неиспользуемый блок;
- управляющий блок, или суперблок, в котором хранится размер логического диска и границы других областей;
- *i*-список, состоящий из описаний файлов, называемых *l*-узлами;
- область для хранения содержимого файлов.

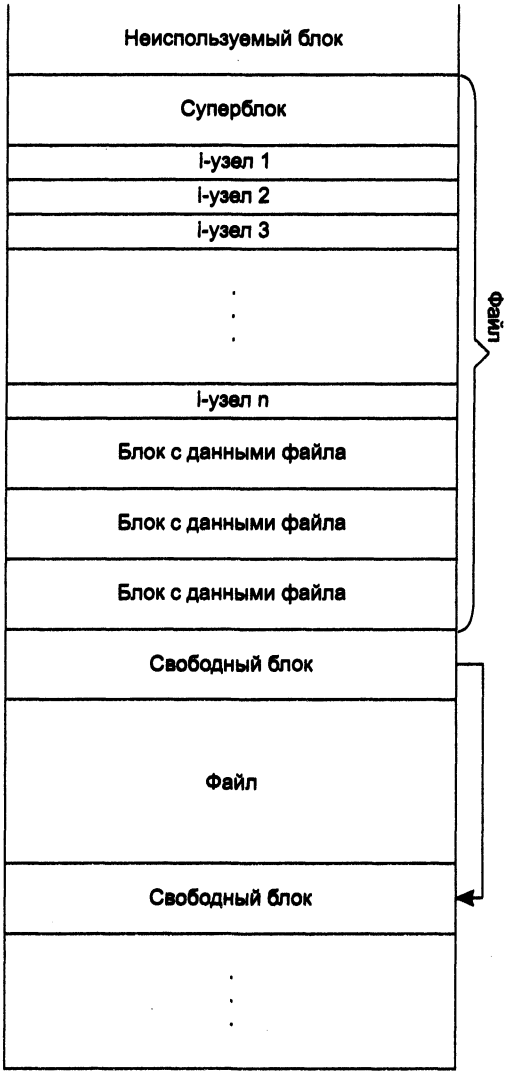


Рис. 10.1. Организация файловой системы в ОС UNIX

Каждый *i*-узел содержит:

- идентификатор владельца;
- идентификатор группы владельца;
- биты защиты;
- физические адреса на диске или ленте, где находится содержимое файла;
- размер файла;
- время создания файла;
- время последнего изменения (modification time) файла;

- время последнего изменения атрибутов (change time) файла;
- число связей-ссылок, указывающих на файл;
- индикатор типа файла (каталог, обычный файл или специальный файл).

Следом за *i*-списком идут блоки, предназначенные для хранения содержимого файлов. Пространство на диске, оставшееся свободным от файлов, образует связанный список свободных блоков.

Таким образом, файловая система UNIX представляет собой структуру данных, размещенную на диске и содержащую управляющий суперблок с описанием файловой системы в целом, массив *i*-узлов, в котором определены все файлы в файловой системе, сами файлы и, наконец, совокупность свободных блоков. Выделение пространства под данные осуществляется блоками фиксированного размера.

Каждый файл однозначно идентифицируется *старшим номером устройства*, *младшим номером устройства* и *i-номером* (индексом *i*-узла данного файла в массиве *i*-узлов). Когда вызывается драйвер устройства, по старшему номеру индексируется массив входных точек в драйверы. По младшему номеру драйвер выбирает одно устройство из группы идентичных физических устройств.

Файл-каталог, в котором перечислены имена файлов, позволяет установить соответствие между именами и самими файлами. Каталоги образуют древовидную структуру. На каждый обычный файл или файл устройства могут иметься ссылки в различных узлах этой структуры. В непривилегированных программах запись в каталог не разрешена, но при наличии соответствующих разрешений они могут читать их. Дополнительных связей между каталогами нет.

Большое число системных каталогов UNIX использует для собственных нужд. Один из них, корневой каталог, является базой для всей структуры каталогов, и, «отталкиваясь» от него, можно найти все файлы. В других системных каталогах содержатся программы и команды, предоставляемые пользователям, а также файлы устройств.

Имена файлов задаются последовательностью имен каталогов, разделенных косой чертой (/) и приводящих к концевому узлу (листу) некоторого дерева. Если имя файла начинается с косой черты, то поиск по дереву начинается в корневом каталоге. Если же имя файла не имеет в начале косой черты, то поиск начинается с текущего каталога. Имена файлов, начинающиеся с символов ../ (две точки и косая черта), подразумевают начало поиска в каталоге, родительском по отношению к текущему. Имя файла stuff (персонал) указывает на элемент stuff в текущем каталоге. Имя файла /work/alex/stuff приводит к поиску каталога work в корневом каталоге, затем к поиску каталога alex в каталоге work и, наконец, к поиску элемента stuff в каталоге alex. Сама по себе косая черта (/) обозначает корневой каталог. В приведенном примере нашла отражение типичная иерархическая структура файловой системы, например work может обозначать диск (устанавливаемый при работе пользователя), alex может быть каталогом пользователя, а stuff может принадлежать alex.

Файл, не являющийся каталогом, может встречаться в различных каталогах, возможно, под разными именами. Это называется *связыванием*. Элемент в каталоге,

относящийся к одному файлу, называется связью. В UNIX-системах все такие связи имеют равный статус. Файлы не принадлежат каталогам. Скорее, файлы существуют независимо от элементов каталогов, а связи в каталогах указывают на реальные (физические) файлы. Файл «исчезает», когда удаляется последняя связь, указывающая на него. Биты защиты, заданные в связях, могут отличаться от битов в исходном файле. Таким образом решается проблема избирательного ограничения на доступ к файлам.

С каждым поддерживаемым системой устройством ассоциируется один или большее число специальных файлов. Операции ввода-вывода для специальных файлов осуществляются так же, как и для обычных дисковых файлов с той лишь разницей, что эти операции активизируют соответствующие устройства. Специальные файлы обычно находятся в каталоге `/dev`. На специальные файлы могут указывать связи точно так же, как на обычные файлы.

От файловой системы не требуется, чтобы она целиком размещалась на том устройстве, где находится корень. Запрос от системы `mount` (на установку носителей и т. п.) позволяет встраивать в иерархию файлов файлы на сменных томах. Команда `mount` имеет несколько аргументов, но обязательных аргументов у стандартного варианта ее использования два: имя файла блочного устройства и имя каталога. В результате выполнения этой команды файловая подсистема, расположенная на указанном устройстве, подключается к системе таким образом, что ее содержимое заменяет собой содержимое заданного в команде каталога. Поэтому для *монтирования* соответствующего тома обычно используют пустой каталог. Команда `umount` выполняет обратную операцию — «отсоединяет» файловую систему, после чего диск с данными можно физически извлечь из системы. Например, для записи данных на дискету необходимо ее «подмонтировать», а после работы — «размонтировать».

Монтирование файловых систем позволяет получить единое логическое файловое пространство, в то время как реально отдельные каталоги с файлами могут находиться в разных разделах одного жесткого диска и даже на разных жестких дисках. Причем, что очень важно, сами файловые системы для монтируемых разделов могут быть разными. Например, при работе в системе Linux мы можем иметь часть разделов с файловой системой EXT2FS, а часть разделов — с файловой системой EXT3FS. Важно, чтобы ядро знало эти файловые системы.

Защита файлов

Защита файлов осуществляется при помощи номера, идентифицирующего пользователя, и десяти битов защиты — атрибутов доступа. Права доступа подразделяются на три типа: чтение (read), запись (write) и выполнение (execute). Эти типы прав доступа могут быть предоставлены трем классам пользователей: владельцу файла, группе, в которую входит владелец, и всем прочим пользователям. Девять из этих битов управляют защитой по чтению, записи и исполнению для владельца файла, других членов группы, в которую входит владелец, и всех других пользователей. Файл всегда связан с определенным пользователем — своим владельцем и с определенной группой, то есть у него есть уже известные нам идентификаторы

пользователя (UID) и группы (GID). Изменять права доступа к файлу разрешено только его владельцу. Изменить владельца файла может только суперпользователь, изменить группу — суперпользователь или владелец файла.

Программа, выполняющаяся в системе, всегда запускается от имени определенных пользователя и группы (обычно основной группы этого пользователя), но связь процессов с пользователями и группами организована сложнее. Различают идентификаторы доступа к файловой системе для пользователя (File System access User ID, FSUID) и для группы (File System access Group ID, FSGID), а также эффективные идентификаторы пользователя (Effective User ID, EUID) и группы (Effective Group ID, EGID). Кроме того, при доступе к файлам учитываются полномочия (capabilities), присвоенные самому процессу.

При создании файл получает идентификатор UID, совпадающий с FSUID процесса, который его создает, а также идентификатор GID, совпадающий с FSGID этого процесса.

Атрибуты доступа определяют, что разрешено делать с данным файлом данной категории пользователей. Имеется всего три операции: чтение, запись и выполнение.

При создании файла (или при создании еще одного имени для уже существующего файла) модифицируется не сам файл, а каталог, в котором появляются новые ссылки на узлы. Удаление файла заключается в удалении ссылки. Таким образом, право на создание или удаление файла — это право на запись в каталог.

Право на выполнение каталога интерпретируется как право на поиск в нем (прохождение через него). Оно позволяет обратиться к файлу с помощью пути, содержащему данный каталог, даже тогда, когда каталог не разрешено читать, и поэтому список всех его файлов недоступен.

Помимо трех названных основных атрибутов доступа существуют дополнительные, используемые в следующих случаях. Атрибуты SUID и SGID важны при запуске программы: они требуют, чтобы программа выполнялась не от имени запустившего ее пользователя (группы), а от имени владельца (группы) того файла, в котором она находится. Если файл программы имеет атрибут SUID (SGID), то идентификаторы FSUID и EUID (FSGID и EGID) соответствующего процесса не наследуются от процесса, запустившего его, а совпадают с UID (GID) файла. Благодаря этому пользователи получают возможность запустить системную программу, которая создает свои рабочие файлы в закрытых для них каталогах.

Кроме того, если процесс создает файл в каталоге, имеющем атрибут SGID, то файл получает GID не по идентификатору FSGID процесса, а по идентификатору GID каталога. Это удобно для коллективной работы: все файлы и вложенные каталоги[1] в каталоге автоматически оказываются принадлежащими одной и той же группе, хотя создавать их могут разные пользователи. Есть еще один атрибут SVTX, который нынче относится к каталогам. Он показывает, что из каталога, имеющего этот атрибут, ссылку на файл может удалить только владелец файла. Существуют две стандартные формы записи прав доступа — символьная и восьмеричная. Символь-

[1] Вложенные каталоги часто называют подкаталогами (subdirectory).

ная запись представляет собой цепочку из десяти знаков, первый из которых не относится собственно к правам, а обозначает тип файла. Используются следующие обозначения:

- - (дефис) — обычный файл;
- d — каталог;
- c — символьное устройство;
- b — блочное устройство;
- p — именованный канал (named pipe);
- s — сокет (socket)[1];
- l — символическая ссылка.

Далее следуют три последовательности, каждая из трех символов, соответствующие правам пользователя, группы и всех остальных. Наличие права на чтение обозначается символом r, на запись — символом w, на выполнение — символом x, отсутствие какого-либо права — символом - (дефис) в соответствующей позиции.

Наличие атрибута SUID (SGID) обозначается прописной буквой S в позиции права на выполнение для владельца (группы), если выполнение не разрешено, и строчной буквой s, если разрешено.

Восьмеричная запись — это шестизначное число, первые два знака которого обозначают тип файла и довольно часто опускаются, третья цифра — атрибуты GUID (4), SGID (2) и SVTX (1), оставшиеся три — права владельца, группы и всех остальных соответственно. Очевидно, что право на чтение можно представить числом 4, право на запись — числом 2, а право на выполнение — числом 1.

Например, стандартный набор прав доступа для каталога /tmp в символьной форме выглядит как drwxrwxrwx, а в восьмеричной — как 041777 (каталог; чтение, запись и поиск разрешены всем; установлен атрибут SVTX). А набор прав -r-S-xw-, или в числовом виде — 102412, означает, что это обычный файл, владельцу разрешается читать его, но не выполнять и не изменять, пользователям из группы (за исключением владельца) — выполнять (причем во время работы программа получит права владельца файла), но не читать и не изменять, а всем остальным — изменять, но не читать и не выполнять.

Большинство программ создают файлы с разрешением на чтение и запись для всех пользователей, а каталоги — с разрешением на чтение, запись и поиск для всех пользователей. Этот исходный набор атрибутов логически складывается с *пользовательской маской создания файла* (user file-creation mask, umask), которая обычно ограничивает доступ. Например, значения u=rwx, g=rwx, o=r-x для пользовательской маски следует понимать так: у владельца и группы остается полный набор прав, а всем остальным запрещается запись. В восьмеричном виде оно запишется как 002 (первая цифра — ограничения для владельца, вторая — для группы, третья — для

[1] Сокет — это понятие, связанное со стеком протоколов TCP/IP, который является «родным» для UNIX. Его следует понимать как некий адрес или порт, через который связываются удаленные программы.

остальных; запрещение чтения — 4, записи — 2, выполнения — 1). Владелец файла может изменить права доступа к нему командой chmod.

Взаимодействие между процессами

Операционная система UNIX в полной мере отвечает требованиям технологии клиент-сервер. Эта универсальная модель служит основой построения любых сколь угодно сложных систем, в том числе и сетевых. Разработчики СУБД, коммуникационных систем, систем электронной почты, банковских систем и т. д. во всем мире широко используют технологию клиент-сервер. Для построения программных систем, работающих по принципам модели «клиент-сервер», в UNIX существуют следующие механизмы:

- сигналы;
- семафоры;
- программные каналы;
- очереди сообщений;
- сегменты разделяемой памяти;
- вызовы удаленных процедур.

Многие из этих механизмов нам уже знакомы, поэтому рассмотрим их вкратце. Для более глубокого изучения этих вопросов можно рекомендовать известную работу [43].

Сигналы

Если рассматривать выполнение процесса на виртуальном компьютере, который предоставляется каждому пользователю, то в такой системе должна существовать система прерываний, отвечающая стандартным требованиям:

- обработка исключительных ситуаций;
- средства обработки внешних и внутренних прерываний;
- средства управления системой прерываний (маскирование и демаскирование).

Всем этим требованиям в UNIX отвечает механизм сигналов, который позволяет не только воспринимать и обрабатывать сигналы, но и порождать их и посылать на другие машины (процессы). Сигналы могут быть синхронными, когда инициатор сигнала — сам процесс, и асинхронными, когда инициатор сигнала — интерактивный пользователь, сидящий за терминалом. Источником асинхронных сигналов может быть также ядро, когда оно контролирует определенные состояния аппаратуры, рассматриваемые как ошибочные.

Сигналы можно рассматривать как простейшую форму взаимодействия между процессами. Они используются для передачи от одного процесса другому или от ядра ОС какому-либо процессу уведомления о возникновении определенного события.

Семафоры

Механизм семафоров, реализованный в UNIX-системах, является обобщением классического механизма семафоров, предложенного известным голландским спе-

циалистом профессором Дейкстрой. Семафор в операционной системе семейства UNIX состоит из следующих элементов:

- значения семафора;
- идентификатора процесса, который хронологически последним работал с семафором;
- числа процессов, ожидающих увеличения значения семафора;
- числа процессов, ожидающих нулевого значения семафора.

Для работы с семафорами имеются следующие три системных вызова:

- semget — создание и получение доступа к набору семафоров;
- semop — манипулирование значениями семафоров (именно этот системный вызов позволяет с помощью семафоров организовать синхронизацию процессов);
- semctl — выполнение разнообразных управляющих операций над набором семафоров.

Системный вызов semget имеет следующий синтаксис:

```
id = semget(key, count, flag);
```

Здесь параметры key и flag определяют ключ объекта и дополнительные флаги. Параметр count задает число семафоров в наборе семафоров, обладающих одним и тем же ключом. После этого индивидуальный семафор идентифицируется дескриптором набора семафоров и номером семафора в этом наборе. Если к моменту выполнения системного вызова semget набор семафоров с указанным ключом уже существует, то обращающийся процесс получит соответствующий дескриптор, но так и не узнает о реальном числе семафоров в группе (хотя позже это все-таки можно узнать с помощью системного вызова semctl).

Основным системным вызовом для манипулирования семафором является semop:

```
oldval = semop(id, oplist, count);
```

Здесь id — это ранее полученный дескриптор группы семафоров, oplist — массив описателей операций над семафорами группы, а count — размер этого массива. Значение, возвращаемое системным вызовом, является значением последнего обработанного семафора. Каждый элемент массива oplist имеет следующую структуру:

- номер семафора в указанном наборе семафоров;
- операция;
- флаги.

Если проверка прав доступа проходит нормально и указанные в массиве oplist номера семафоров не выходят за пределы общего размера набора семафоров, то системный вызов выполняется следующим образом. Для каждого элемента массива oplist значение соответствующего семафора изменяется в соответствии со значением поля операции, как показано ниже.

- Если значение поля операции положительно, то значение семафора увеличивается на единицу, а все процессы, ожидающие увеличения значения семафора, активизируются (*пробуждаются* — в терминологии UNIX).

- Если значение поля операции равно нулю и значение семафора также равно нулю, выбирается следующий элемент массива oplist. Если же значение поля операции равно нулю, а значение семафора отлично от нуля, то ядро увеличивает на единицу число процессов, ожидающих нулевого значения семафора, причем обратившийся процесс переводится в состояние ожидания (*засыпает* — в терминологии UNIX).
- Если значение поля операции отрицательно и его абсолютное значение меньше или равно значению семафора, то ядро прибавляет это отрицательное значение к значению семафора. Если в результате значение семафора стало нулевым, то ядро активизирует (пробуждает) все процессы, ожидающие нулевого значения этого семафора. Если же значение семафора оказывается меньше абсолютной величины поля операции, то ядро увеличивает на единицу число процессов, ожидающих увеличения значения семафора, и откладывает (*усыпляет*) текущий процесс до наступления этого события.

Интересно заметить, что основным поводом для введения массовых операций над семафорами было стремление дать программистам возможность избегать тупиковых ситуаций, возникающих в связи с семафорной синхронизацией. Это обеспечивается тем, что системный вызов semop, каким бы длинным он ни был (по причине потенциально неограниченной длины массива oplist), выполняется как атомарная операция, то есть во время выполнения semop ни один другой процесс не может изменить значение какого-либо семафора.

Наконец, среди флагов-параметров системного вызова semop может содержаться флаг с символическим именем IPC_NOWAIT, наличие которого заставляет ядро UNIX не блокировать текущий процесс, а лишь сообщать в ответных параметрах о возникновении ситуации, приведшей к блокированию процесса в случае отсутствия флага IPC_NOWAIT. Мы не будем обсуждать здесь возможности корректного завершения работы с семафорами при незапланированном завершении процесса; заметим только, что такие возможности обеспечиваются.

Системный вызов semctl имеет следующий формат:

```
semctl(id, number, cmd, arg);
```

Здесь id — это дескриптор группы семафоров, number — номер семафора в группе, cmd — код операции, arg — указатель на структуру, содержимое которой интерпретируется по-разному в зависимости от операции. В частности, с помощью вызова semctl можно уничтожить индивидуальный семафор в указанной группе. Однако детали этого системного вызова настолько громоздки, что лучше рекомендовать в случае необходимости обращаться к технической документации используемого варианта операционной системы.

Программные каналы

Мы уже познакомились с программными каналами в главе 7. Рассмотрим этот механизм еще раз, так сказать, в его исходном, изначальном толковании.

Программные *каналы* (pipes) в системе UNIX являются очень важным средством взаимодействия и синхронизации процессов. Теоретически программный канал

позволяет взаимодействовать любому числу процессов, обеспечивая дисциплину *FIFO* (First In First Out — первый пришедший первым и выбывает). Другими словами, процесс, читающий из программного канала, прочитает те данные, которые были записаны в программный канал раньше других. В традиционной реализации программных каналов для хранения данных использовались файлы. В современных версиях операционных систем семейства UNIX для реализации программных каналов применяются другие средства взаимодействия между процессами (в частности, очереди сообщений).

В UNIX различаются два вида программных каналов — именованные и неименованные. Именованный программный канал может служить для общения и синхронизации произвольных процессов, знающих имя данного программного канала и имеющих соответствующие права доступа. Неименованным программным каналом могут пользоваться только породивший его процесс и его потомки (необязательно прямые).

Для создания именованного программного канала (или получения к нему доступа) используется обычный файловый системный вызов open. Для создания же неименованного программного канала существует специальный системный вызов pipe (исторически более ранний). Однако после получения соответствующих дескрипторов оба вида программных каналов используются единообразно с помощью стандартных файловых системных вызовов read, write и close.

Системный вызов pipe имеет следующий синтаксис:

```
pipe(fdptr);
```

Здесь fdptr — это указатель на массив из двух целых чисел, в который после создания неименованного программного канала будут помещены дескрипторы, предназначенные для чтения из программного канала (с помощью системного вызова read) и записи в программный канал (с помощью системного вызова write). Дескрипторы неименованного программного канала — это обычные дескрипторы файлов, то есть такому программному каналу соответствуют два элемента таблицы открытых файлов процесса. Поэтому при последующих системных вызовах read и write процесс совершенно не обязан отличать случай использования программных каналов от случая использования обычных файлов (собственно, на этом и основана идея перенаправления ввода-вывода и организации конвейеров).

Для создания именованных программных каналов (или получения доступа к уже существующим каналам) используется обычный системный вызов open. Основным отличием от случая открытия обычного файла является то, что если именованный программный канал открывается для записи и ни один процесс не открыл тот же программный канал для чтения, то обращающийся процесс блокируется до тех пор, пока некоторый процесс не откроет данный программный канал для чтения. Аналогично обрабатывается открытие для чтения.

Запись данных в программный канал и чтение данных из программного канала (независимо от того, именованный он или не именованный) выполняются с помощью системных вызовов read и write. Отличие от случая использования обычных файлов состоит лишь в том, что при записи данные помещаются в начало канала, а при чтении выбираются (освобождая соответствующую область памяти) из конца канала.

Окончание работы процесса с программным каналом (независимо от того, именованный он или не именованный) производится с помощью системного вызова close.

Очереди сообщений

Для обмена данными между процессами используется механизм очередей сообщений, который поддерживается следующими системными вызовами:

- msgget — образование новой очереди сообщений или получение дескриптора существующей очереди;
- msgsnd — отправка сообщения (точнее, его постановка в указанную очередь сообщений);
- msgrcv — прием сообщения (точнее, выборка сообщения из очереди сообщений);
- msgctl — выполнение ряда управляющих действий.

Ядро хранит сообщения в виде связного списка (очереди), а дескриптор очереди сообщений является индексом в массиве заголовков очередей сообщений.

Системный вызов msgget имеет следующий синтаксис:

```
msgqid = msgget(key, flag);
```

Здесь параметры key и flag имеют то же значение, что и в вызове semget при запросе семафора.

При выполнении системного вызова msgget ядро UNIX-системы либо создает новую очередь сообщений, помещая ее заголовок в таблицу очередей сообщений и возвращая пользователю дескриптор вновь созданной очереди, либо находит элемент таблицы очередей сообщений, содержащий указанный ключ, и возвращает соответствующий дескриптор очереди.

Для отправки сообщения используется системный вызов msgsnd:

```
msgsnd(msgqid, msg, count, flag);
```

Здесь msg — указатель на структуру, содержащую определяемый пользователем целочисленный тип сообщения и символьный массив (собственно сообщение); count — размер сообщения в байтах; flag — значение, которое определяет действия ядра при выходе за пределы допустимых размеров внутренней буферной памяти.

Для приема сообщения используется системный вызов msgrcv:

```
count = msgrcv(id, msg, maxcount, type, flag);
```

Здесь msg — указатель на структуру данных в адресном пространстве пользователя, предназначенную для размещения принятого сообщения; maxcount — размер области данных (массива байтов) в структуре msg; type — тип сообщения, которое требуется принять; flag — значение, которое указывает ядру, что следует предпринять, если в указанной очереди сообщений отсутствует сообщение с указанным типом. Возвращаемое значение системного вызова задает реальное число байтов, переданных пользователю.

Следующий системный вызов служит для опроса состояния описателя очереди сообщений, изменения его состояния (например, изменения прав доступа к очереди) и для уничтожения указанной очереди сообщений:

```
msgctl(id, cmd, mstatbuf);
```

Разделяемая память

Для работы с разделяемой памятью используются четыре системных вызова:

- shmget — создает новый сегмент разделяемой памяти или находит существующий сегмент с тем же ключом;
- shmat — подключает сегмент с указанным дескриптором к виртуальной памяти обращающегося процесса;
- shmdt — отключает от виртуальной памяти ранее подключенный к ней сегмент с указанным виртуальным адресом начала;
- shmctl — служит для управления разнообразными параметрами, связанными с существующим сегментом.

После того как сегмент разделяемой памяти подключен к виртуальной памяти процесса, процесс может обращаться к соответствующим элементам памяти с использованием обычных машинных команд чтения и записи, не прибегая к дополнительным системным вызовам.

Синтаксис системного вызова shmget выглядит следующим образом:

```
shmid = shmget(key, size, flag);
```

Параметр size определяет желаемый размер сегмента в байтах. Далее работа происходит по общим правилам. Если в таблице разделяемой памяти находится элемент, содержащий заданный ключ, и права доступа не противоречат текущим характеристикам обращающегося процесса, то значением системного вызова является дескриптор существующего сегмента (и обратившийся процесс так и не узнает реального размера сегмента, хотя впоследствии его можно узнать с помощью системного вызова shmctl). В противном случае создается новый сегмент, размер которого не меньше, чем установленный в системе минимальный размер сегмента разделяемой памяти, и не больше, чем установленный максимальный размер. Создание сегмента не означает немедленного выделения для него основной памяти. Это действие откладывается до первого системного вызова подключения сегмента к виртуальной памяти некоторого процесса. Аналогично, при выполнении последнего системного вызова отключения сегмента от виртуальной памяти соответствующая основная память освобождается.

Подключение сегмента к виртуальной памяти выполняется путем обращения к системному вызову shmat:

```
virtaddr = shmat(id, addr, flags);
```

Здесь id — ранее полученный дескриптор сегмента; addr — требуемый процессу виртуальный адрес, который должен соответствовать началу сегмента в виртуальной памяти. Значением системного вызова является реальный виртуальный адрес начала сегмента (его значение не обязательно совпадает со значением параметра addr). Если значением addr является нуль, ядро выбирает подходящий виртуальный адрес начала сегмента.

Для отключения сегмента от виртуальной памяти используется системный вызов shmdt:

```
shmdt(addr);
```

Здесь addr — виртуальный адрес начала сегмента в виртуальной памяти, ранее полученный с помощью системного вызова shmat. При этом система гарантирует (опираясь на данные таблицы сегментов процесса), что указанный виртуальный адрес действительно является адресом начала разделяемого сегмента в виртуальной памяти данного процесса.

Для управления памятью служит системный вызов shmctl:

```
shmctl(id, cmd, shsstatbuf);
```

Параметр cmd идентифицирует требуемое конкретное действие, то есть ту или иную функцию. Наиболее важной является функция уничтожения сегмента разделяемой памяти, которое производится следующим образом. Если к моменту выполнения системного вызова ни один процесс не подключил сегмент к своей виртуальной памяти, то основная память, занимаемая сегментом, освобождается, а соответствующий элемент таблицы разделяемых сегментов объявляется свободным. В противном случае в элементе таблицы сегментов выставляется флаг, запрещающий выполнение системного вызова shmget по отношению к этому сегменту, но процессам, успевшим получить дескриптор сегмента, по-прежнему разрешается подключать сегмент к своей виртуальной памяти. При выполнении последнего системного вызова отключения сегмента от виртуальной памяти операция уничтожения сегмента завершается.

Вызовы удаленных процедур

Во многих случаях взаимодействие процессов соответствует отношениям клиент-сервер. Один из процессов (клиент) запрашивает у другого процесса (сервера) некоторую услугу (сервис) и не продолжает свое выполнение до тех пор, пока эта услуга не будет выполнена (то есть пока процесс-клиент не получит соответствующие результаты). Видно, что семантически такой режим взаимодействия эквивалентен вызову процедуры. Отсюда и соответствующее название — вызов удаленной процедуры (Remote Procedure Call, RPC). Другими словами, процесс обращается к процедуре, которая не принадлежит данному процессу. Она может находиться даже на другом компьютере. Операционная система UNIX по своей «идеологии» идеально подходит для того, чтобы быть сетевой операционной системой, на основе которой можно создавать распределенные системы и организовывать распределенные вычисления. Свойства переносимости позволяют создавать «операционно-однородные» сети, включающие разнородные компьютеры. Однако остается проблема разного представления данных в компьютерах разной архитектуры. Поэтому одной из основных идей RPC является автоматическое обеспечение преобразования форматов данных при взаимодействии процессов, выполняющихся на разнородных компьютерах.

Реализация механизма вызовов удаленных процедур (RPC) достаточно сложна, поскольку этот механизм должен обеспечить работу взаимодействующих процессов, находящихся на разных компьютерах. Если в случае обращения к процедуре, расположенной на том же компьютере, процесс общается с ней через стек или общие области памяти, то в случае удаленного вызова передача параметров процедуре превращается в передачу запроса по сети. Соответственно, и получение результата также осуществляется с помощью сетевых механизмов.

Вызов удаленных процедур включает следующие шаги [39].

1. Процесс-клиент осуществляет вызов локальной процедуры, которую называют *заглушкой* (stub). Задача этого модуля-заглушки — принять аргументы, преобразовать их в стандартную форму и сформировать сетевой запрос. Упаковка аргументов и создание сетевого запроса называется *сборкой* (marshalling).
2. Сетевой запрос пересылается на удаленную систему, где соответствующий модуль ожидает такой запрос и при его получении извлекает параметры вызова процедуры, то есть выполняет *разборку* (unmarshalling), а затем передает их серверу удаленной процедуры. После выполнения осуществляется обратная передача.

Операционная система Linux

Linux — это современная UNIX-подобная операционная система для персональных компьютеров и рабочих станций, удовлетворяющая стандарту POSIX.

Как известно, Linux — это свободно распространяемая версия UNIX-систем, которая первоначально разрабатывалась Линусом Торвальдсом (torvalds@kruuna.helsinki.fi) в университете Хельсинки (Финляндия). Он предложил разрабатывать ее совместно и выдвинул условие, согласно которому исходные коды являются открытыми, любой может их использовать и изменять, но при этом обязан оставить открытым и свой код, внесенный в тот или иной модуль системы. Все компоненты системы, включая исходные тексты, распространяются с лицензией на свободное копирование и установку для неограниченного числа пользователей.

Таким образом, система Linux была создана с помощью многих программистов и энтузиастов UNIX-систем, общающихся между собой через Интернет. К данному проекту добровольно подключились те, кто имеет достаточно навыков и способностей развивать систему. Большинство программ Linux разработаны в рамках проекта GNU из Free Software Foundation (Кембридж, штат Массачусетс). Но в него внесли свою лепту и многие программисты со всего мира.

Изначально система Linux создавалась как «самодельная» UNIX-подобная реализация для машин типа IBM PC с процессором i80386. Однако вскоре Linux стала настолько популярна и ее поддержало такое большое число компаний, что в настоящее время имеются реализации этой операционной системы практически для всех типов процессоров и компьютеров на их основе. На базе Linux создаются и встроенные системы, и суперкомпьютеры. Система поддерживает кластеризацию и большинство современных интерфейсов и технологий.

Большинство свойств Linux присущи другим реализациям UNIX, кроме того, имеются некоторые уникальные свойства. Этот раздел представляет собой лишь краткий обзор этих свойств.

Linux — это полноценная многозадачная многопользовательская операционная система (точно так же, как и все другие версии UNIX). Это означает, что одновременно много пользователей могут работать на одной машине, параллельно выполняя множество программ. Поскольку при работе за персональным компьютером практически никто не подключает к нему дополнительные терминалы (хотя это в

принципе возможно), пользователь просто имитирует работу за несколькими терминалами. В этом смысле можно говорить о *виртуальных терминалах*. По умолчанию пользователь регистрируется на первом терминале. При этом он получает примерно следующее сообщение:

```
Mandrake Linux release 9.0 (dolphin) for i586
Kernel 2.4.16-16mdk on an i686 /tty1
vienna login:
```

Здесь во второй строке слово tty1 означает, что пользователь сейчас взаимодействует с системой через первый виртуальный терминал. Собственно работа на нем возможна только после аутентификации — ввода своих учетного имени и пароля.

При желании открыть второй или последующий сеанс работы на соответствующем терминале, пользователь должен нажать комбинацию клавиш Alt+Fi, где i обозначает номер функциональной клавиши и одновременно номер соответствующего виртуального терминала. Всего Linux поддерживает до семи терминалов, причем седьмой терминал связан с графическим режимом работы и использованием одного из оконных менеджеров. Однако если пользователь работает в графическом режиме, то для перехода в один из алфавитно-цифровых терминалов следует воспользоваться комбинацией клавиш Ctrl+Alt+Fi. В каждом сеансе пользователь может запускать свои задачи.

Система Linux достаточно хорошо совместима с рядом стандартов для UNIX (насколько можно говорить о стандартизации UNIX) на уровне исходных текстов, включая IEEE POSIX.1, System V и BSD. Она и создавалась с расчетом на такую совместимость. Большинство свободно распространяемых через Интернет программ для UNIX может быть откомпилировано для Linux практически без особых изменений[1]. Кроме того, все исходные тексты для Linux, включая ядро, драйверы устройств, библиотеки, пользовательские программы и инструментальные средства распространяются свободно. Другие специфические внутренние черты Linux включают контроль работ по стандарту POSIX (используемый оболочками, такими как csh и bash), псевдотерминалы (pty), поддержку национальных и стандартных раскладок клавиатур динамически загружаемыми драйверами клавиатур.

Linux поддерживает различные типы файловых систем для хранения данных. Некоторые файловые системы, такие как EXT2FS, были созданы специально для Linux. Поддерживаются также другие типы файловых систем, например Minix-1 и Xenix. Кроме того, реализована система управления файлами на основе FAT, позволяющая непосредственно обращаться к файлам, находящимся в разделах с этой файловой системой. Поддерживается также файловая система ISO 9660 CD-ROM для работы с дисками CD-ROM. Имеются системы управления файлами и на томах с HPFS и NTFS, правда, они работают только на чтение файлов. Созданы варианты системы управления файлами и для доступа к FAT32; эта файловая система в операционной системе Linux называется VFAT.

---

[1] Справедливости ради следует заметить, что в последнее время в Linux наметились тенденции все большего отхода от принятых в семействе UNIX стандартов и увеличения количества различий в разных дистрибутивах Linux. Эти различия распространяются и на структуру каталогов файловой системы, что приводит к определенным проблемам при переносе прикладных программ из одной системы Linux в другую.

Linux, как и все UNIX-системы, поддерживает полный набор протоколов стека TCP/IP для сетевой работы. Программное обеспечение для работы в Интернет/интранет включает драйверы устройств для многих популярных сетевых адаптеров технологии Ethernet, протоколы SLIP (Serial Line Internet Protocol), PLIP (Parallel Line Internet Protocol), PPP (Point-to-Point Protocol), NFS (Network File System) и пр. Поддерживается весь спектр клиентов и услуг TCP/IP, таких как FTP, telnet, NNTP и SMTP.

Ядро Linux сразу было создано с учетом возможностей защищенного режима 32-разрядных процессоров 80386 и 80486 фирмы Intel. В частности, в Linux используется парадигма описания памяти в защищенном режиме и другие новые свойства процессоров с архитектурой ia32. Для защиты пользовательских программ друг от друга и операционной системы от них Linux работает исключительно в защищенном режиме[1], реализованном в процессорах фирмы Intel. В защищенном режиме только программный код, исполняющийся в нулевом кольце защиты, имеет непосредственный доступ к аппаратным ресурсам компьютера — памяти и устройствам ввода-вывода. Пользовательские и системные обрабатывающие программы работают в третьем кольце защиты. Они обращаются к аппаратным ресурсам компьютера исключительно через системные подпрограммы, функционирующие в нулевом кольце защиты. Таким образом, пользовательским программам предоставляются только те услуги, которые реализованы разработчиками операционной системы. При этом системные подпрограммы обеспечивают выполнение только тех функций, которые безопасны с точки зрения операционной системы.

Как и в классических UNIX-системах, Linux имеет макроядро, которое содержит уже известные нам три подсистемы. Ядро обеспечивает выделение каждому процессу отдельного адресного пространства, так что процесс не имеет возможности непосредственного доступа к данным других процессов и ядра операционной системы. Тем более что сегмент кода, сегмент данных и стек ядра располагаются в нулевом кольце защиты. Для обращения к физическим устройствам компьютера ядро вызывает соответствующие драйверы, управляющие аппаратурой компьютера. Поскольку драйверы функционируют в составе ядра, их код будет выполняться в нулевом (привилегированном) кольце защиты, и они могут получить прямой доступ к аппаратным ресурсам компьютера.

В отличие от старых версий UNIX, в которых задачи выгружались во внешнюю память на магнитных дисках целиком, ядро Linux использует аппаратную поддержку процессорами страничного механизма организации виртуальной памяти. Поэтому в Linux замещаются отдельные страницы. То есть с диска в память загружаются те виртуальные страницы образа, которые сейчас реально требуются, а неиспользуемые страницы выгружаются на диск в файл подкачки. Возможно разделение страниц кода, то есть использование одной страницы, физически уже один раз загруженной в память, несколькими процессами. Другими словами, реентерабельность кода, присущая всем UNIX-системам, осталась. В настоящее время имеются ядра для этой системы, оптимизированные для работы с процессорами Intel и AMD

[1] Напомним, что только в этом режиме процессоры с архитектурой ia32 используют 32-разрядную адресацию и имеют доступ ко всей оперативной памяти.

последнего поколения, хотя основные архитектурные особенности защищенного режима работы изменились мало. Уже разработаны ядра для работы с 64-разрядными процессорами от Intel и AMD.

Ядро также поддерживает универсальный пул памяти для пользовательских программ и дискового кэша. При этом для кэширования может использоваться вся свободная память, и наоборот, требуемый объем памяти, отводимой для кэширования файлов, уменьшается при работе больших программ. Этот механизм, называемый агрессивным кэшированием, позволяет более эффективно расходовать имеющуюся память и увеличить производительность системы.

Исполняемые программы задействуют динамически связываемые библиотеки (Dynamic Link Library, DLL), то есть эти программы могут совместно использовать библиотеку, представленную одним физическим файлом на диске. Это позволяет занимать меньше места на диске исполняемым файлам, особенно тем, которые многократно вызывают библиотечные функции. Есть также статические связываемые библиотеки для тех, кто желает пользоваться отладкой на уровне объектных кодов или иметь «полные» исполняемые программы, не нуждающиеся в разделяемых библиотеках. В Linux разделяемые библиотеки динамически связываются во время выполнения, позволяя программисту заменять библиотечные модули своими собственными.

Операционная система FreeBSD

Помимо Linux к свободно распространяемым операционным системам семейства UNIX следует отнести FreeBSD. Принципиальное и самое важное различие между этими операционными системами заключается в том, что согласно принятому соглашению в системы Linux каждый может внести свои изменения, но при этом обязан также сделать свой код открытым. Не все компании на это согласны. Многие предпочитают воспользоваться исходными текстами и готовыми решениями, но не открывать секретов своего программного обеспечения, сделанного с помощью использованного открытого кода. Поэтому в настоящее время сложилась такая ситуация, что имеется уже несколько десятков компаний, занимающихся созданием дистрибутивов для этой операционной системы. Каждая компания, подготавливающая дистрибутив, помимо собственно операционной системы добавляет к нему свой инсталлятор[1], утилиты, в том числе менеджер пакетов программ, конфигураторы и, наконец, большой набор прикладного программного обеспечения. При этом она привносит в систему свои изменения, не согласуя их с другими (за исключением самого ядра, работу над которым по-прежнему курирует Торвальдс). Таким образом, можно констатировать, что у системы Linux как совокупности собственно операционной системы и программного обеспечения, поставляемого с ней, нет единого координатора. С одной стороны, это приводит к заметному прогрессу системы, она быстро реагирует на новые устройства и технологии. Сейчас уже на компьютере с Linux можно играть в современные трехмерные игры, просматри-

[1] Программа установки программного обеспечения на компьютер, в том числе программа установки операционной системы (от англ. «install» — установить).

вать видеофильмы, кодированные в соответствии с самыми современными форматами, слушать и писать музыку и т. д. С другой стороны, пользователи сталкиваются с проблемами переносимости приложений, созданных для этих (и других UNIX-подобных) систем, поскольку нет единого координатора.

В противоположность Linux операционная система FreeBSD имеет такого координатора — это университет в Беркли, Калифорния. Любой может изучить тексты кодов этой операционной системы и предложить внести в нее свои изменения, но это не означает, что так и будет сделано, даже если изменения разумны. Только координирующая группа BSD имеет на это право.

Итак, FreeBSD — это тоже UNIX-подобная операционная система с открытым исходным кодом. Однако несмотря на то, что она родилась раньше и в той же мере бесплатна, что и Linux, многие о ней даже не слышали. Дело в том, что эта операционная система не имеет такой раскрученной рекламы, как проект Linux, хотя история BSD уходит корнями в более далекие годы. При этом необходимо заметить, что в плане производительности, стабильности, качества кода специалисты практически единодушно отдают предпочтение операционной системе FreeBSD. В частности, еще одним важным отличием FreeBSD от Linux является то, что ядро FreeBSD построено по принципам микроядерных операционных систем, тогда как Linux — это макроядерная операционная система.

Сетевая операционная система реального времени QNX

Вспомним основные принципы, обязательная реализация которых позволяет создавать операционные системы реального времени (ОСРВ). Первым обязательным требованием к архитектуре операционной системы реального времени является *многозадачность* в истинном смысле этого слова. Очевидно, что варианты с псевдомногозадачностью (а точнее, с невытесняющей многозадачностью) в системах Windows 3.X или Novell NetWare неприемлемы, поскольку они допускают возможность блокировки или даже полного развала системы одним неправильно работающим процессом. Для предотвращения блокировок вычислений ОСРВ должна использовать квантование времени (то есть использовать вытесняющую, а не кооперативную многозадачность), что сделать достаточно просто. Вторая проблема — организация *надежных вычислений* — может быть эффективно решена за счет специальных аппаратных возможностей процессора. При построении системы для работы на персональных компьютерах типа IBM PC для этого необходимы процессоры типа Intel 80386 и выше, чтобы иметь возможность организовать функционирование операционной системы в защищенном (32-разрядном) режиме работы процессора. Для эффективного обслуживания прерываний операционная система должна использовать алгоритм диспетчеризации, обеспечивающий *вытесняющее планирование, основанное на приоритетах*. Наконец, крайне желательна эффективная поддержка *сетевых коммуникаций* и наличие развитых механизмов *взаимодействия между процессами*, поскольку реальные технологические системы обычно управляются целым комплексом компьютеров и/или контроллеров. Весь-

ма желательно также, чтобы операционная система поддерживала многопоточность (не только мультипрограммный, но и мультизадачный режимы) и симметричную мультипроцессорность. И наконец, при соблюдении всех перечисленных условий операционная система должна быть способна *работать на ограниченных аппаратных ресурсах*, поскольку одна из ее основных областей применения — встроенные системы. К сожалению, данное условие обычно реализуется путем простого урезания стандартных сервисных средств.

Операционная система QNX является мощной операционной системой, разработанной для процессоров с архитектурой ia32. Она позволяет проектировать сложные программные комплексы, работающие в реальном времени как на отдельном компьютере, так и в локальной вычислительной сети. Встроенные средства QNX обеспечивают поддержку многозадачного режима на одном компьютере и взаимодействие параллельно выполняемых задач на разных компьютерах, работающих в среде локальной вычислительной сети. Таким образом, эта операционная система хорошо подходит для построения распределенных систем.

Основным языком программирования в системе является C. Основная операционная среда соответствует стандарту POSIX. Это позволяет с небольшими доработками переносить ранее разработанное программное обеспечение в QNX для организации их работы в среде распределенной обработки.

Операционная система QNX, будучи сетевой и мультизадачной, в то же время является многопользовательской (многотерминальной). Кроме того, она масштабируема. С точки зрения пользовательского интерфейса и интерфейса прикладного программирования она очень похожа на UNIX, поскольку выполняет требования стандарта POSIX. Однако QNX — это не версия UNIX, хотя почему-то многие так считают. Система QNX была разработана, что называется, «с нуля» канадской фирмой QNX Software Systems Limited в 1989 году по заказу Министерства обороны США, причем на совершенно иных архитектурных принципах, нежели использовались при создании операционной системы UNIX.

QNX была первой коммерческой операционной системой, построенной на принципах микроядра и обмена сообщениями. Система реализована в виде совокупности независимых (но взаимодействующих путем обмена сообщениями) процессов различного уровня (менеджеры и драйверы), каждый из которых реализует определенный вид услуг. Эти идеи позволили добиться нескольких важнейших преимуществ. Вот как об этом написано на сайте, посвященном операционной системе QNX [14].

- *Предсказуемость* означает применимость системы к задачам жесткого реального времени. QNX является операционной системой, которая дает полную гарантию того, что процесс с наивысшим приоритетом начнет выполняться практически немедленно, и критически важное событие (например, сигнал тревоги) никогда не будет потеряно. Ни одна версия UNIX не может достичь подобного качества, поскольку нереентерабельный код ядра слишком велик. Любой системный вызов из обработчика прерывания в UNIX может привести к непредсказуемой задержке (то же самое можно сказать про Windows NT).

- *Масштабируемость* и *эффективность* достигаются оптимальным использованием ресурсов и означают применимость QNX для встроенных (embedded) систем. В данном случае мы не увидим в каталоге /dev множества файлов, соответствующих ненужным драйверам, что характерно для UNIX-систем. Драйверы и менеджеры можно запускать и удалять (кроме файловой системы, что очевидно) динамически, просто из командной строки. Мы можем иметь только те услуги, которые нам реально нужны, причем это не требует серьезных усилий и не порождает проблем.
- *Расширяемость* и *надежность* обеспечиваются одновременно, поскольку написанный драйвер не нужно компилировать в ядро, рискуя вызвать нестабильность системы. Менеджеры ресурсов (служба логического уровня) работают в третьем кольце защиты, и вы можете добавлять свои менеджеры, не опасаясь за систему. Драйверы работают в первом кольце и могут вызвать проблемы, но не фатального характера. Кроме того, их достаточно просто писать и отлаживать.
- *Быстрый сетевой протокол FLEET*[1] прозрачен для обмена сообщениями, автоматически обеспечивает отказоустойчивость, балансирование нагрузки и маршрутизацию между альтернативными путями доступа.
- *Компактная графическая подсистема Photon*, построенная на тех же принципах модульности, что и сама операционная система, позволяет получить полнофункциональный интерфейс GUI (расширенный интерфейс Motif), работающий вместе с POSIX-совместимой операционной системой всего в 4 Мбайт памяти, начиная с i80386 процессора.

Архитектура системы QNX

Итак, QNX — это операционная система реального времени для персональных компьютеров, позволяющая эффективно организовать распределенные вычисления. В системе реализована концепция связи между задачами на основе сообщений, посылаемых от одной задачи к другой, причем задачи эти могут решаться как на одном и том же компьютере, так и на разных, но связанных между собой локальной вычислительной сетью. Реальное время и концепция связи между процессами посредством сообщений оказывают решающее влияние и на разрабатываемое для операционной системы QNX программное обеспечение, и на программиста, стремящегося с максимальной выгодой использовать преимущества системы.

Микроядро операционной системы QNX имеет объем всего в несколько десятков килобайтов (в одной из версий — 10 Кбайт, в другой — менее 32 Кбайт, хотя есть вариант и на 46 Кбайт), то есть это одно из самых маленьких ядер среди всех существующих операционных систем. В этом объеме помещаются [26]:

- механизм передачи сообщений между процессами IPC (Inter Process Communication — взаимодействие между процессами);
- редиректор (redirector) прерываний;

[1] Это фирменная технология, о которой несколько более подробно рассказано далее.

- блок планирования выполнения задач (иначе говоря, диспетчер задач);
- сетевой интерфейс для перенаправления сообщений (менеджер Net).

Механизм IPC обеспечивает пересылку сообщений между процессами и является одной из важнейших частей операционной системы, так как все взаимодействие между процессами, в том числе и системными, происходит через сообщения. Сообщение в операционной системе QNX — это последовательность байтов произвольной длины (0–65 535 байт) произвольного формата. Протокол обмена сообщениями может выглядеть, например, таким образом. Задача блокируется для ожидания сообщения. Другая задача посылает первой сообщение и при этом блокируется сама, ожидая ответа. Первая задача деблокируется, обрабатывает сообщение и отвечает, деблокируя вторую задачу.

Сообщения и ответы, пересылаемые между процессами при их взаимодействии, находятся в теле отправляющего их процесса до того момента, когда они могут быть приняты. Это означает, что, с одной стороны, снижается вероятность повреждения сообщения в процессе передачи, а с другой — уменьшается объем оперативной памяти, необходимый для работы ядра. Кроме того, становится меньше пересылок из памяти в память, что разгружает процессор. Особенностью процесса передачи сообщений является то, что в сети, состоящей из нескольких компьютеров, работающих под управлением QNX, сообщения могут прозрачно передаваться процессам, выполняющимся на любом из узлов. Определены в QNX еще и два дополнительных метода передачи сообщений — *метод представителей* (proxy) и *метод сигналов* (signal).

Представители используются в случаях, когда процесс должен передать сообщение, но не должен при этом блокироваться на передачу. Тогда вызывается функция qnx_proxy_attach() и создается представитель. Он накапливает в себе сообщения, которые должны быть доставлены другим процессам. Любой процесс, знающий идентификатор представителя, может вызвать функцию Trigger(), после чего будет доставлено первое в очереди сообщение. Функция Trigger() может вызываться несколько раз, и каждый раз представитель будет доставлять следующее сообщение. При этом представитель содержит буфер, в котором может храниться до 65 535 сообщений.

Как известно, механизм сигналов уже давно используется в операционных системах, в том числе и в UNIX. Операционная система QNX также поддерживает множество сигналов, совместимых с POSIX, большое количество сигналов, традиционно использовавшихся в UNIX (поддержка этих сигналов требуется для совместимости с переносимыми приложениями, ни один из системных процессов QNX их не генерирует), а также несколько сигналов, специфичных для самой системы QNX. По умолчанию любой сигнал, полученный процессом, приводит к завершению процесса (кроме нескольких сигналов, которые по умолчанию игнорируются). Но процесс с приоритетом уровня суперпользователя может защититься от нежелательных сигналов. В любом случае процесс может содержать обработчик для каждого возможного сигнала. Сигналы удобно рассматривать как разновидность программных прерываний.

Редиректор прерываний является частью ядра и занимается перенаправлением аппаратных прерываний в связанные с ними процессы. Благодаря такому подходу

возникает один побочный эффект — с аппаратной частью компьютера работает ядро, оно перенаправляет прерывания процессам — обработчикам прерываний. Обработчики прерываний обычно встроены в процессы, хотя каждый из них исполняется асинхронно с процессом, в который он встроен. Обработчик исполняется в контексте процесса и имеет доступ ко всем глобальным переменным процесса. При работе обработчика прерываний прерывания разрешены, но обработчик приостанавливается только в том случае, если произошло более высокоприоритетное прерывание. Если это позволяется аппаратной частью, к одному прерыванию может быть подключено несколько обработчиков, каждый из которых получит управление при возникновении прерывания.

Этот механизм позволяет пользователю избегать работы с аппаратным обеспечением напрямую и тем самым избегать конфликтов между различными процессами, работающими с одним и тем же устройством. Для обработки сигналов от внешних устройств чрезвычайно важно минимизировать время между возникновением события и началом непосредственной его обработки. Этот фактор существен в любой области применения: от работы терминальных устройств до обработки высокочастотных сигналов.

Блок планирования выполнения задач обеспечивает многозадачность. В этом плане операционная система QNX предоставляет разработчику огромный простор для выбора той дисциплины выделения ресурсов процессора задаче, которая обеспечит наиболее подходящие условия для выполнения критически важных приложений, а обычным приложениям обеспечит такие условия, при которых они будут выполняться за разумное время, не мешая работе критически важных приложений.

К выполнению своих функций как диспетчера ядро приступает в следующих случаях:

- какой-либо процесс вышел из блокированного состояния;
- истек квант времени для процесса, владеющего центральным процессором;
- работающий процесс прерван каким-либо событием.

Диспетчер выбирает процесс для запуска среди неблокированных процессов в порядке значений их приоритетов в диапазоне от 0 (наименьший) до 31 (наибольший). Обслуживание каждого из процессов зависит от метода его диспетчеризации (приоритет и метод диспетчеризации могут динамически меняться во время работы). В QNX существуют три метода диспетчеризации:

- очередь (First In First Out, FIFO) — раньше пришедший процесс раньше обслуживается;
- карусель (Round Robin, RR) — процессу выделяется определенный квант времени для работы, после чего процессор предоставляется следующему процессу;
- адаптивный метод (используется чаще других).

Метод FIFO наиболее близок к невытесняющей многозадачности. То есть процесс выполняется до тех пор, пока он не перейдет в состояние ожидания сообщения, в состояние ожидания ответа на сообщение или не отдаст управление ядру. При переходе в одно из таких состояний процесс помещается последним в очередь про-

цессов с таким же уровнем приоритета, а управление передается процессу с наибольшим приоритетом.

В методе RR все происходит так же, как и в предыдущем, с той разницей, что период, в течение которого процесс может работать без перерыва, ограничивается неким квантом времени.

Процесс, работающий в соответствии с адаптивным методом, ведет себя следующим образом:

- если процесс полностью использует выделенный ему квант времени, а в системе есть готовые к исполнению процессы с тем же уровнем приоритета, его приоритет снижается на 1;
- если процесс с пониженным приоритетом остается необслуженным в течение секунды, его приоритет увеличивается на 1;
- если процесс блокируется, ему возвращается исходное значение приоритета.

По умолчанию процессы запускаются в режиме адаптивной многозадачности. В этом же режиме работают все системные утилиты QNX. Процессы, работающие в разных режимах многозадачности, могут одновременно находиться в памяти и исполняться. Важный элемент реализации многозадачности — приоритет процесса. Обычно приоритет процесса устанавливается при его запуске. Но есть дополнительная возможность, называемая *вызываемым клиентом приоритетом*. Как правило, она реализуется для серверных процессов (исполняющих запросы на какое-либо обслуживание). При этом приоритет процесса-сервера устанавливается только на время обработки запроса и становится равным приоритету процесса-клиента.

Сетевой интерфейс в операционной системе QNX является неотъемлемой частью ядра. Он, конечно, взаимодействует с сетевым адаптером через сетевой драйвер, но базовые сетевые службы реализованы на уровне ядра. При этом передача сообщения процессу, находящемуся на другом компьютере, ничем не отличается с точки зрения приложения от передачи сообщения процессу, выполняющемуся на том же компьютере. Благодаря такой организации сеть превращается в однородную вычислительную среду. При этом для большинства приложений не имеет значения, с какого компьютера они были запущены, на каком исполняются и куда поступают результаты их работы.

Все службы операционной системы QNX, не реализованные непосредственно в ядре, работают как обычные стандартные процессы в полном соответствии с основными концепциями микроядерной архитектуры. С точки зрения операционной системы эти системные процессы ничем не отличаются от всех остальных. Как, впрочем, и драйверы устройств. Единственное, что нужно сделать, чтобы новый драйвер устройства стал частью операционной системы, — изменить конфигурационный файл системы так, чтобы драйвер запускался при загрузке.

Основные механизмы организации распределенных вычислений

QNX является сетевой операционной системой, которая позволяет организовать эффективные распределенные вычисления. Для этого на каждой машине, называ-

емой узлом, помимо ядра и менеджера процессов должен быть запущен уже упомянутый ранее менеджер Net. Менеджер Net не зависит от аппаратной реализации сети. Эта аппаратная независимость обеспечивается за счет сетевых драйверов. В операционной системе QNX имеются драйверы для сетей с различными технологиями: Ethernet и FastEthernet, Arcnet, IBM Token Ring и др. Кроме того, имеется возможность организации сети через последовательный канал или модем.

В QNX версии 4 полностью реализовано встроенное сетевое взаимодействие типа «точка-точка». Например, сидя за машиной *A*, вы можете скопировать файл с гибкого диска, подключенного к машине *B*, на жесткий диск, подключенный к машине *C*. По существу, сеть из машин с операционными системами QNX действует как один мощный компьютер. Любые ресурсы (модемы, диски, принтеры) могут быть добавлены к системе простым их подключением к любой машине в сети. QNX обеспечивает возможность одновременной работы в сетях Ethernet, Arcnet, Serial и Token Ring, более одного пути для связи и балансировку нагрузки в сетях. Если кабель или сетевая плата выходит из строя и связь через эту сеть прекращается, система автоматически перенаправит данные через другую сеть. Все это происходит в режиме подключения (on-line), предоставляя пользователю автоматическую сетевую избыточность и увеличивая эффективность взаимодействия во всей системе.

Каждому узлу в сети соответствует уникальный целочисленный идентификатор — логический номер узла. Любой поток выполнения в сети QNX имеет прозрачный доступ (при наличии достаточных привилегий) ко всем ресурсам сети; то же самое относится и к взаимодействию потоков. Для взаимодействия потоков, находящихся на разных узлах сети, используются те же самые вызовы ядра, что и для потоков, выполняемых на одном узле. В том случае, если потоки находятся на разных узлах сети, ядро переадресует запрос менеджеру сети. Для обмена в сети используется надежный и эффективный протокол транспортного уровня FLEET. Каждый из узлов может принадлежать одновременно нескольким QNX-сетям. В том случае, если сетевое взаимодействие может быть реализовано несколькими путями, для передачи выбирается менее загруженная и более скоростная сеть.

Сетевое взаимодействие является узким местом в большинстве операционных систем и обычно создает значительные проблемы для систем реального времени. Для того чтобы обойти это препятствие, разработчики операционной системы QNX создали собственную специальную сетевую технологию FLEET и соответствующий протокол транспортного уровня FTL (FLEET Transport Layer). Этот протокол не базируется ни на одном из распространенных сетевых протоколов вроде IPX или NetBios и обладает рядом качеств, которые делают его уникальным. Основные его качества зашифрованы в аббревиатуре FLEET, которая расшифровывается следующим образом:

- **F**ault-Tolerant Networking — QNX может одновременно использовать несколько физических сетей, при выходе из строя любой из них данные будут «на лету» перенаправлены через другую сеть;
- **L**oad-Balancing on the Fly — при наличии нескольких физических соединений QNX автоматически распараллеливает передачу пакетов по соответствующим сетям;

- **E**fficient Performance — специальные драйверы, разрабатываемые фирмой QSSL для широкого спектра оборудования, позволяют использовать это оборудование с максимальной эффективностью;
- **E**xtensable Architecture — любые новые типы сетей могут быть поддержаны путем добавления соответствующих драйверов;
- **T**ransparent Distributed Processing — благодаря отсутствию разницы между передачей сообщений в пределах одного узла и между узлами нет необходимости вносить какие-либо изменения в приложения, для того чтобы они могли взаимодействовать через сеть.

Благодаря технологии FLEET сеть компьютеров с операционными системами QNX фактически можно представлять как один виртуальный суперкомпьютер. Все ресурсы любого из узлов сети автоматически доступны другим, и для этого не нужно создавать никаких дополнительных механизмов с использованием технологии RPC. Это значит, что любая программа может быть запущена на любом узле, причем ее входные и выходные потоки могут быть направлены на любое устройство на любых других узлах [18].

Например, утилита make в операционной системе QNX автоматически распараллеливает компиляцию пакетов из нескольких модулей на все доступные узлы сети, а затем собирает исполняемый модуль по мере завершения компиляции на узлах. Специальный драйвер, входящий в комплект поставки, позволяет использовать для сетевого взаимодействия любое устройство, с которым может быть ассоциирован файловый дескриптор, например последовательный порт, что открывает возможности для создания глобальных сетей.

Достигаются все эти удобства за счет того, что поддержка сети частично обеспечивается и микроядром (специальный код в его составе позволяет операционной системе QNX фактически объединять все микроядра в сети в одно ядро). Разумеется, за такие возможности приходится платить тем, что мы не можем получить драйвер для какой-либо сетевой платы от кого-либо еще, кроме фирмы QSSL, то есть использоваться может только то оборудование, которое уже поддерживается. Однако ассортимент такого оборудования достаточно широк и периодически пополняется новейшими устройствами.

Когда ядро получает запрос на передачу данных процессу, находящемуся на удаленном узле, он переадресовывает этот запрос менеджеру Net, в подчинении которого находятся драйверы всех сетевых карт. Имея перед собой полную картину состояния всего сетевого оборудования, Net может отслеживать состояние каждой сети и динамически перераспределять нагрузку между ними. В случае, когда одна из сетей выходит из строя, поток данных автоматически перенаправляется в другую доступную сеть, что очень важно при построении высоконадежных систем. Кроме поддержки собственного протокола, Net обеспечивает передачу пакетов TCP/IP, SMB (Server Message Block)[1] и многих других, используя то же сете-

[1] Сетевая технология взаимодействия клиента и сервера, разработанная фирмой IBM и активно используемая компанией Microsoft в своих операционных системах. В последнее время компания Microsoft стала называть ее CIFS (Common Internet File System).

вое оборудование. При этом производительность компьютеров с операционной системой QNX в сети приближается к производительности аппаратного обеспечения — настолько малы задержки, вносимые операционной системой.

При проектировании системы реального времени, как правило, необходимо обеспечить одновременное выполнение нескольких приложений. В QNX/Neutrino[1] параллельность выполнения достигается за счет использования *потоковой модели POSIX*, в которой процессы в системе представляются в виде совокупности потоков выполнения. Поток является минимальной единицей выполнения и диспетчеризации для ядра Neutrino; процесс определяет адресное пространство для потоков. Каждый процесс состоит минимум из одного потока. Операционная система QNX предоставляет богатый набор функций для синхронизации потоков. В отличие от потоков, само ядро не подлежит диспетчеризации. Код ядра исполняется только в том случае, когда какой-нибудь поток вызывает функцию ядра, или при обработке аппаратного прерывания.

Напомним, что операционная система QNX базируется на концепции *передачи сообщений*. Передачу и диспетчеризацию сообщений осуществляет ядро системы. Кроме того, ядро управляет временными прерываниями. Выполнение остальных функций обеспечивается задачами-администраторами. Программа, желающая создать задачу, посылает сообщение администратору задач (модуль task) и блокируется для ожидания ответа. Если новая задача должна выполняться одновременно с порождающей ее задачей, администратор задач task создает ее и, отвечая, выдает порождающей задаче идентификатор созданной задачи. В противном случае никакого сообщения не посылается до тех пор, пока новая задача не закончится сама по себе. Тогда в ответе администратора задач будут содержаться конечные характеристики закончившейся задачи.

Сообщения различаются количеством данных, которые передаются от одной задачи точно к другой задаче. Данные копируются из адресного пространства первой задачи в адресное пространство второй, и выполнение первой задачи приостанавливается до тех пор, пока вторая задача не вернет ответное сообщение. В действительности обе задачи кратковременно взаимодействуют во время выполнения передачи. Ничто, кроме длины сообщения (максимальная длина может достигать 64 Кбайт), не заботит QNX при передаче сообщения. Существует несколько протоколов, которые могут быть использованы для этой цели.

Основные операции над сообщениями: *послать*, *получить* и *ответить*, а также несколько их вариантов для обработки специальных ситуаций. Получатель всегда идентифицируется своим идентификатором задачи, хотя существуют способы ассоциировать имена с идентификатором задачи. Наиболее интересные варианты операций включают в себя возможность получать (копировать) только первую часть сообщения, а затем получать оставшуюся часть такими кусками, какие потребуются. Это может быть полезным, поскольку позволяет сначала узнать длину сообщения, а затем динамически распределить принимающий буфер. Если необходимо задержать ответное сообщение до тех пор, пока не будет получено и обра-

[1] Neutrino — один из проектов микроядерной ОС.

ботано другое сообщение, то чтение первых нескольких байтов дает вам компактный «обработчик», через который позже можно получить доступ ко всему сообщению. Таким образом, задача оказывается избавленной от необходимости хранить в себе большое количество буферов.

Другие функции позволяют программе получать сообщения только тогда, когда она уже ожидает их приема, а не блокироваться до тех пор, пока не прибудет сообщение. Можно также транслировать сообщение другой задаче без изменения идентификатора передатчика. Задача, которая транслировала сообщение, в транзакции невидима.

Кроме того, операционная система QNX обеспечивает объединение сообщений в структуру данных, называемую очередью. *Очередь сообщений* — это просто область данных в третьей, отдельной задаче, которая временно принимает передаваемое сообщение и немедленно отвечает передатчику. В отличие от стандартной передачи сообщений, передатчик немедленно освобождается для того, чтобы продолжить свою работу. Задача администратора очереди — хранить в себе сообщение до тех пор, пока приемник не будет готов прочитать его; делает он это, запрашивая сообщение у администратора очереди. Любое количество сообщений (ограничено только возможностью памяти) может храниться в очереди. Сообщения хранятся и передаются в том порядке, в котором они были приняты. Может быть создано любое количество очередей. Каждая очередь идентифицируется своим именем.

Помимо сообщений и очередей в операционной системе QNX для взаимодействия задач и организации распределенных вычислений имеются так называемые *порты*, которые позволяют формировать сигнал одного конкретного условия и механизм исключений, о котором мы уже упоминали ранее.

Порт подобен флагу, известному всем задачам на одном и том же узле (но не на разных узлах). Он имеет только два состояния, которые могут трактоваться как «присоединить» и «освободить», хотя пользователь может интерпретировать их по-своему, например «занят» и «доступен». Порты используются для быстрой простой синхронизации между задачей и обработчиком прерываний устройства. Они нумеруются от нуля до 32 максимум (на некоторых типах узлов возможно и больше). Первые 20 номеров зарезервированы для операционной системы.

С портом может быть выполнено три операции:

- присоединить порт,
- отсоединить порт,
- послать сигнал в порт.

Одновременно к порту может быть присоединена только одна задача. Если другая задача попытается «отсоединиться» от того же самого порта, то произойдет отказ при вызове функции, и управление вернется к задаче, которая в настоящий момент присоединена к этому порту. Это самый быстрый способ обнаружить идентификатор другой задачи, подразумевая, что задачи могут договориться использовать один номер порта. Напомним, что все рассматриваемые задачи должны находиться на одном и том же узле. При работе нескольких узлов специальные функции обеспечивают большую гибкость и эффективность.

Любая задача может посылать сигнал в любой порт независимо от того, была она присоединена к нему или нет (предпочтительно, чтобы не была). Сигнал подобен неблокирующей передаче пустого сообщения. То есть передатчик не приостанавливается, а приемник не получает какие-либо данные; он только отмечает, что конкретный порт изменил свое состояние.

Задача, присоединенная к порту, может ожидать прибытия сигнала или может периодически читать порт. Система QNX хранит информацию о сигналах, передаваемых в каждый порт, и уменьшает счетчик после каждой операции «приема» сигнала («чтение» возвращает счетчик и устанавливает его в нуль). Сигналы всегда принимают перед сообщениями, давая им тем самым больший приоритет над сообщениями. В этом смысле сигналы часто используются обработчиками прерываний для того, чтобы оповестить задачу о внешних (аппаратных) событиях. Действительно, обработчики прерываний не имеют возможности посылать сообщения и должны использовать сигналы.

В отличие от описанных выше методов, которые строго синхронизируются, *исключения* обеспечивают асинхронное взаимодействие. То есть исключение может прервать нормальное выполнение потока задачи. Они, таким образом, являются аварийными событиями. Операционная система QNX резервирует для себя 16 исключений, чтобы оповещать задачи о прерываниях с клавиатуры, нарушении памяти и подобных необычных ситуациях. Остальные 16 исключений могут быть определены и использованы прикладными задачами.

Системная функция может быть вызвана для того, чтобы позволить задаче реализовать собственный механизм обработки исключений и во время возникновения исключения выполнять свою внутреннюю функцию.

Заметим, что функция исключения задачи вызывается асинхронно операционной системой, а не самой задачей. Поэтому исключения могут негативно повлиять на операции (например, передачу сообщений), которые выполняются в это же время. Обработчики исключений должны быть написаны очень аккуратно.

Одна задача может установить одно или несколько исключений для другой задачи. Эти исключения могут быть комбинацией системных исключений и исключений, определяемых приложениями, обеспечивая другие возможности для межзадачного взаимодействия.

Благодаря такому свойству QNX, как возможность обмена посланиями между задачами и узлами сети, программы не заботятся о конкретном размещении ресурсов в сети. Это свойство придает системе необычную гибкость. Так, узлы могут произвольно добавляться в систему и изыматься из системы, не затрагивая системные программы. QNX имеет эту конфигурационную независимость благодаря концепции *виртуальных задач*. У виртуальных задач непосредственный код и данные, будучи на одном из удаленных узлов, возникают и ведут себя так, как если бы они были локальными задачами какого-то узла со всеми их атрибутами и привилегиями. Программа, посылающая сообщение в сеть, никогда не направляет его точно. Сначала она открывает *виртуальный канал*. *Виртуальный канал* связывает между собой все виртуальные задачи. На обоих концах такой связи имеются буферы, которые позволяют хранить самое большое послание из тех, которые канал

может нести в данном сеансе связи. Сетевой администратор помещает в эти буферы все сообщения для соединенных задач. Виртуальная задача, таким образом, занимает всего лишь пространство, необходимое для буфера и входа в таблице задач. Чтобы открыть виртуальный канал, необходимо знать идентификатор узла и задачи, с которой устанавливается связь. Для этого требуется идентификатор задачи-администратора, ответственного за данную функцию, или глобальное имя сервера. Не раскрывая здесь подробно механизм обмена посланиями, добавим лишь, что задача может вообще выполняться на другом узле, где, допустим, имеется более совершенный процессор.

Семейство операционных систем OS/2 Warp компании IBM

История появления, расцвета и практического ухода со сцены операционных систем под общим названием OS/2 и странна, и поучительна. Будучи одной из самых лучших операционных систем для персональных компьютеров по очень большому числу параметров и появившись существенно раньше систем своих основных конкурентов, она тем не менее не смогла стать самой распространенной, хотя могла бы, и с легкостью. Основная причина тому — законы бизнеса (умение рекламировать свой товар, всячески поддерживать его продвижение, вкладывать деньги в завоевание рынка), а не качество самой операционной системы. Во-первых, компания IBM не сочла необходимым продвигать свою операционную систему на рынок программного обеспечения, ориентированного на конечного пользователя, а решила продолжить свою практику работы исключительно с корпоративными клиентами. А этот рынок (корпоративного программного обеспечения) оказался существенно у́же для персональных компьютеров, чем рынок программного обеспечения для конечного пользователя, ибо компьютеры типа IBM PC прежде всего являются персональными. Во-вторых, основные доходы компания IBM получала не от продажи системного программного обеспечения для персональных компьютеров, а за счет продаж дорогостоящих серверов и другого оборудования. Доходы от продажи операционной системы OS/2 не представлялись руководству компании IBM значимыми. Чтобы добиться успеха на рынке операционных систем для персональных компьютеров, необходимо было обеспечить всестороннюю поддержку своей системы соответствующей учебной литературой, широкой рекламой, заинтересовать разработчиков программного обеспечения. Увы, этого сделано не было, и сегодня уже практически мало кто знает о системах семейства OS/2. В то же время следует отметить, что те организации и предприятия, которые в свое время освоили эту систему и создали для нее соответствующее прикладное программное обеспечение, до сих пор не переходят на ныне чрезвычайно популярные операционные системы Windows NT/2000/XP, поскольку последние требуют существенно больше системных ресурсов. Любопытный факт: всем известные банкоматы работают под управлением OS/2.

Семейство 32-разрядных операционных систем OS/2 для IBM-совместимых персональных компьютеров начало свою историю с появления первой OS/2 v 2.0

в 1992 году. Ей предшествовала 16-разрядная операционная система с таким же названием — OS/2, которая была разработана для микропроцессора i80286. Этот микропроцессор, несмотря на множество принципиальных новаций, оказался неудачным. Защищенный режим работы этого 16-разрядного микропроцессора был несовершенным. Он обеспечивал работу с относительно небольшим объемом оперативной памяти, имел слабую аппаратную поддержку для организации виртуальной памяти, слишком низкое быстродействие (для того, чтобы выступать в качестве основы для построения мультизадачных операционных систем). Неудачная судьба 16-разрядной системы OS/2 1.x во многом повлияла и на 32-разрядную операционную систему, хотя по очень многим позициям архитектура 32-разрядной версии операционной системы OS/2 принципиально отличалась от своей предшественницы.

Компания IBM оставила этот проект, когда его версия имела номер 4.5. Сейчас из состава IBM отделилась небольшая компания, которая, выкупив проект OS/2, продолжает над ним работу и обеспечивает приверженцев этой операционной системы пакетами обновления и всевозможными добавлениями.

Все последние версии операционной системы OS/2 в своем названии имеют слово Warp, что переводится с английского как «основа». Операционная система OS/2 Warp 4.0 практически представляет собой OS/2 Warp 3.0 (вышедшую еще в 1994 году) с несколько улучшенной поддержкой DOS-задач и обновленными элементами объектно-ориентированного интерфейса. Для этой системы характерны:

- вытесняющая многозадачность (preemptive multitasking) и поддержка DOS- и Windows- (Win32s[1]) приложений;
- по-настоящему интуитивно понятный и действительно удобный объектный пользовательский интерфейс;
- поддержка стандарта открытого объектного документооборота OpenDoc;
- поддержка стандарта OpenGL;
- поддержка Java-апплетов и встроенных средств разработки на языке Java;
- поддержка шрифтов True Type (TTF);
- управление голосом без предварительной подготовки (технология Voice Type);
- полная поддержка сетевых технологий Интернет/интранет, доступ в сети CompuServe[2];
- средства построения одноранговых сетей и клиентские части для сетевых операционных систем IBM LAN Server, Windows, Lantastic, Novell Netware 4.1 (в том числе поддержка службы каталогов);
- система удаленного доступа через модемные соединения;
- файловая система Mobile File System для поддержки мобильных пользователей;
- стандарт автоматического распознавания аппаратных устройств (Plug-and-Play), но без столь навязчивого механизма, который реализован в Windows;

[1] Win32s — это одно из расширений интерфейса прикладного программирования систем Windows.
[2] Популярная американская служба.

- набор офисных приложений[1] (базы данных, электронные таблицы, текстовый процессор, генератор отчетов, деловая графика, встроенная система приема-передачи факсимильных сообщений, информационный менеджер);
- полная поддержка мультимедиа, включая средства работы с видеокамерой, расширенную систему помощи WarpGuide.

Однако наиболее заманчивы не перечисленные из рекламного буклета возможности системы, а удобная и надежная для работы с корпоративными базами данных и в сетях среда, предоставляющая клиентское рабочее место.

Операционная система OS/2 Warp предлагает единый интерфейс прикладного программирования (API), совместимый с рядом операционных систем, что позволяет снизить стоимость разработок. Все версии операционных систем OS/2 и LAN Server, включая текущие версии OS/2 Warp и OS/2 Warp Server 4.5, совместимы по восходящей линии, что позволяет экономить средства, необходимые для поддержания уже существующих прикладных программ.

Чрезвычайно важным для пользователей является тот факт, что компания IBM для всех версий своей операционной системы регулярно выпускает пакеты обновления (FixPak). Эти пакеты исправляют обнаруженные ошибки, а также вносят новые функции. Для пользователей такая практика сопровождения фирмой своей операционной системы, безусловно, более выгодна, нежели практика частого выпуска новых версий операционных систем (ей следует компания Microsoft).

Так, например, для одной из своих самых удачных операционных систем — Windows NT 4.0 — компания Microsoft выпустила всего 6 пакетов обновления (ServicePak), тогда как для уже совсем старой операционной системы OS/2 Warp 3.0, которая вышла в свет в 1994 году, компания IBM выпустила уже несколько десятков пакетов FixPak. Для операционной системы OS/2 Warp 4.0 вышло более 15 пакетов исправлений и обновлений.

Пакеты исправлений и обновлений пользователи получают бесплатно, тогда как за новую операционную систему приходится платить большие деньги. К тому же, длительная работа по исправлению имеющихся в системе ошибок приводит к тому, что количество последних со временем, как правило, уменьшается и система становится все более надежной и функциональной, в то время как новая версия операционной системы содержит не меньше ошибок, чем предыдущая. Последнее обстоятельство объясняется в том числе и тем, что объем ее исходного кода становится все больше и больше, а времени на создание операционной системы отводится столько же, если не меньше.

Немаловажным фактором является и то, что значительные капиталовложения требуются не только на приобретение новой операционной системы, но и на ее освоение. Для многих желательно, чтобы время жизни операционной системы составляло до 10 лет и более. В противном случае мы будем не только напрасно тратить

[1] Справедливости ради следует заметить, что этот набор приложений (называемый BonusPak) несовместим с современными версиями Microsoft Office, поэтому его используют, как правило, только в «закрытых системах», когда не предусматривается обмен документами, изготовленными посредством приложений Microsoft Office.

деньги на приобретение новых систем, но и не сможем обеспечить квалифицированную работу пользователей в этих системах. Современные операционные системы и прикладное программное обеспечение для своего освоения требуют длительного и дорогостоящего обучения пользователей. Поэтому желательно, чтобы все это программное обеспечение не требовало частого переобучения сотрудников (однако, с другой стороны, прогресс не стоит на месте, и большое количество конечных пользователей с нетерпением ожидают появления все более новых операционных систем и приложений).

Весьма полезным, как для управления приложениями, так и для создания несложных собственных программ, является наличие системы программирования на языке высокого уровня REXX, который иногда называют языком процедур. Можно сказать, что это встроенный командный язык, который служит для тех же целей, что и язык для пакетных (batch) файлов в среде DOS, но он обладает несравнимо большими возможностями. Это язык высокого уровня с нетипизированными переменными. Язык легко расширяем, любая программа OS/2 может добавлять в него новые функции. Помимо встроенного интерпретатора с языка REXX имеется система программирования Visual REXX. Имеется и объектно-ориентированная версия языка REXX с соответствующим интерпретатором.

Наиболее сильное впечатление при работе в операционной системе OS/2 оставляет объектно-ориентированный графический пользовательский интерфейс, а особой популярностью у программистов эта система пользовалась вследствие очень хорошей организации VDM-машин и высокого быстродействия при выполнении обычных DOS-приложений.

Особенности архитектуры и основные возможности

Строение и функционирование операционной системы OS/2 можно считать практически идеальными с точки зрения теории и довольно неплохими в реализации. В качестве подтверждения этому можно привести один пример, который представляется очень показательным: OS/2 до сегодняшних дней практически неизменна, начиная с версии 2.0, увидевшей свет в 1992 году. Этот факт говорит о глубокой продуманности архитектуры системы, ведь и по сей день OS/2 является одной из самых мощных и продуктивных операционных систем. Здесь самым показательным примером являются тесты серверов. В одной из вычислительных лабораторий Санкт-Петербургского государственного университета аэрокосмического приборостроения (ГУАП) с 1995 года в течение нескольких лет функции сервера кафедры вычислительных систем и сетей выполняла система OS/2 Warp Advanced Server. При переходе на сервер Windows NT 4.0 пришлось в два раза увеличить объем оперативной памяти и поменять процессор (с Pentium 90 на Pentium II 300), и даже после этого скорость работы обычных приложений на рабочих станциях не достигла той производительности, какую имели пользователи при работе сервера под управлением OS/2. Аналогичные замечания не так давно можно было прочесть и в зарубежных публикациях — однопроцессорная машина под управлением OS/2 Warp Server обгоняет по производительности двухпроцессорную машину под управлением Windows NT.

Разработчики системы OS/2 решили не использовать всех возможностей защищенного режима, заложенных в микропроцессоры i80x86. Например, обработка прерываний чаще всего ведется через коммутаторы прерываний, а не через коммутаторы задач. Используется плоская модель памяти. Хорошо продуманная архитектура, в которой задействована модель клиент-сервер, и тщательное кодирование позволили получить систему, требующую очень небольших вычислительных ресурсов. Очень удачно реализована диспетчеризация задач. Представление различных системных информационных структур в статической форме (в виде таблиц) привело к более высокому быстродействию.

В OS/2 имеется несколько видов виртуальных машин для выполнения прикладных программ. Собственные 32- и 16-разрядные программы OS/2 выполняются на отдельных виртуальных машинах в режиме вытесняющей многозадачности и могут общаться между собой с помощью средств DDE OS/2. Прикладные программы DOS и Win16 могут запускаться на отдельных виртуальных машинах в многозадачном режиме. При этом они поддерживают полноценные связи DDE и OLE 2.0 друг с другом, а также связи DDE с 32-разрядными программами OS/2. Кроме того, при желании можно запустить несколько программ Win16 на общей виртуальной машине Win16, где они работают в режиме невытесняющей многозадачности, как в Windows 3.x. Конечно, нынче это уже неактуально, поскольку появилось огромное количество приложений, использующих API Win32, но в 90-е годы XX века эти факты имели существенное значение.

Разнообразные сервисные функции API OS/2, в том числе SOM (System Object Model — модель системных объектов), обеспечиваются с помощью системных библиотек DLL, к которым можно обращаться без требующих затрат времени переходов между кольцами защиты. Ядро операционной системы OS/2 предоставляет многие базовые сервисные функции API, обеспечивает поддержку файловой системы, управление памятью, имеет диспетчер аппаратных прерываний. В ядре виртуальных DOS-машин (Virtual DOS Machine, VDM), или в VDM-ядре, осуществляется эмуляция DOS и процессора 8086, а также управление VDM. Драйверы виртуальных устройств обеспечивают уровень аппаратной абстракции. Драйверы физических устройств напрямую взаимодействуют с аппаратурой.

Модуль реализации механизмов виртуальной памяти в ядре OS/2 поддерживает большие постраничные разбросанные адресные пространства, составленные из объектов памяти. Каждый объект памяти управляется так называемым *пейджером* — задачей вне ядра, обеспечивающей резервное хранение страниц объекта памяти. Адресные пространства управляются путем отображения или размещения объектов памяти внутри них. Ядро управляет защитой памяти и ее распределением на основе объектов памяти абстрактным образом, вне зависимости от каких-либо конкретных аппаратных средств трансляции процессорных адресов. В частности, ядро интенсивно использует режим копирования при записи для придания программам способности делить объекты памяти, не копируя множество страниц, когда новое адресное пространство получает доступ к объекту памяти. Новые копии страниц создаются, только когда программа в одном из адресных пространств обновляет их. Когда ядро принимает страничный сбой в объекте памяти и не име-

ет страницы памяти в наличии, или когда оно должно удалить страницы из памяти по требованию других работающих программ, ядро с помощью механизма IPC уведомляет пейджер об объекте памяти, в котором произошел сбой. После этого пейджер сервера приложений определяет, каким образом предоставить или сохранить данные. Это позволяет системе устанавливать различные семантики для объектов памяти, основываясь на потребностях программ, которые их используют.

Ядро управляет средами исполнения для программ, обеспечивая множественность заданий (процессов) и потоков выполнения. Каждое задание (процесс[1]) имеет свое собственное адресное пространство, или отображение. Ядро распределяет объекты памяти, которые задание отобразило на диапазон адресов внутри адресного пространства. Задание также является блоком размещения ресурсов и защиты, при этом заданиям придаются возможности и права доступа к средствам IPC системы. Для поддержки параллельного исполнения с другой программой в пределах одного адресного пространства ядро отделяет среду исполнения от реально выполняющегося потока. Таким образом, программа задания может быть загружена и исполнена в нескольких различных местах кода в одно и то же время на мультипроцессоре или параллельной машине. Это может привести к повышению быстродействия приложения.

Система IPC обеспечивает базовый механизм, позволяющий потокам работать в различных заданиях, взаимодействуя друг с другом, и надежную доставку сообщений в порты. *Порты* представляют собой защищенные каналы связи между заданиями. Каждому заданию, использующему порт, приписывается набор прав на этот порт. Права могут быть различными для разных заданий. Только одно задание может получить какой-либо порт, хотя любой поток внутри задания может выполнять операцию приема. Одно или более заданий могут иметь право посылать информацию в порт. Ядро позволяет заданиям применять систему IPC для передачи друг другу прав на порт. Оно также обеспечивает высокопроизводительный способ передачи больших объемов данных в сообщениях. Вместо того чтобы копировать данные, сообщение содержит указатель на них, который называется указателем на данные вне линии. Когда ядро передает сообщение от передатчика к приемнику, оно заставляет память, передаваемую через указатель, появиться в адресном пространстве приемника и, как вариант, исчезнуть из адресного пространства передатчика. Ядро само структурировано как задание с потоками, и большинство системных служб реализованы как механизмы IPC-обращений к ядру, а не как прямые системные вызовы.

Для поддержки операций ввода-вывода и доступа к внешним устройствам ядро операционной системы OS/2 обеспечивает доступ к ресурсам ввода-вывода, таким как устройства с отображаемой памятью, порты ввода-вывода и каналы прямого доступа к памяти (Direct Memory Access, DMA), а также возможность отображать прерывания на драйверы устройств, исполняемые в пользовательском пространстве. Службы ядра позволяют приоритетным программам получать устройства в свое владение: такими программами обычно являются программы, не связанные с заданиями, вроде серверов драйверов устройств, работающих как приложения. Поскольку ядро обязано обслужить все прерывания (в силу того, что прерывания обычно выдаются

[1] Здесь термины «задание» и известный нам «процесс» используются как синонимы.

в приоритетном состоянии компьютера, а также в целях поддержания целостности системы), оно имеет логику, которая определяет, должно ли оно обрабатывать прерывание или его следует отобразить на сервер. Если прерывание следует отобразить на приложение, это приложение должно быть зарегистрировано в ядре и содержать код, контролирующий отображение прерывания. Сразу после отображения в приложении запускается поток по обработке прерывания.

В соответствии с концепцией микроядерных операционных систем, непосредственно поверх ядра системы OS/2, которое построено с использованием этой архитектуры, располагается ряд служебных приложений, предоставляющих системные службы общего назначения, то есть службы, не зависящие от операционной среды, в которой выполняется приложение. Эти службы зависят только от ядра, некоторых вспомогательных служб, экспортируемых доминирующей задачей операционной системы, и от самих себя. В числе задачно-нейтральных служб имеются пейджер умолчания, мастер-сервер, который загружает другие задачно-нейтральные серверы в память, служба низкоуровневых имен, служба защиты, службы инициализации, набор драйверов устройств со связанным кодом поддержки, а также библиотечные подпрограммы для стандартной программной среды. Дополнительные задачно-нейтральные сервисы, например выделенный файловый сервер, могут быть просто добавлены.

С помощью ядра операционной системы и задачно-нейтральных сервисов приоритетная задача может обеспечить операционную системную среду типа UNIX. Поскольку приоритетная задача является прикладным сервером, можно добавлять другие серверы для различных задач, исполняющих программы, написанные в разных операционных системах, работающих на машине в одно и то же время.

Существуют некоторые операционные системные сервисы (вроде трансляции сообщений об ошибках), не обеспечиваемые задачно-нейтральными сервисами. Поскольку лучше не дублировать подобные сервисы, приоритетная задача предоставляет эти сервисы не только своим клиентским приложениям, но и любой другой задаче, исполняющейся в машине.

Особенности интерфейсов

В операционных системах OS/2 Warp в качестве стандартной графической оболочки используется среда WorkPlace Shell (WPS), организованная логичней и удобней, чем известный Windows-интерфейс. Оболочка Workplace Shell основана на модели системных объектов (SOM) фирмы IBM — мощной технологии, специально разработанной для решения таких проблем, как жесткая привязка объектов к их клиентам и необходимость использования одного и того же языка программирования. Объекты Workplace Shell работают в среде SOM, доступ в которую можно реализовать почти на всех языках программирования, где предусмотрены внешние процедуры, в том числе и на языке REXX.

В отличие от интерфейса GUI в Windows, где ярлыки (shortcuts) объектов никак не связаны между собой, в WPS объекты, имеющие аналогичные ярлыки (shadow[1]

[1] Shadow (по-английски «тень») — значок на рабочем столе OS/2, который является частью объекта, то есть имеется постоянная двухсторонняя связь между этим значком и собственно объектом.

в терминологии WPS), просто имеют дополнительное свойство — возможность многократно отображаться почти как самостоятельные объекты. Можно сделать несколько таких ярлыков из уже существующего ярлыка или из объекта. При этом любые ярлыки могут перемещаться в любое место и при этом их связи с основным объектом не теряются. Вроде бы то же самое происходит в GUI, но в WPS можно переместить основной объект, и его ярлыки тут же изменят свои параметры, тогда как в GUI произойдет разрушение связей, поскольку связи являются односторонними.

Про SOM можно сказать, что это не связанная ни с одним конкретным языком объектно-ориентированная технология для создания, хранения и использования двоичных библиотек классов. Ключевые слова здесь «двоичные» и «не связанная ни с одним конкретным языком». Хотя нынче многие считают OS/2 технологией прошлого, модель SOM на самом деле представляет собой одну из наиболее интересных разработок в области компьютерной индустрии даже на сегодняшний день. По существу, некоторые идеи, реализованные в OS/2 в начале 90-х годов прошлого столетия, сейчас только обещают реализовать в новом поколении операционных систем Windows с кодовым названием Whistler. Объектно-ориентированное программирование (ООП) заслужило безоговорочное признание в качестве основной парадигмы, однако его применению в коммерческом программном обеспечении препятствуют отсутствие в языках ООП средств обращения к библиотекам классов, подготовленным на других языках, и необходимость поставлять с библиотеками классов исходные тексты. Многим независимым разработчикам библиотек классов приходится продавать заказчикам исходные тексты, поскольку разные компиляторы по-разному отображают объекты. Настоящий потенциал модели SOM заключается в ее совместимости практически с любой платформой и любым языком программирования. Технология SOM соответствует спецификации CORBA (Common Object Request Broker Architecture — общая архитектура посредника объектных запросов), которая определяет стандарт условий взаимодействия между прикладными программами в неоднородной сети.

Графический интерфейс в системах OS/2 не единственный. Интересно отметить тот факт, что существует довольно много альтернативных оболочек для операционных систем OS/2, начиная с программы FileBar, которая хотя и кажется примитивной, но зато отлично работает на компьютерах с оперативной памятью объемом 4 Мбайт, и кончая мощной системой Object Desktop, которая значительно улучшает внешний вид экрана OS/2 и делает работу более удобной.

Помимо оболочек, улучшающих интерфейс операционной системы OS/2, имеется также ряд программ, расширяющих ее функциональность. В первую очередь, это Xfree86 for OS/2 — полноценная система X-Window, которая может использоваться как графический терминал при работе в сети с UNIX-машинами, а также для запуска программ, перенесенных из UNIX в OS/2. К сожалению, таких программ немного, однако большое количество UNIX-программ поставляется вместе с исходными кодами, которые, как правило, практически не нужно изменять для перекомпиляции под Xfree86/OS2.

Серверная операционная система OS/2 Warp 4.5

Серверная операционная система компании IBM, предназначенная для работы на персональных компьютерах и вышедшая в свет в 1999 году, носит название OS/2 WarpServer for e-Business, что подчеркивает ее основное назначение. Однако в процессе ее создания система носила кодовое название Аврора (Aurora), поэтому все ее так теперь и называют.

Как известно, предыдущие версии системы OS/2 могли предоставить программисту только 512 Мбайт виртуального адресного пространства для «родных» 32-разрядных приложений. В свое время это было очень много. Однако хотя задачи, требующие столь большого объема оперативной памяти, встречаются пока еще редко, некоторые считают ограничение в 512 Мбайт серьезным недостатком. Поэтому в последней версии системы это ограничение снято (напомним, что в операционной системе Windows NT 4.0 объем виртуального адресного пространства для задач пользователя составляет 2 Гбайт), и теперь максимальный объем виртуальной памяти для задачи в операционной системе OS/2 v. 4.5 по умолчанию составляет 2 Гбайт, но командой `VIRTUALADDRESSLIMIT=3072` в конфигурационном файле `CONFIG.SYS` он может быть увеличен до 3 Гбайт.

В операционной системе OS/2 v. 4.5 разработчики постарались все «остатки» старого 16-разрядого кода, который еще частично оставался в предыдущих версиях системы, полностью заменить 32-разрядными реализациями, что повысило быстродействие системы. Прежде всего, обеспечена поддержка 32-раздядных драйверов устанавливаемых файловых систем (IFS), ибо в предыдущих системах работа с ними велась через трансляцию вызовов 32bit→16bit→32bit. В то же время для совместимости со старым программным обеспечением помимо 32-раздядного используется и 16-раздядный интерфейс API.

Создана новая файловая система JFS (Journaling File System — файловая система с протоколированием), призванная повысить надежность и живучесть файловой подсистемы по сравнению с файловой системой HPFS386.IFS. Файловая система JFS обеспечивает большую безопасность в структурах данных благодаря технике, разработанной для систем управления базами данных. Работа с JFS происходит в режиме транзакций с ведением журнала транзакций. В случае системных сбоев есть возможность обработки журнала транзакций с целью принятия или отмены изменений, произведенных во время системного сбоя. Эта система управления файлами также повышает скорость восстановления файловой системы после сбоя. Сохраняя целостность файловой системы, она, подобно файловой системе NTFS, не гарантирует восстановление пользовательских данных. Следует отметить, что файловая система JFS обеспечивает самую высокую скорость работы с файлами из всех известных систем, созданных для персональных компьютеров, что очень важно для серверной операционной системы.

Для работы с дисками создан специальный менеджер дисков — LVM (Logical Volume Manager — менеджер логических дисков). LVM хранит информацию обо всех устанавливаемых файловых системах и определяет имена дисков для программ, которые этого требуют. Это позволяет избирательно назначить любую букву любому разделу диска, что в ряде случаев можно считать удобным. И даже больше — теперь

операционной системе более не нужно использовать имена дисков. Менеджер логических дисков в совокупности с файловой системой JFS позволяет объединять несколько томов и даже несколько физических дисков в один большой логический том.

Контрольные вопросы и задачи

1. Изложите основные архитектурные особенности операционных систем семейства UNIX. Попробуйте объяснить основные различия между системами UNIX и Windows.
2. Перечислите и поясните основные понятия, относящиеся к UNIX-системам.
3. Что делает системный вызов fork()? Каким образом осуществляется в операционных системах семейства UNIX запуск новой задачи?
4. Изложите основные моменты, связанные с защитой файлов в UNIX.
5. Сравните разрешения NTFS, имеющиеся в Windows NT/2000/XP, с правами на доступ к файлам, реализованные в UNIX-системах.
6. Расскажите об особенностях семафоров в UNIX. Почему семафорные операции осуществляются сразу над множеством семафоров?
7. Что представляет собой вызов удаленной процедуры (RPC)?
8. Найдите в Интернете описание лицензии GNU и изучите его основные положения. Изложите их. Перечислите сильные и слабые стороны программного обеспечения с открытым исходным кодом.
9. Расскажите об операционной системе Linux. Какие проблемы, на ваш взгляд, наиболее важны для Linux? Расскажите об основных различиях между Linux и FreeBSD.
10. Что представляет собой X-Window? Что такое оконный менеджер? Какие оконные менеджеры для операционной системы Linux вы знаете?
11. Что представляет собой операционная система QNX? Перечислите ее основные особенности.
12. Почему про QNX часто говорят, что это «сетевая» операционная система? Что такое сетевой протокол FLEET?
13. Какие функции реализует ядро QNX?
14. В чем вы видите принципиальные различия между ядром Windows NT 4.0, которое считают построенным по микроядерным принципам, и ядром QNX?
15. Расскажите об основных механизмах взаимодействия для организации распределенных вычислений в операционной системе QNX.
16. Расскажите о проекте OS/2. Какие особенности архитектуры этой операционной системы представляются наиболее интересными?
17. Какие механизмы использует операционная система OS/2, чтобы уменьшить потребности в оперативной памяти и повысить производительность системы?

Глава 11. Операционные системы Windows

Как известно, компания Microsoft является безусловным лидером в разработке программного обеспечения для персональных компьютеров. Среди разнообразных программных продуктов этой компании особое место занимают ее операционные системы. Начав с разработки простейшей однопрограммной операционной системы для первого персонального компьютера, эта компания недавно выпустила несколько версий серверной операционной системы Windows 2003, которые предназначены для построения корпоративных сетей и считаются на сегодняшний день одними из самых сложных и полнофункциональных. Для встроенных систем (в том числе систем для карманных компьютеров и других мобильных систем) Microsoft разработала операционные системы семейства Windows CE. Последняя такая операционная система для популярных компьютеров типа Pocket PC получила название Microsoft Windows Mobile 2003 for Pocket PC. (Операционные системы Windows CE имеют тот же интерфейс Win32 API, что и системы для персональных компьютеров.)

Впервые слово «Windows», что, как известно, в переводе с английского дословно означает *окна*, компания Microsoft использовала в названии своей программной системы для персональных компьютеров, призванной предоставить пользователям графический интерфейс и возможность работать с несколькими приложениями. Первые системы Windows представляли собой своеобразную оболочку, запускаемую из операционной системы MS DOS, которая переключала центральный процессор в защищенный режим работы (см. главу 4) и позволяла организовать параллельное выполнение нескольких задач. Но главным на тот момент было предоставление пользователям графического интерфейса, которым в те времена обладали пользователи компьютеров фирмы Apple. Вначале возможность работать на персональном компьютере в графическом режиме вместо текстового некоторым не казалась такой уж актуальной, хотя, конечно же, всем было понятно, что графический режим богаче по своему потенциалу. Наличие графического интерфейса пользователя (Graphical User Interface, GUI) и широкая поддержка его со стороны компании Microsoft привели к тому, что большинство новых программных продуктов стали создаваться в расчете на эти новые возможности. Со време-

нем компания Microsoft все больше внимания стала уделять обеспечению надежности вычислений и их эффективности, однако задача обеспечить пользователя интуитивно понятным и в целом удобным графическим интерфейсом, похоже, так и осталась главной.

Общим для операционных систем, имеющих в своем названии слово «Windows», является графический интерфейс пользователя. Все эти операционные системы похожи друг на друга. Приложения, написанные для среды Windows, будут одинаково выглядеть и в Windows 95, и в Windows XP. В результате пользователи, умеющие работать с одной операционной системой, достаточно легко могут освоить другую. И это одно из важнейших достоинств.

Основной особенностью систем Windows является то, что все они *предназначены для диалогового режима работы*, и поэтому в качестве основного интерфейса выбран графический, как более функциональный и удобный. Если в таких операционных системах, как Linux, QNX или OS/2, можно работать с системой через интерфейс командной строки и этим ограничиться, то во всех системах Windows невозможно получить текстовый интерфейс командной строки без графического.

Многие считают, что интерфейс командной строки нужен только для относительно редкого вмешательства в работу операционной системы. Однако это не совсем так. Дело в том, что посредством *скриптов* можно автоматизировать выполнение большинства функций, связанных с управлением вычислительными процессами. Скрипт — это текстовый файл, содержащий программу действий, составленную на соответствующем языке[1]. Например, пакетные (batch) файлы в операционных системах от компании Microsoft, которые имеют расширение `bat`, обрабатываются командным интерпретатором `COMMAND.COM`, если речь идет о сеансах DOS, или командным процессором `CMD.EXE`, если речь идет о системах типа Windows NT/2000/XP и в скрипте имеются соответствующие команды. При запуске программы `CMD.EXE` открывается сеанс обычного защищенного 32-разрядного режима.

В ряде случаев графический режим не нужен, поскольку выполняющиеся вычисления не требуют диалога с пользователем. К таким случаям, прежде всего, можно отнести работу серверов, которые, будучи правильно и разумно сконфигурированы, способны работать месяцами без какого-либо вмешательства человека и полностью выполнять поставленные перед ними задачи. К таким случаям можно отнести и задачи автоматизированного управления различными технологическими процессами, специальным автоматизированным оборудованием. А поскольку в этих случаях графический диалоговый режим работы с системой не нужен, не нужны операционной системе и соответствующие вычислительные ресурсы, необходимые для функционирования этого режима. Если же вдруг потребуется организовать диалоговое взаимодействие с операционной системой, то тот же графический режим может быть запущен непосредственно из командной строки, что и делается в уже упомянутых операционных системах семейства UNIX (Linux, FreeBSD и т. д.), QNX, OS/2.

---

[1] Язык для составления программ, которые имеют текстовую форму даже на момент своего исполнения и состоят из команд, понятных операционной системе, часто называют языком скриптов.

Операционные системы Windows 9x

Краткая историческая справка

В те годы, когда появилась первая система Windows, а это произошло в ноябре 1985 года, наибольшее распространение имели компьютеры на базе процессора i80286. Этот процессор хотя и имел средства для организации мультизадачного режима работы (в нем компания Intel впервые реализовала защищенный режим работы, поддержку виртуальной памяти с сегментным механизмом, кольца защиты со шлюзованием для доступа к сегментам кода и многое другое), но аппаратная поддержка была слишком слаба и несовершенна. Только с широким распространением 32-разрядных процессоров (i80386 и все последующие) появилась возможность со временем отказаться в системах Windows от поддержки 16-разрядного защищенного режима работы процессоров и в качестве основного выбрать полноценный 32-разрядный защищенный режим. Как известно, микропроцессор i80386 появился в том же 1985 году. Возможности этого микропроцессора, заложенные в него специально для организации полноценных мультизадачных операционных систем, мы рассмотрели в главе 4. Однако первые несколько лет этот микропроцессор использовался просто как более быстродействующий 16-разрядный микропроцессор i8086 или i80286 (благо он поддерживал такую возможность), поскольку для него долгое время не существовало полноценной 32-разрядной операционной системы.

После первой системы Windows, которая себя только обозначила, компания Microsoft в течение нескольких лет принимала активное участие в работах по созданию операционной системы OS/2. Кстати, операционная система Windows NT «выросла» из проекта OS/2, который имел версию 3.0. Однако проблемы во взаимоотношениях между этими фирмами и желание стать первыми в создании новых мультизадачных операционных систем, имеющих графический интерфейс, привели к тому, что компания Microsoft продолжила работу над Windows вопреки существовавшей договоренности, а в последующем даже разорвала отношения с IBM. Было выпущено несколько версий Windows, пока наконец в 1990 году вышла одна из самых популярных систем того времени — система Windows 3.0. Это была операционная система, предназначенная для работы на процессорах i80386, однако прикладные программы, которые могли выполняться под ее управлением, рассчитывались на интерфейс Win16 API. Само собой, обеспечивалось выполнение DOS-приложений, которые на тот момент доминировали. Для своего запуска эта операционная система требовала наличия среды MS DOS. При запуске программы `WIN.COM` последняя переводила микропроцессор в защищенный режим работы и начинала загружать ядро Windows. Часть драйверов заменялась новыми, а часть могла остаться от MS DOS. После загрузки Windows 3.0 на компьютере можно было параллельно выполнять несколько приложений.

После системы Windows 3.0 появилась система Windows 3.1 и, наконец, сетевая операционная система Windows 3.11 for Workgroups. Все эти операционные системы, хотя и были популярны, имели определенные недостатки, в частности, нельзя было задействовать более 16 Мбайт памяти, не обеспечивались должные надежность и производительность, поскольку не использовались все те возможности 32-раз-

рядного защищенного режима работы микропроцессора, которые этот режим предоставлял.

Первой операционной системой от компании Microsoft, которая должна была исправить существовавшее положение вещей, была Chicago. Она вышла в свет в августе 1995 года и получила известное всему миру название Windows 95. Операционная система Windows 95, по сути дела, произвела революцию в персональных компьютерах: И это несмотря на появившуюся еще в 1992 г. великолепную 32-разрядную операционную систему OS/2 версии 2.0, которая к 1995 году «доросла» уже до версии 2.2 и имела существенно более совершенный графический интерфейс. К сожалению, большинство пользователей почти ничего не знали о существовании этой полноценной, надежной и эффективной операционной системы, поскольку фирма IBM мало беспокоилась об этом. Да и предыдущая (первая) версия операционной системы с тем же названием OS/2 себя не зарекомендовала. В противоположность той позиции, которую заняла IBM, компания Microsoft задолго до появления своей операционной системы Windows 95 серьезно занималась ее продвижением. Пользователи хорошо знали возможности 16-разрядной системы Windows 3.x, они умели с ней работать. Правильно организованная рекламная кампания успешно делала свое дело, и пользователи ждали новую 32-разрядную операционную систему с нетерпением.

Появление операционной системы Windows 95 сопровождалось выходом большого числа соответствующего прикладного программного обеспечения и, что немаловажно, выпуском так необходимых всем книг, в которых излагались принципы работы с новой операционной системой и создания для нее прикладных программ. Многие фирмы, занимающиеся разработкой программного обеспечения, стали выпускать для Windows 95 различные пакеты прикладных программ, системы управления базами данных, системы программирования и другие программы. Все это вместе взятое стало мощнейшим стимулом, обеспечившим победу операционной системе Windows 95, несмотря на то что по своей архитектуре и возможностям она почти по всем параметрам уступала операционной системе OS/2. Операционная система Linux в те годы еще только начинала о себе заявлять.

Затем в 1996 году вышла вторая редакция операционной системы Windows 95 (это был проект Nashville), которая получила название Windows 95 OSR 2 (OEM Service Release 2). В этой операционной системе были улучшены система управления файлами (введена поддержка файловой системы FAT32), а также средства для работы с мультимедиа и Интернетом.

Далее в 1998 году компания Microsoft обновила свою операционную систему еще раз, дав ей имя Windows 98. Эта операционная система имела еще больше именно 32-разрядного собственного кода, обладала большей стабильностью и производительностью, поскольку был устранен почти весь прежний 16-разрядный код, выполнявшийся достаточно часто и имевший все характерные для него недостатки. В частности, была введена новая модель 32-разрядных драйверов WDM (Windows Driver Model)[1],

[1] Новая модель многоуровневой организации драйверов для систем Windows, которая пришла на смену прежней.

которая позволяет использовать драйверы, создаваемые для операционных систем семейства Windows NT. Важной для успеха этой операционной системы была также полноценная поддержка интерфейса USB (Universal Serial Bus — универсальная последовательная шина). Обнаруженные в системе ошибки были исправлены во второй редакции этой операционной системы. Нынче операционная система Windows 98 SE[1] является одной из самых распространенных в мире.

Наконец, в канун начала нынешнего тысячелетия[2] Microsoft выпустила свою последнюю версию операционной системы, которая также была основана на архитектуре системы Windows 95. Это была Windows Millennium Edition[3] (ME). Выпуская Windows ME, Microsoft преследовала несколько первоочередных целей: превратить потребительскую операционную систему в полноценную мультимедийную (не только игровую) платформу; максимально упростить обслуживание системы; обеспечить удобные средства создания домашних сетей; обеспечить доступ ко всему богатству ресурсов Интернета. Основным, принципиальным моментом (и основным недостатком, как это ни покажется странным) был отказ от поддержки сеансов DOS, что позволяет потенциально немного увеличить надежность организуемых вычислений. Операционная система Windows ME была предназначена для использования на домашних компьютерах, и это обстоятельство, вкупе с невозможностью организовать выполнение программ, требующих открытия сеансов DOS, не позволило ей получить широкое распространение. Тем более что вскоре для потребительских целей Microsoft стала продавать новую операционную систему Windows XP Home Edition, которая уже относится к системам типа Windows NT.

Общие сведения

Операционные системы Windows 9x создавались для работы только на IBM-совместимых персональных компьютерах. Они не являются переносимыми и на других платформах (на процессорах, не совместимых с архитектурой ia32) не работают. Как и для всего остального программного обеспечения от Microsoft, исходные коды операционных систем закрыты, поэтому подробного описания ее архитектуры практически нет; имеются только многочисленные публикации о том, как следует использовать эти системы.

Операционные системы семейства Windows 9x предназначены, главным образом, для домашнего, а не корпоративного применения. Уже многие годы они являются самыми распространенными в мире. Хотя они допускают возможность работы с компьютером нескольких пользователей (естественно, по очереди, поскольку системы являются однотерминальными), в них не поддерживается механизм учетных записей, как в остальных 32-разрядных операционных системах. Каждый пользователь может иметь свое собственное рабочее окружение, то есть свои вид *рабочего стола* (desktop), состав *панели задач* (taskbar) и меню Пуск (Start), параметры настройки используемых программ и многое другое. Это собственное рабо-

[1] SE означает «Second Edition» — вторая редакция.

[2] Эта система вышла осенью 2000 года.

[3] Слово «millennium» как раз и означает канун тысячелетия.

чее окружение называется *профилем* (profile), и при включении такой возможности в системном каталоге образуется вложенный каталог[1] с именем Profiles, в котором и размещаются профили пользователей. Независимо от того, имеет каждый пользователь свой профиль или не имеет, он должен зарегистрироваться, если система сконфигурирована для работы в вычислительной сети. Для выполнения *процедуры аутентификации* используются файлы с расширением pwl, к которым, к сожалению, имеется свободный доступ, в результате узнать пароль того или иного пользователя не составляет большой проблемы для злоумышленника.

Для облегчения работы с компьютером все эти системы поддерживают механизм автоматического обнаружения подключенных к нему устройств (так называемый механизм *Plug and Play* — «включай и работай»). Эту задачу выполняет специальный модуль — диспетчер конфигурации (Configuration Manager). Он гарантирует, что каждое устройство, входящее в состав персонального компьютера, сможет использовать линии IRQ (Interrupt Request — запрос на прерывание), адреса портов ввода-вывода, каналы прямого доступа к памяти и прочие ресурсы без конфликтов с другими устройствами. Кроме того, диспетчер конфигурации отслеживает текущие изменения в конфигурации компьютера. Поскольку операционные системы Windows 9x получили самое широкое распространение, все выпускаемое периферийное оборудование имеет необходимые драйверы, причем достаточно правильно написанные и успешно работающие. Поэтому серьезные проблемы на сегодняшний день с этим механизмом мало кто испытывает. Динамическое конфигурирование аппаратно-программной среды значительно упрощает использование операционной системы и позволяет без лишних операций ручной настройки работать на компьютере пользователям, не являющимся специалистами в вычислительной технике.

С точки зрения базовой архитектуры операционные системы семейства Windows 9x являются 32-разрядными и мультизадачными (многопоточными) системами с вытесняющей многозадачностью. Ядра у всех этих операционных систем построены по макроядерной архитектуре. Ядро состоит из трех основных компонентов: *Kernel*, *User* и *GDI*. Модуль Kernel обеспечивает основную функциональность операционной системы, в том числе: планирование процессов; поддержку потоков выполнения; синхронизацию объектов; работу с файлами, отображаемыми на память; управление памятью; файловый ввод-вывод; обработку исключений; работу консолей; взаимодействие 32-разрядного и 16-разрядного кода с преобразованием 16-разрядного формата кода и данных в 32-разрядный (и наоборот) посредством механизма шлюзования; некоторые другие функции. Компонент User управляет вводом с клавиатуры и координатных устройств (типа мыши) и выводом через пользовательский интерфейс. Когда то или иное устройство ввода генерирует прерывания, обработчик прерываний, используя модель асинхронного ввода, преобразует их в сообщения и посылает потоку необработанного ввода, который распределяет их по соответствующим очередям сообщений. Наконец, компонент ядра, называемый GDI (Graphical Device Interface — графический интерфейс устрой-

[1] Вместо термина *каталог* (directory) в системах с графическим интерфейсом гораздо чаще используют термин *папка* (folder).

ства), представляет собой графическую подсистему, которая отвечает за прорисовку графических примитивов, операции с растровыми изображениями и взаимодействие с аппаратно-независимыми графическими драйверами. GDI управляет выводом на экран, принтеры и другие устройства.

Все операционные системы Windows 9x централизованно хранят всю системную информацию об аппаратных средствах, установленном системном и прикладном программном обеспечении и его настройке, в том числе и индивидуальных параметрах каждого пользователя. Такая централизованная информационная база данных называется *реестром* (registry). Реестр избавляет от необходимости иметь дело с множеством INI-файлов, как это было в системах Windows 3.x. Физически содержимое реестра определяется файлами system.dat и user.dat, которые располагаются в каталоге с файлами операционной системы. В режиме, когда каждый пользователь имеет собственный профиль, определяющий персональную настройку его рабочего окружения, в состав реестра включается еще файл user.dat того пользователя, который в этот момент работает на компьютере. Файлы с именем user.dat располагаются в профилях пользователей и определяют права пользователей в операционной системе.

В операционных системах Windows 9x для работы с периферийными устройствами используется архитектура *универсальный драйвер—мини-драйвер*. Она позволяет упростить разработку драйверов для создателей нового оборудования. Операционные системы Windows 9x сами предоставляют базовые услуги для различных классов аппаратных устройств. Для этого существуют универсальные драйверы, которые включают большую часть кода, необходимого конкретному классу устройств для взаимодействия с компонентами операционной системы. Поэтому изготовителям оборудования необходимо написать относительно небольшой код минидрайвера, который должен содержать какие-либо дополнительные функции, нужные для управления конкретным устройством и учитывающие именно его специфику. Во многих случаях универсальные драйверы реализуют практически все функции, которые необходимы для управления операциями ввода-вывода при обмене данными с периферийным устройством, и иметь дополнительный минидрайвер не требуется.

Помимо этих драйверов, которые относятся к драйверам низкого уровня и непосредственно завязаны на аппаратуру, в Windows 9x используются *драйверы виртуальных устройств*. Эти драйверы предназначены для управления системными ресурсами, причем они позволяют разделять ресурс между несколькими процессами. Аббревиатура VxD (Virtual Device — виртуально устройство), которую мы можем встретить при детальном знакомстве с этими операционными системами, означает, что речь идет именно о драйверах виртуальных устройств. Вместо средней буквы x в названии драйвера виртуального устройства может стоять, например, латинская буква P, которая означает, что речь идет о драйвере принтера. Если же название виртуального драйвера — VDD, то мы имеем дело с драйвером дисплея.

Драйверы VxD поддерживают все основные устройства персонального компьютера, включая контроллеры на системной плате, контроллеры дисковых устройств, таймер, видеоконтроллеры, коммуникационные порты (параллельный и последовательный), принтеры, клавиатуры и многие другие. Они обеспечивают динами-

ческую поддержку драйверов устройств, а виртуальное устройство отслеживает состояние соответствующего реального аппаратного устройства для любого процесса, которое им используется. Поскольку системы Windows 9x обеспечивают мультизадачный режим, передача устройства от одного процесса другому происходит очень часто. Каждое выполняемое приложение или системный процесс может прервать работу с устройством другого приложения. Поскольку такое вмешательство в принципе могло бы вызвать полный крах процессов управления вводом-выводом, драйвер виртуального устройства проверяет и соответственно изменяет состояние устройства для любого приложения и/или системного процесса ввода-вывода. При этом, естественно, гарантируется, что устройство будет корректно функционировать с каждым из процессов, запрашивающим ту или иную операцию ввода-вывода на этом устройстве. Некоторые драйверы виртуальных устройств предназначены для управления программными компонентами операционной системы; они содержат код, который эмулирует определенные программные средства или отслеживает, чтобы выполняющиеся процессы использовали только свои данные. Во всех операционных системах Windows 9x в память загружаются только те драйверы виртуальных устройств, которые необходимы в данный момент. Это позволяет экономить оперативную память компьютера.

Одним из драйверов виртуальных устройств является системный драйвер, управляющий файловой системой защищенного режима и драйверами блочных устройств. Это супервизор ввода-вывода (Input/Output Supervisor, IOS). Он принимает запросы от файловых систем и загружает драйверы, обеспечивающие доступ к локальным дискам и дисковым устройствам.

Драйверы файловых систем являются компонентами кода с нулевым уровнем привилегий. Они поддерживают следующие файловые системы:

- VFAT (Virtual FAT) — файловые операции на дисковых устройствах и взаимодействие с подсистемой блочного ввода-вывода;
- CDFS — работа с компакт-дисками;
- UDF (Universal Disk Format) — соответствует спецификациям, принятым организацией Optical Storage Technology Association, и предназначена для доступа к дискам DVD-ROM и CD-ROM (эта файловая система не поддерживается в Windows 95);
- сетевые редиректоры для обеспечения связи с серверами компаний Microsoft и Novell (Netware).

Все эти файловые системы управляются диспетчером устанавливаемых файловых систем (IFS). Помимо перечисленных в операционную систему можно установить и иные файловые системы. Например, при работе с Windows 98 мы можем установить драйвер для доступа к дискам NTFS. Правда, реализация этого драйвера такова, что он игнорирует все расширенные атрибуты. В результате не работают разрешения NTFS для ограничения на доступ к файлам, которые и составляют одно из основных достоинств этой файловой системы.

По умолчанию системы Windows 98 и Windows ME позволяют работать с файловой системой FAT12 (для работы с дискетами), FAT16 и FAT32. Последняя является основной для этих операционных систем.

Операционные системы Windows являются сетевыми. В дистрибутивы входит все необходимое системное сетевое программное обеспечение, которое легко и быстро устанавливается и конфигурируется. Используется программный интерфейс NetBIOS и технология SMB (Server Message Blocks). Системы главным образом предназначены для работы в составе рабочих групп, то есть для построения одноранговых вычислительных сетей, хотя операционные системы Windows 95 и Windows 98 допускают работу в составе домена в сетях клиент-сервер. Для этого они имеют все необходимые программные модули и интерфейсные экранные формы. Однако, поскольку в случае их использования в корпоративных сетях существенным образом начинает страдать информационная безопасность, делать это не рекомендуется.

Организация многозадачности

Одним из наиболее актуальных вопросов, которые решает любая многозадачная операционная система, в том числе и системы Windows 9x, состоит в организации по возможности простого, но эффективного способа предоставления процессорного времени различным параллельно выполняющимся программам. Другими словами, речь идет о диспетчеризации задач.

Мы уже знаем, что *многозадачность*, в общем случае, означает способность операционной системы обеспечивать совместное использование процессора несколькими программами. Большинство разработчиков операционных систем называют работающие программы *задачами*, поэтому задачей можно считать загруженную в память программу, которая что-то делает. В большинстве операционных систем, в том числе и в Windows NT, и в UNIX, выполнение приложения называется *процессом*. Однако в уже упомянутых операционных системах Windows 3.x почти всегда использовался термин «задача», и лишь изредка — «процесс». Имейте в виду, что в операционных системах Windows 9x используется исключительно термин «процесс», а понятие задачи было официально исключено из терминологии Windows. Вкладывая совершенно такой же смысл в слово «процесс», разработчики Microsoft тем самым попытались поставить операционные системы семейства Windows 9x как бы на один уровень с другими операционными системами, такими, например, как Windows NT. В большей части документации по Windows 3.1 мы можем обнаружить оба упомянутых слова. Основная причина изменения терминологии — реализация мультизадачности при сохранении мультипрограммного режима работы. Другими словами, речь идет о поддержке в этих операционных системах возможности *многопоточного* выполнения приложений. Поэтому помимо отхода от термина *задача* и использования термина *процесс*, мы должны отметить, что во всех этих операционных системах стал использоваться термин *поток выполнения*, или *тред* (thread).

Напомним, что поток выполнения — это одна из ветвей вычислительного процесса. Потоку выделяется процессорное время, этим занимается диспетчер задач операционной системы, называемый *планировщиком*. Поток может быть создан любым работающим под управлением Windows 9x 32-разрядным приложением или виртуальным драйвером устройства. Поток имеет собственный стек и контекст

выполнения (а именно содержимое рабочих регистров процессора). Потоки используют память совместно с процессом-родителем. Когда Windows 95/98 загружает приложение и создает необходимые ему структуры данных, система настраивает процесс в виде отдельного потока. Потоки могут использовать весь код и глобальные данные процесса-родителя. Это означает, что создание нового потока требует минимальных затрат памяти. Один процесс может породить множество параллельно выполняющихся потоков. Многие приложения на протяжении всего времени своей работы используют единственный поток, хотя могут (а многие так и делают) использовать еще несколько потоков для выполнения определенных кратковременных операций в фоновом режиме, что позволяет либо увеличить скорость выполнения приложений, либо дать возможность пользователю выполнять следующую операцию в своей программе, не дожидаясь завершения текущей операции.

Термин «задача» мы все же будем использовать и дальше, поскольку с точки зрения распределения процессорного времени требуется все то же: выполнять вычисления, то есть выполнять некий конкретный код с конкретными данными.

В Windows 95/98 работа с потоками доступна только 32-разрядным приложениям и виртуальным драйверам устройств. Виртуальные машины MS DOS и старые 16-разрядные приложения Windows не могут обращаться к функциям API, которые поддерживают потоки. Каждая виртуальная машина MS DOS работает в отдельном потоке. Аналогично, каждое 16-разрядное приложения Windows при своем исполнении образует процесс, который использует всего один поток, что позволяет обеспечить для старых приложений Windows модель кооперативной многозадачности. Любое 32-разрядное приложение или виртуальный драйвер устройства может создавать дополнительные потоки, а Windows 95/98 может организовать диспетчеризацию всех этих потоков в соответствии с алгоритмами вытеснения, что представляет собой еще один аспект многозадачности в Windows. Несмотря на то что все эти потоки могут представлять принципиально разные типы программ, в системе они представлены в виде одинаковых структур данных. Вследствие этого диспетчер и остальной 32-разрядный системный код, использующий эти внутренние структуры данных, и удалось реализовать так, что разработчикам не понадобилось учитывать особенности преобразований 16-разрядного кода в 32-разрядный.

Как известно, в основе диспетчеризации задач ныне уже почти всеми забытой Windows 3.x лежал принцип невытесняющей, или кооперативной, многозадачности (cooperative multitasking). При работе в среде Windows 95/98 кооперативная многозадачность используется только для диспетчеризации старых 16-разрядных приложений, в то время как работа приложений Win32 планируется в соответствии с иным алгоритмом. Для 32-разрядных приложений система использует вытесняющую многозадачность (preemptive multitasking).

Поддержкой многозадачности занимается планировщик (scheduler). Он имеет дело главным образом с временем и событиями. Процессу в Windows 95/98 выделяется квант времени, который определяет, как долго данный процесс может использовать процессор. По окончании кванта времени планировщик определяет, следует

ли передать процессор в распоряжение другого процесса. В отличие от Windows NT, Windows 95/98 не поддерживает мультипроцессорные системы, в которых планировщик может выделять процессам больше одного процессора.

Решения, принимаемые планировщиком, определяются событиями. Так, щелчок мыши является для планировщика событием. Это событие может привести к передаче процессора в распоряжение процесса, «владеющего» окном, в котором произошел щелчок. Впрочем, планировщик может решить, что завершение передачи данных по сети заслуживает большего внимания, чем щелчок мыши, и тогда процессор будет передан в распоряжение процесса, обслуживающего сеть, а всем остальным процессам придется подождать.

Как мы знаем, при вытесняющем планировании только система может решать, в каком порядке, как долго и какие задачи (в нашем случае — потоки) будут выполняться. Планировщик может в любой момент отнять процессор у одного из потоков выполнения и передать его в распоряжение другого. Обычно такой акт вытеснения происходит в результате реакции на событие, требующее внимания. Планировщик присваивает каждому из работающих процессов приоритет (priority). Потоки его наследуют, хотя при создании нового потока ему можно задать и иной приоритет. В случае если происходит событие, относящееся к потоку, обладающему более высоким приоритетом, планировщик приостанавливает (вытесняет) текущий поток и начинает выполнять тот, у которого приоритет больше.

Для обеспечения гарантированного обслуживания введен механизм динамических приоритетов, при котором приоритеты потоков постоянно пересчитываются. Например, если бы системе надо было выбирать между двумя потоками, у одного из которых приоритет больше, а у другого меньше, то поток, обладающий низким приоритетом, никогда бы не смог выполняться, если бы планировщик динамически не изменял значения приоритетов. При расчете приоритетов также играет роль длительность квантов времени.

Диспетчер задач (потоков выполнения) использует следующие три механизма, с помощью которых он пытается равномерно распределять время процессора между всеми вычислениями в целях обеспечения бесперебойной и одновременно быстрой реакции системы.

- *Динамическое изменение приоритета.* Диспетчер на время может повысить или понизить приоритет того или иного потока. Так, например, нажатие клавиши или щелчок мыши говорит ему о том, что приоритет потока, к которому относится действие пользователя, должен быть повышен.
- *Постсинхронизированное снижение приоритета.* Ранее повышенное значение приоритета постепенно возвращается к исходному значению.
- *Наследование приоритета.* Служит для быстрого повышения приоритета. Обычно это делается для того, чтобы позволить потоку с низким приоритетом быстро закончить работу с выделенным для монопольного использования ресурсом, который необходим потокам с высоким приоритетом. Windows 95/98 восстанавливает исходное значение унаследованного приоритета сразу же после удовлетворения конфликтного условия.

В операционных системах семейства Windows 9x имеется два модуля для диспетчеризации потоков выполнения: *основной планировщик* (primary scheduler) отвечает за вычисление приоритетов потоков; *планировщик квантования* (timeslice scheduler) занимается расчетами, необходимыми для выделения квантов времени. По сути дела, модуль квантования решает, какой процент доступного процессорного времени какому потоку выделить. Если некий поток не получает времени на выполнение, значит, он находится в состоянии *ожидания выполнения* (suspended) и не будет работать, пока ситуация не изменится.

Основной планировщик просматривает все потоки выполнения и рассчитывает для каждого из них *приоритет выполнения* (execution priority), который представляет собой целое число, находящееся в пределах от 0 до 31. Далее он переводит в состояние ожидания выполнения все потоки, приоритет выполнения которых меньше текущего наибольшего значения, имеющегося у одного из потоков. После того как поток переведен в состояние ожидания выполнения, основной планировщик более не обращает на него никакого внимания при дальнейших вычислениях приоритетов на протяжении данного кванта времени. По умолчанию длительность кванта времени составляет 20 мс. Затем планировщик квантования рассчитывает процентную долю кванта времени, которую необходимо выделить каждому потоку. Для этого он использует значения приоритетов и информацию о текущем состоянии виртуальной машины.

После окончания выделенного потоку кванта времени планировщик перемещает его в конец очереди, состоящей из потоков с равным приоритетом. Этот классический механизм диспетчеризации, называемый карусельным, обеспечивает всем потокам, обладающим равным наивысшим приоритетом, одинаковый доступ к процессору. Если некоторый поток не занимает весь выделенный ему квант процессорного времени, диспетчер передает процессор в распоряжение следующего потока с таким же приоритетом и позволяет тому использовать остаток данного кванта времени.

Если в системе одновременно работают только 32-разрядные приложения, то более быструю реакцию системы и значительно менее конфликтный отклик программ пользователю обеспечивает так называемое *упреждающее планирование*. Как известно, в диалоговых системах используется событийное программирование, при котором выполнение той или иной процедуры начинается после определенного события. Сама операционная система также функционирует по этому принципу. Управляющая (супервизорная) подсистема Windows 9x просматривает направляемый ей подсистемой ввода-вывода поток сообщений, из которых она и узнает о новых событиях, таких, например, как щелчки мыши в одном из ее окон, запуск новых программ и т. д. В системах Windows 3.x все сообщения находились в одной системной очереди, вследствие чего одна некорректно работающая программа могла заблокировать поток сообщений, предназначенных всем остальным приложениям. Windows 95/98 дает системе возможность помещать сообщения, предназначенные приложениям Win32, в отдельные очереди, что снижает вероятность зависания системы в тех случаях, когда одно из приложений не обслуживает очередь сообщений должным образом.

Распределение оперативной памяти

Для загрузки операционные системы Windows 95/98 используют операционную систему MS DOS 7.0 (MS DOS 98), и в случае если в секции [Options] файла MSDOS.SYS имеется строка BootGUI = 0, процессор работает в обычном реальном режиме (см. главу 4). Распределение памяти в MS DOS 7.0 такое же, как и в предыдущих версиях DOS. Однако при загрузке интерфейса GUI перед загрузкой ядра Windows 95/98 процессор переключается в защищенный режим работы и начинает распределять память уже с помощью страничного механизма.

Приложения и подсистемы Windows 9x (за исключением ядра) никогда не работают с физической памятью. Разделение на виртуальную и физическую память является ключевым аспектом работы системы. Приложения и подсистемы Windows 9x имеют дело с определенными интерфейсами прикладного программирования и виртуальными адресными пространствами. Базовая система работает как с физической памятью, так и с виртуальными адресными пространствами.

В основе поддержки виртуальных машин и виртуального адресного пространства, которую обеспечивают операционные системы Windows 9x, лежит работа с реальной (физической) памятью компьютера, ограниченной в своих размерах. Операционная система выгружает неактивные страницы памяти виртуальных адресных пространств выполняющихся процессов из оперативной памяти на диск и загружает страницу, запрошенную при выполнении текущей команды. Другими словами, загрузка страницы в оперативную память осуществляется по требованию, как это принято в большинстве операционных систем, использующих страничный механизм организации виртуальной памяти. В то же время, освобождается оперативная память от неактивных страниц группами по нескольку страниц за одну операцию. Реализованный в операционных системах Windows 9x алгоритм замещения представляет собой стандартную дисциплину LRU (Least Recently Used — дольше других неиспользуемый), заключающуюся, как мы уже знаем, в освобождении тех страниц физической памяти, которые дольше других не использовались.

Многие страницы физической памяти компьютера не участвуют в замещении, они распределены постоянно. Их занимают, в частности, резидентные компоненты ядра. На эти цели отводится примерно один мегабайт памяти. За оставшуюся физическую память конкурируют различные программы: динамически загружаемые компоненты системы и загружаемые виртуальные драйверы устройств, код и данные приложений, а также динамически размещаемые данные, такие как области кэширования, необходимые для работы файловой системы, и буферы прямого доступа к памяти (DMA).

В отличие от тех мультитерминальных систем, в которых операционная система должна заботиться о равноправном совместном использовании ресурсов, в системах Windows 9x сделано иначе. Поскольку это однопользовательские операционные системы, они позволяют заполнять память так, как это нужно пользователю и его программам. Динамически загружаемые компоненты системы конкурируют за память с прикладными программами. Если пользователь хочет, чтобы его приложение работало быстрее, ему будет позволено занять столько памяти, сколько вообще возможно. Система накладывает ограничение на максимальный объем па-

мяти, который может быть отдан в распоряжение отдельных приложений, — если не следить за этим, становится возможным возникновение тупиковых ситуаций. После того как вся физическая память заполнена, первый же новый запрос на выделение памяти инициирует замещение страниц. Интересным побочным эффектом такого подхода является то, что у приложений нет надежного способа определения объема памяти, доступного в системе. Функция API GlobalMemorySatus() возвращает целый ряд параметров, характеризующих состояние системной памяти, однако это не более чем «мгновенный снимок» текущей обстановки — еще один вызов этой функции вполне может дать другие значения.

Страницы поступают в память и уходят из нее по-разному: в большинстве случаев они либо непосредственно размещаются в выделенной для этого памяти (как результат соответствующих запросов), либо загружаются при старте программы из EXE-файла приложения. Впоследствии эти страницы начинают перемещаться между физической памятью и файлом подкачки. Страницы, в которых содержится только код 32-разрядных приложений и динамически связываемых библиотек (DLL), система всегда загружает только из исходных исполняемых файлов.

Для того чтобы облегчить управление всем разнообразием типов страниц памяти, каждая активная страница, то есть каждая страница, которая является частью выполняющегося в данный момент системного модуля или приложения, снабжена хранящимся совместно с ней *страничным дескриптором* (Page Descriptor, PD). В этом дескрипторе содержатся адреса процедур, которые занимаются перемещением страницы из памяти на диск и обратно. Независимо от того, что именно находится в данной странице, диспетчер физической памяти, чтобы переместить страницу в оперативную память или из нее, просто вызывает соответствующую функцию, адрес которой определен в поле дескриптора страницы. В случае, если некоторая страница еще никогда не заполнялась, она называется *абсолютно чистой* (virgin"). Например, именно так обозначаются страницы, содержащие код, использующий вызовы Win32. После того как с момента размещения страницы в памяти в нее будет в первый раз произведена запись данных, она считается *испорченной* (tainted) и может быть либо *грязной* (dirty), либо *чистой* (clean), в зависимости от того, осуществлялась ли в нее запись с момента последней ее подкачки в физическую память. Если запись в эту страницу производилась, и в этой физической странице требуется разместить иную виртуальную страницу, ее содержимое должно быть сохранено в файле подкачки.

Для наблюдения за распределением памяти и использованием иных ресурсов компьютера можно воспользоваться, например, программой SYSMON.EXE (системный монитор). Эта программа входит в состав утилит операционных систем Windows 9x, поэтому после ее установки команда для ее запуска располагается в подменю Служебные меню Стандартные. Она позволяет выбрать интересующие нас параметры и наблюдать за их текущими значениями.

Использование так называемой *плоской модели памяти*, когда программист может использовать только один сегмент кода и один сегмент данных, которые имеют максимально возможные размеры, определяемые системными соглашениями операционной системы, приводит к тому, что с точки зрения программиста память получается неструктурированной. Программы используют классическую малую

(small) модель памяти [40]. Каждая прикладная программа определяется 32-разрядными адресами, в которых сегмент кода имеет то же значение, что и сегменты данных. Единственный сегмент программы отображается непосредственно в область виртуального линейного адресного пространства, которая, в свою очередь, состоит из 4-килобайтных страниц. Каждая страница может располагаться где угодно в оперативной памяти (естественно, в том месте, где ее разместит диспетчер памяти, который сам находится в невыгружаемой области) или быть «сброшена» на диск, если не запрещено использовать страничный файл.

В операционных системах Windows 9x младшие адреса виртуального адресного пространства совместно используются всеми процессами. Это сделано для совместимости с драйверами устройств реального режима, резидентными программами и некоторыми 16-разрядными программами Windows. Безусловно, это плохое решение с точки зрения надежности, поскольку оно приводит к тому, что любой процесс может непреднамеренно (или же, наоборот, специально) испортить компоненты, находящиеся в этих адресах.

В Windows 9x каждая 32-разрядная прикладная программа выполняется в собственном адресном пространстве, но все они используют совместно один и тот же 32-разрядный системный код. Доступ к чужим адресным пространствам в принципе возможен. Другими словами, виртуальные адресные пространства не задействуют всех аппаратных средств защиты, заложенных в микропроцессор. В результате неправильно написанная 32-разрядная прикладная программа может привести к аварийному сбою всей системы. Все 16-разрядные прикладные программы Windows разделяют общее адресное пространство, поэтому они так же уязвимы друг для друга, как и в среде Windows 3.X.

Собственно системный код Windows 9x размещается выше границы 2 Гбайт. В пространстве с отметками 2 и 3 Гбайт находятся системные библиотеки DLL, используемые несколькими программами. Напомним, что в 32-разрядных микропроцессорах семейства i80x86 имеется четыре уровня защиты, именуемые кольцами с номерами от 0 до 3. Кольцо с номером 0 является наиболее привилегированным, то есть максимально защищенным. Компоненты операционных систем Windows 9x, относящиеся к кольцу 0, отображаются на виртуальное адресное пространство между 3 и 4 Гбайт. К этим компонентам относятся собственно ядро Windows, подсистема управления виртуальными машинами, модули файловой системы и драйверы виртуальных устройств (VxD).

Область памяти между 2 и 4 Гбайт адресного пространства каждой 32-разрядной прикладной программы совместно используется всеми 32-разрядными прикладными программами. Такая организация позволяет обслуживать вызовы API непосредственно в адресном пространстве прикладной программы и ограничивает размер рабочего множества. Однако за это приходится расплачиваться снижением надежности. Ничто не может помешать программе, содержащей ошибку, произвести запись в адреса, принадлежащие системным библиотекам DLL, и вызвать крах всей системы.

В области между 2 и 3 Гбайт также находятся все запускаемые 16-разрядные прикладные программы Windows. С целью обеспечения совместимости эти програм-

мы выполняются в совместно используемом адресном пространстве, где они могут испортить друг друга так же, как и в Windows 3.x.

Адреса памяти ниже 4 Мбайт также отображаются в адресное пространство каждой прикладной программы и совместно используются всеми процессами. Благодаря этому становится возможной совместимость с существующими драйверами реального режима, которым необходим доступ к этим адресам. Это делает еще одну область памяти не защищенной от случайной записи. К самым нижним адресам (менее 64 Кбайт) этого адресного пространства 32-разрядные прикладные программы обращаться не могут, что дает возможность перехватывать неверные указатели, но 16-разрядные программы, которые, возможно, содержат ошибки, могут записывать туда данные.

Вышеизложенную модель распределения памяти можно проиллюстрировать с помощью рис. 11.1.

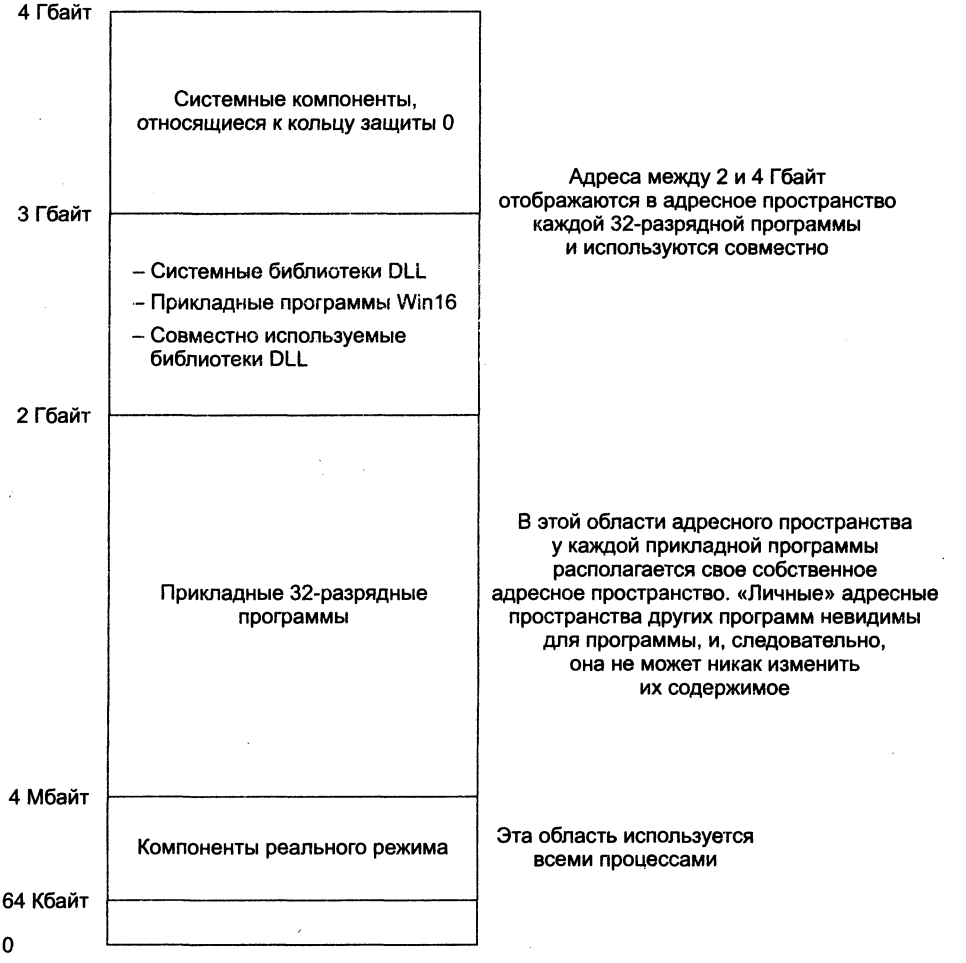


Рис. 11.1. Модель памяти операционных систем Windows 95/98

В операционных системах Windows термином *модуль* (module) называют присутствующую в памяти совокупность кода, данных и других ресурсов (в частности таких, как битовые массивы). Обычно такая совокупность объектов представляет собой отдельную прикладную программу или библиотеку DLL. Windows формирует и поддерживает структуру данных под названием *база данных модулей* (module database), в которой учитываются все активные в данный момент модули системы. База данных модулей описывает статическую совокупность объектов в отличие от той динамической, что поддерживает база данных задач. Учет загруженных в данный момент модулей необходим, потому что он служит основой поддерживаемого Windows 9x механизма совместного использования ресурсов. Так, например, когда мы вторично запускаем программу Word или Internet Explorer, операционная система Windows обнаруживает, что сегменты кода и формирующий значок этой программы — битовый массив — уже загружены, и вместо того чтобы загружать еще одну копию, которая только отнимет память, она попросту заводит дополнительные ссылки на уже используемые ресурсы.

На протяжении всего времени работы системы Windows 9x для каждого ресурса поддерживают счетчик обращений к нему. По мере того как приложения используют тот или иной ресурс, Windows 9x увеличивают значение соответствующего счетчика, а по завершении работы приложения уменьшают его. Значение счетчика, равное нулю, свидетельствует о том, что ресурс больше не используется, а значит, система может удалить ресурс и освободить память, которую он занимал.

Минимально допустимый объем оперативной памяти, начиная с которого эти операционные системы могут функционировать, равен 4 Мбайт для Windows 95 и 8 Мбайт для Windows 98. Однако при таких маленьких объемах физической памяти пробуксовка столь велика, что быстродействие системы становится слишком малым, и практически работать нельзя.

Страничный файл, с помощью которого реализуется механизм виртуальной памяти, по умолчанию располагается в каталоге самой системы Windows и имеет переменный размер. Система отслеживает его длину, увеличивая или сокращая этот файл при необходимости. Вместе с фрагментацией файла подкачки это приводит к тому, что быстродействие системы становится меньше, чем если бы этот файл был фиксированного размера и располагался в смежных кластерах (был бы дефрагментирован). Создать файл подкачки заданного размера можно либо через специально разработанный для этого апплет (**Панель управления ▸ Система ▸ Быстродействие ▸ Файловая система**), либо просто прописав в секции [386Enh] файла **SYSTEM.INI** строки с указанием диска и имени файла подкачки, например:

```
PagingDrive=C:
PagingFile=C:\PageFile.sys
MinPagingFileSize=65536
MaxPagingFileSize=262144
```

Здесь первая и вторая строки описывают размещение страничного файла и его имя, а две последних — начальный и предельный размеры страничного файла (значения указываются в килобайтах). Для определения необходимого минимального размера можно рекомендовать запустить уже упомянутую выше программу

SYSMON.EXE[1] (системный монитор) и, выбрав в качестве наблюдаемых параметров размер файла подкачки и объем свободной памяти, оценить потребности в памяти, запуская те приложения, с которыми чаще всего приходится работать.

Большое влияние на использование оперативной памяти и общую производительность системы оказывает драйвер виртуального устройства VCache, занимающийся кэшированием файлов. Он взаимодействует с менеджером физической памяти, запрашивая и освобождая области памяти, которые впоследствии могут быть выделены отдельным драйверам файловой системы для выполнения операций кэширования. Этот драйвер работает по методу агрессивного кэширования, в результате он может захватывать почти всю свободную оперативную память. Как ни странно, это не всегда приводит к увеличению скорости работы с файлами, поскольку поиск нужных блоков данных среди блоков, находящихся в кэше, осуществляется простым последовательным перебором, а количество просматриваемых блоков в этом случае существенно больше. Поэтому в ряде случаев имеет смысл ограничивать «аппетит» драйвера VCache. Сделать это можно путем редактирования все того же файла SYSTEM.INI. Только теперь нужно найти другую секцию файла — [VCache]. В эту секцию следует добавить строки и прописать значения для максимального и минимально объемов оперативной памяти, которую операционная система будет предоставлять подсистеме кэширования. Выглядеть эти новые строки могут, например, так:

```
MinFileCache=16384
MaxFileCache=65536
ChunkSize=2048
NameCache=4096
DirectoryCache=128
```

Назначение первой и второй строк представляется очевидным. Третья строка описывает размер блока, четвертая строка — количество хранимых в кэше имен файлов, а последняя — количество каталогов. Прописанные в приведенных строках значения, естественно, зависят от объема оперативной памяти, имеющейся в компьютере. В данном случае компьютер имеет 512 Мбайт оперативной памяти. Кстати, если персональный компьютер имеет более 256 Мбайт памяти, то наличие первых двух строк в секции [VCache] файла SYSTEM.INI обязательно. В противном случае из-за недальновидности разработчиков драйвера виртуального устройства VCache он может запросить у системы более 256 Мбайт памяти, причем она может выделить ему эту память. Это неминуемо приведет к критической ошибке в его дальнейшей работе и краху вычислительного процесса.

Операционные системы Windows NT/2000/XP

Краткая историческая справка

Компания Microsoft в 1990 году объявила о начале работ по созданию принципиально новой операционной системы для персональных IBM PC-совместимых ком-

[1] Программа SYSMON.EXE входит в состав штатного программного обеспечения систем Windows 9x, но при обычной установке она не устанавливается — требуется выборочная установка.

пьютеров с прицелом на корпоративный сектор, которая помимо банальной мультизадачности и поддержки виртуальной памяти обладала бы, в частности, такими качествами, как:

- микроядерная архитектура — сказалось влияние идей проекта Mach 3, выполненного в университете Карнеги Меллон (Carnegie Mellone University), которое в то время было очень велико;
- аппаратная независимость (platform independent), что должно было обеспечить легкую переносимость системы;
- мультипроцессорная обработка и масштабируемость (в то время операционные системы семейства UNIX обеспечивали работу на мультипроцессорных компьютерах и фактически доминировали как мощные корпоративные серверные системы);
- возможность выполнения приложений, созданных для других операционных систем, в частности приложений для UNIX и 16-разрядных программ OS/2;
- защита информации и вычислений от несанкционированного доступа;
- наличие высокопроизводительной и надежной файловой системы и возможность работать с несколькими файловыми системами;
- встроенные сетевые функции и поддержка распределенных вычислений.

Этот проект изначально имел название OS/2 version 3.0, однако впоследствии Microsoft назвала его Windows NT. Аббревиатура NT означала «New Technology», что подчеркивало принципиальную новизну этой операционной системы. Операционная система вышла в 1993 г. в двух вариантах и имела название Windows NT 3.1 и Windows NT Advanced Server 3.1. Эти системы обладали большими возможностями. Однако Windows NT 3.1 в качестве рабочей станции уступала системе OS/2, поскольку требовала существенно больше оперативной памяти и имела относительно низкое быстродействие. Кроме этого, при работе с дисками, отформатированными под файловую систему FAT, она не поддерживала длинные имена. Основным конкурентом серверной системы был сервер Novell Netware 3.x. После выхода первой версии Windows NT Microsoft выпустила Windows NT 3.5 для рабочих станций и одноименную серверную операционную систему. Последняя имела встроенное программное обеспечение для связи с серверами от Novell, поддерживала длинные имена при работе с дисками FAT, и много других усовершенствований. В те годы в качестве серверов для локальных вычислительных сетей преимущественно использовалась операционная система Netware 3.x компании Novell. В последующем эта сетевая операционная система была заменена существенно более мощной Netware 4.x, которая была предназначена для больших корпоративных сетей и имела службу каталогов, предназначенную для централизованного хранения информации о сетевых ресурсах. Она имела продуманные механизмы администрирования и была высокоэффективной. Завершилось поколение операционных систем Windows NT 3.x версиями под номером 3.5.1. Системы Windows NT 3.x не смогли тогда завоевать признание ни в качестве серверных, ни в качестве обычных настольных систем, поскольку требовали очень больших (по меркам того времени) вычислительных ресурсов.

Как ни странно, но еще одним недостатком этих первых систем Windows NT было строгое следование идеям микроядерной архитектуры. Согласно идеологии клиент-сервер, которой придерживались разработчики Windows NT 3.x, только ядро и низкоуровневые драйверы работали в нулевом кольце привилегий. А драйверы графической подсистемы, модули GDI, менеджер окон (Window Manager) и другие компоненты графической подсистемы работали как службы, то есть в пользовательском режиме работы процессора. Такое решение обеспечивало высокую надежность системы, но отрицательно сказывалось на ее производительности, поскольку приходилось многократно переключаться из режима ядра в пользовательский режим и обратно. Полезно напомнить, что сделать это можно только через механизм шлюзования. К тому же интерфейс этих первых операционных систем класса NT соответствовал обычной 16-разрядной системе Windows 3.x, быстро уходившей в прошлое, и заметно отличался от интерфейса Windows 95. Желая исправить эти недочеты, Microsoft запустила проект Cairo и в 1996 г. выпустила операционные системы Windows NT 4.0 Sever и Windows NT 4.0 Workstation.

Операционные системы Windows NT 4.0 оказались на редкость удачными. К моменту их выхода вычислительные ресурсы среднего персонального компьютера уже были достаточными для эффективной работы. Эти операционные системы в качестве основного ресурса требовали оперативную память. Официально серверная система требовала 16 Мбайт, а рабочая станция — 12 Мбайт, в то время как для реальной работы памяти нужно было иметь раза в четаре больше. И поскольку стоимость модулей полупроводниковой памяти для персональных компьютеров в те годы очень заметно снизилась, организации и отдельные пользователи стали массово осваивать эти операционные системы. А упомянутый перевод части кода, ответственного за работу графической подсистемы, в привилегированный режим работы процессора существенно увеличил быстродействие при обработке графики и позволил в последующем начать перенос пользовательских операционных систем на NT.

К сожалению, в своей новой операционной системе компания Microsoft отказалась от поддержки высокопроизводительной файловой системы HPFS, с которой работают операционные системы OS/2, хотя при желании пользователь мог сам добавить соответствующие драйверы из дистрибутива предыдущей Windows NT 3.x. Это был один из тех мелких уколов, которые в совокупности помогали компании Microsoft «уводить» пользователей от операционных систем OS/2.

Желая противопоставить свою серверную операционную систему известным сетевым операционным системам корпоративного уровня Novell Netware 4.x и Netware 5.x, компания Microsoft разработала новое семейство операционных систем класса NT, которое должно было изначально называться Windows NT 5.0, однако из маркетинговых соображений было переименовано в Windows 2000. В семейство этих систем вошли четыре операционные системы.

- Windows 2000 Professional — для использования в качестве рабочей станции вместо Windows NT.40 Workstation или Windows 98. Эта операционная система может работать на 2-процессорных компьютерах.
- Windows 2000 Server — для использования в качестве контроллера домена и/или сервера (файлов, приложений, баз данных, web и/или FTP, печати и т. д.) в от-

носительно небольшой сети, которую могут себе позволить иметь предприятия малого и среднего бизнеса. Эта операционная система поддерживает 4-процессорные конфигурации.

- Windows 2000 Advanced Server — для тех же целей, что и Windows 2000 Server, но с упором на выполнение функций сервера приложений и сервера баз данных. Обладает возможностью работать на компьютере с восемью процессорами и, самое главное, организовать кластер из двух машин.
- Windows 2000 Datacenter Server — специальная версия операционной системы, предназначенная для работы в вычислительных сетях крупных предприятий. Система хорошо масштабируется, позволяет построить 4-узловой кластер, причем каждая из машин может иметь вплоть до 16 процессоров[1].

Наверное, самыми главными особенностями этих операционных систем (по сравнению с предыдущими Windows NT 4.0) следует назвать поддержку механизма Plug and Play (как и в системах Windows 9x) и использование службы каталогов как основы для построения сетей клиент-сервер. Служба каталогов Microsoft получила наименование *Active Directory*. Принципиальной особенностью этой технологии является ее глубокая интеграция с TCP/IP. Кроме этого, нельзя не отметить, что новые операционные системы получили переработанную систему управления файлами, которая получила наименование NTFS5[2]. Интересно отметить, что были удалены все остатки кода, до этого позволявшие устанавливать файловую систему HPFS.

Для этого поколения операционных систем Microsoft сочла нецелесообразным переносить их на платформы Alpha (DEC), PowerPC, MIPS.

Осенью 2001 года Microsoft обновила операционную систему Windows 2000 Professional до Windows XP (eXPerience). При этом она выпустила две редакции. Одна из них представляла собой «облегченный» вариант системы для домашнего применения. Она получила название Windows XP Home Edition. Эту операционную систему Microsoft считает основной для современного персонального компьютера. Вторая — полноценная система с предназначением работать в качестве рабочей станции, которая, как правило, подключается к локальной вычислительной сети с выходом в Интернет. Эти операционные системы, прежде всего, получили возможность выполнять приложения, которые использовали оба подмножества функций Win32 API: и для Windows 9x, и для систем класса NT. Системы Windows XP в еще большей мере стали мультимедийными и ориентированными на Интернет. Интересным новшеством для систем Windows стала возможность организовать одновременную работу с компьютером двух пользователей: для одного непосредственно (локально), а для второго удаленно с другого компьютера. В принципе, в этом нет ничего особенного. Например, операционная система UNIX позволяет без проблем организовать не только такое взаимодействие, но и полноценную мультитерминальную работу. Но для систем Windows — это явно новая возможность.

Наконец, весной 2003 года на замену семейству Windows 2000 вышли несколько серверных операционных систем, которые получили в название число 2003. Это сле-

---

[1] За счет специальных расширений, которые могут быть разработаны изготовителями аппаратуры, допускают возможность работать на компьютерах, насчитывающих до 32 процессоров.

[2] Подробней об этой файловой системе см. в главе 6.

дующие 32-разрядные операционные системы для микропроцессоров с архитектурой ia-32.

- Windows Small Business Server 2003 — предназначена для построения небольших локальных вычислительных сетей.
- Windows Server 2003 Web Edition — это самая «облегченная» система, она не может выступать в роли контроллера домена и быть сервером приложений.
- Windows Server 2003 Standard Edition — основная многоцелевая операционная система, пришедшая на смену Windows 2000 Server.
- Windows Server 2003 Enterprise Edition — аналог Windows 2000 Advanced Server.
- Windows Server 2003 Datacenter Edition.

Последние две операционные системы имеют разновидности для 64-разрядных процессоров Itanium 2 производства компании Intel.

Ничего революционного эти системы не привнесли, но существенно обновили предыдущие серверные операционные системы. В качестве основных особенностей новых систем Microsoft отмечает упрощение администрирования, более безопасную инфраструктуру и более высокую надежность, интеграцию в системы активно продвигаемой технологии *.NET* (произносится как «дот нет»).

Основные особенности архитектуры

Наиболее принципиальным отличием между системами класса Windows 9x и Windows NT является то, что у них разная архитектура.

Большинство операционных систем использует такую особенность современных процессоров, как возможность работать в одном из двух режимов: *привилегированном* (режиме ядра, или режиме супервизора) и *пользовательском* (режиме выполнения приложений). При описании своей системы Windows NT Microsoft для указания этих режимов использует термины *kernel mode* и *user mode* соответственно. Программные коды, которые выполняются процессором в привилегированном режиме, имеют доступ и к системным аппаратным средствам, и к системным данным. Чтобы защитить операционную систему и данные, располагающиеся в оперативной памяти, от ошибок приложений или их преднамеренного вмешательства в чужие вычисления, только системному коду, относящемуся к управляющей (супервизорной) части операционной системы, разрешают выполняться в привилегированном режиме работы процессора. Все остальные программные модули должны выполняться в пользовательском режиме.

Поскольку при создании Windows NT разработчики хотели обеспечить ее мобильность, то есть легкую переносимость на другие платформы, они приняли решение использовать только два уровня привилегий из четырех, имеющихся в микропроцессорах Intel семейства i80x86. Как мы уже знаем, нулевой уровень привилегий в микропроцессорах с архитектурой ia32 обеспечивает возможность выполнять любые команды и иметь доступ ко всем регистрам процессора. Наименьшие привилегии имеются у кода, выполняемого в третьем кольце защиты (см. главу 4), которое и предназначается для выполнения обычных приложений. Напомним, что код,

работающий в этом режиме, не может ни при каких обстоятельствах получить доступ к данным, расположенным в нулевом кольце защиты. Поэтому, если бы системный код использовал не два уровня привилегий, а все четыре, то появились бы очевидные проблемы при переносе системы на другой процессор.

Системы типа Windows NT построены по микроядерной технологии. Конечно, их ядро никак нельзя назвать маленьким, особенно в сравнении с ядром операционной системы QNX. Однако в целом архитектура Windows NT безусловно отвечает идеям построения операционной системы, в которой управляющие модули организованы с четким выделением центральной части и взаимодействием этой части с остальными по принципу клиент-сервер. Это означает, что в состав ядра включены только самые важные основообразующие управляющие процедуры, а остальные управляющие модули операционной системы вызываются из ядра как службы. Причем только часть служб использует процессор в режиме ядра, а остальные — в пользовательском режиме, как и обычные приложения пользователей (рис. 11.2). А для обеспечения надежности они располагаются в отдельном виртуальном адресном пространстве, к которому ни один модуль и ни одна прикладная программа, помимо системного кода, не может иметь доступа.

Рис. 11.2. Архитектура операционных систем класса Windows NT

Ядро (микроядро) систем Windows NT выполняет диспетчеризацию задач (точнее, потоков), обработку прерываний и исключений, поддерживает механизмы синхронизации потоков и процессов, обеспечивает взаимосвязи между всеми остальными компонентами операционной системы, работающими в режиме ядра. Если компьютер имеет микропроцессорную архитектуру (системы класса Windows NT поддерживают симметричную мультипроцессорную архитектуру[1]), ядро повышает его производительность, синхронизируя работу процессоров.

Из рисунка видно, что помимо собственно ядра в том же режиме супервизора работают модуль HAL (Hardware Abstraction Layer — уровень абстракции аппаратных средств), низкоуровневые драйверы устройств и исполняющая система Windows NT, называемая *Win32 Executive*. Начиная с Windows NT 4.0 в режиме ядра работают и диспетчер окон (Window Manager), который иногда называют «User», и модули графического интерфейса устройств (GDI).

Программное обеспечение, абстрагирующее работу исполняющей системы и собственно ядра от специфики работы конкретных устройств и контроллеров, во многом упрощает перенос операционной системы на другую платформу. Оно представлено в системе модулем динамически связываемой библиотеки HAL.DLL.

Одним из важнейших компонентов операционных систем Windows NT/2000/XP, который появился вследствие следования микроядерному принципу их построения, является исполняющая система (Win32 Executive). Она выполняет такие базовые функции операционной системы, как управление процессами и потоками, управление памятью, взаимодействие между процессами, защиту, операции ввода-вывода (включая файловые операции, кэширование, работу с сетью и некоторые другие). Ниже перечислены компоненты исполняющей системы.

- Диспетчер процессов (Process Manager) создает, отслеживает и удаляет процессы. Для выполнения этих функций создается соответствующий дескриптор, определяются базовый приоритет процесса и карта адресного пространства, создается и поддерживается список всех готовых к выполнению потоков.
- Диспетчер виртуальной памяти (Virtual Memory Manager) предоставляет виртуальную память выполняющимся процессам. Каждый процесс имеет отдельное адресное пространство, используется страничное преобразование линейных адресов в физические, поэтому потоки одного процесса не имеют доступа к физическим страницам, отведенным для другого процесса.
- Диспетчер объектов (Object Manager) создает и поддерживает объекты. В частности, поддерживаются дескрипторы объектов и атрибуты защиты объектов. Объектами считаются каталоги, файлы, процессы и потоки, семафоры и события и многие другие.
- Монитор безопасности (Security Reference Monitor) обеспечивает санкционирование доступа к объектам, контроль полномочий доступа и ведение аудита. Совместно с процессом входа в систему (logon) и защищенными подсистемами реализует модель безопасности Windows NT.

[1] Микроядро может одновременно выполняться на всех процессорах, а потоки одного процесса могут одновременно выполняться на нескольких процессорах.

- Диспетчер ввода-вывода (Input/Output Manager) управляет всеми операциями ввода-вывода в системе. Организует взаимодействие и передачу данных между всеми драйверами, включая драйверы файловых систем, драйверы физических устройств, сетевые драйверы, для чего используются структуры данных, называемые *пакетами запросов на ввод-вывод* (I/O Request Packet, IRP). Запросы на ввод-вывод обрабатываются в порядке приоритетов, а не в порядке их поступления. Операции ввода-вывода кэшируются, этим процессом управляет диспетчер кэша (Cache Manager). Поддерживаются различные файловые системы, причем драйверы[1] этих систем воспринимаются диспетчером ввода-вывода как драйверы физических устройств. Специальное сетевое системное программное обеспечение (*редиректор*[2] и *сервер*[3]) трактуются как сетевые драйверы и также имеют непосредственную связь с диспетчером ввода-вывода.
- Средства *вызова локальных процедур* (Local Procedure Call, LPC) обеспечивают выполняющиеся подсистемы среды выполнения и приложения пользователей коммуникационным механизмом, в котором взаимодействие строится по принципу клиент-сервер.

Для системных данных и программного кода, работающего в режиме ядра, не предусмотрено никакой защиты. Это означает, что некорректно написанный драйвер устройства может разрушить вычисления, выполняемые собственно операционной системой. Поэтому необходимо очень осторожно относиться к выбору таких драйверов и использовать только те, которые были тщательно оттестированы. Последние версии операционных систем, включая поколение Windows 2000, имеют специальный механизм проверки цифровой подписи Microsoft, наличие которой означает, что драйвер прошел всестороннее тестирование. Это должно выступать гарантом качества системного кода.

Остальные системные модули операционной системы, относящиеся к организации соответствующей среды выполнения, выполнению ряда функций, связанных с обеспечением защиты, модуль серверного процесса, который обеспечивает возможность приложениям обращаться к операционной системе с соответствующими запросами, и многие другие выполняются в пользовательском режиме работы процессора.

Диспетчеризация в системах Windows NT/2000/XP организована почти так же, как и в системах Windows 95/98/ME. Все эти операционные системы относятся к мультизадачным и поддерживают потоковые вычисления. 16-разрядные приложения Windows, работая на одной виртуальной машине, разделяют процессорное время кооперативно. 32-разрядные потоки разделяют процессорное время, вытесняя друг друга через некоторые моменты времени. При этом диспетчер задач (пла-

---

[1] Здесь можно было бы употребить термин «системы управления файлами».

[2] Этот модуль перехватывает запрос на ввод-вывод и проверяет, к каким ресурсам он относится, локальным (расположены непосредственно на том же компьютере) или удаленным (предоставлены в общий доступ через сеть и могут быть расположены на любом компьютере).

[3] В данном контексте речь идет о модуле системного сетевого программного обеспечения, который получает запрос на обслуживание или ресурс от другого компьютера (посылаемый им после прохождения через свой модуль редиректора) и генерирует соответствующий запрос к операционной системе.

нировщик потоков) работает с несколькими очередями. Всего существует 32 уровня приоритетов — от 0 до 31. Распределение приоритетов между выполняющимися процессами и потоками осуществляется по следующим правилам:

- Low — 4 (низкий приоритет);
- BelowNormal — ниже среднего;
- Normal — 8 (нормальный приоритет);
- AboveNormal — выше среднего;
- High — 16 (высокий приоритет);
- RealTime — 24 (приоритет реального времени).

Собственно исполняемыми элементами процесса являются потоки. Как мы уже знаем, каждый процесс имеет, по крайней мере, один поток. Поток получает базовый приоритет от своего процесса, а фактическое значение приоритета присваивается потоку операционной системой. Те потоки, которые выполняются на *переднем плане* (foreground), получают приращение приоритета относительно базового.

У потоков, выполняемых в *фоновом режиме* (background), приоритет уменьшается. По умолчанию все задачи запускаются с нормальным приоритетом. Обычный пользователь может изменить приоритет задачи вплоть до высокого. Приоритет реального времени может присвоить только администратор.

Используемые дисциплины диспетчеризации у всех этих операционных систем одинаковы. Однако если внимательно понаблюдать за тем, как ведут себя системы Windows NT/2000/XP и системы Windows 95/98/ME, выполняя параллельно множество запущенных приложений, то можно без особого труда заметить, что многозадачность у первых реализована значительно лучше. Причина такого явления заключается в том, что с разными затратами времени происходят изменения в подсистеме управления памятью. При переключении с одного вычислительного процесса на другой необходимо поменять значение регистра CR3, с помощью которого линейные адреса команд и операндов пересчитываются в реальные физические. В операционных системах Windows NT/2000/XP (как и в OS/2, и в Linux) используется вся та аппаратная поддержка двухэтапного вычисления физических адресов, которая имеется в микропроцессорах. То есть при переключении процессора на новую задачу смена значения регистра CR3, а значит, и замена всех дескрипторных таблиц, описывающих местонахождение виртуальных страниц процесса и его потоков, осуществляется автоматически. А в системах Windows 95/98/ME вместо инициализации одного регистра, указывающего на адрес таблицы PDE (см. в главе 4 описание страничного способа организации виртуальной памяти в микропроцессорах i80x86), операционная система переписывает все содержимое целой физической страницы, на которую указывает регистр CR3 вместо простой замены содержимого этого регистра. И поскольку такая операция требует совершенно иных затрат времени, мы и наблюдаем тот факт, что многозадачность в системах Windows 95/98/ME реализована намного хуже, чем в системах класса NT.

Полезно знать, что операционные системы, предназначенные для построения рабочих станций (ранее Workstation, позже Professional), и серверные варианты строятся практически на одном ядре, но имеют разные настройки в реестре. Более того,

их дистрибутивы почти полностью совпадают (более чем на 90 %). Однако серверы не имеют ограничений на количество сетевых подключений к ним (эти ограничения определяются только количеством приобретенных лицензий) и позволяют установить и выполнять различные сетевые службы, например службу именования Windows для Интернета (Windows Internet Name Service, WINS), систему доменного именования (Domain Name System, DNS), протокол управления динамической адресацией компьютеров (Dynamic Host Control Protocol, DHCP), контроллер домена (domain controller) в локальной вычислительной сети и многие другие. В доказательство этому можно упомянуть известную утилиту NTSwitch.exe, которая при запуске превращает рабочую станцию в сервер или, наоборот, сервер — в рабочую станцию.

В заключение заметим, что мы очень кратко познакомились с архитектурой операционных систем Windows NT/2000/XP. Для более детального изучения целого ряда вопросов, связанных с этой темой, рекомендую такие известные книги, как [16, 42].

Модель безопасности

При разработке всех операционных систем семейства Windows NT/2000/XP компания Microsoft уделяла самое пристальное внимание обеспечению информационной безопасности. Как следствие, эти системы предоставляют надежные механизмы защиты, которые просты в использовании и легки в управлении. Сертификат безопасности на соответствие уровню C2 имеют операционные системы Windows NT 3.5 и Windows NT 4.0. Операционные системы семейства Windows 2000 имеют еще более серьезные средства обеспечения безопасности, однако на момент написания этой книги они еще не сертифицировались.

В отличие от операционных систем семейства Windows 9x, как, впрочем, и от системы OS/2, в разработке первой версии которой Microsoft тоже принимала участие, системы класса Windows NT имеют совершенно иную модель безопасности. Средства защиты изначально глубоко интегрированы в операционную систему. Подсистема безопасности осуществляет контроль за тем, кто и какие действия совершает в процессе работы, к каким объектам пытается получить доступ. Все действия пользователя, в том числе и обращения ко всем объектам, как нетрудно догадаться, на самом деле могут быть совершены только через соответствующие запросы к операционной системе. Операционная система использует этот факт и имеет все необходимые механизмы для тотального контроля всех запросов к ней. Запрашиваемые у операционной системы операции и обращения к конкретным объектам разрешаются, только если у пользователя для этого имеются необходимые *права* и/или *разрешения*. При этом обязательно следует различать эти понятия.

Права (rights) определяют уровень полномочий при работе в системе. Например, если нет права форматировать диск, то выполнить это действие пользователь не сможет. Кстати, конкретно таким правом при работе с Windows NT/2000/XP обладают только члены группы администраторов. Можно говорить и о праве изменения настроек дисплея, и о праве работать на компьютере. Очевидно, что пере-

чень прав является достаточно большим. Права могут быть изменены посредством применения соответствующих политик.

Термин *разрешение* (permission) обычно применяют по отношению к конкретным объектам, таким как файлы и каталоги, принтеры и некоторые другие. Можно говорить о разрешениях на чтение, на запись, на исполнение, на удаление и проч. Например, можно иметь разрешения на чтение и запуск некоторой программы, но не иметь разрешений на ее переименование и удаление.

Важно, что права имеют преимущество перед разрешениями. Например, если у некоторого пользователя нет разрешения «стать владельцем» того или иного файлового объекта, но при этом мы дадим ему право стать владельцем любого объекта, то он, дав запрос на владение упомянутым объектом, получит его в свою собственность.

Модель безопасности Windows NT гарантирует, что не удастся получить доступ к ее объектам без того, чтобы предварительно пройти *аутентификацию* и *авторизацию*. Для того чтобы иметь право работать на компьютере, необходимо иметь *учетную запись* (account). Учетные записи хранятся в базе данных учетных записей, которая представлена файлом SAM (Security Account Management). Каждая учетная запись в базе данных идентифицируется не по имени, а по специальному системному идентификатору. Такой идентификатор в Windows NT называется *идентификатором безопасности* (Security IDentifier, SID). Подсистема безопасности этих операционных систем гарантирует уникальность идентификаторов безопасности. Они генерируются при создании новых учетных записей и никогда не повторяются. Имеются встроенные учетные записи, но они тоже уникальны. Помимо учетных записей пользователей имеются учетные записи групп. Учетные записи имеют и компьютеры. Идентификаторы несут в себе информацию о типе учетной записи.

Учетные записи могут быть объединены в группы. Имеются встроенные группы. Принадлежность учетной записи к одной из встроенных групп определяет полномочия (права, привилегии) пользователя при работе на этом компьютере. Например, члены встроенной группы администраторов имеют максимально возможные права при работе на компьютере (встроенная учетная запись администратора равносильна учетной записи суперпользователя в UNIX-системах).

Вновь создаваемые учетные записи групп (их называют группами безопасности) используются для определения разрешений на доступ к тем или иным объектам. Для этого каждый объект может иметь *список управления доступом* (Access Control List, ACL)[1]. Список ACL состоит из записей — *ACE* (Access Control Entry). Каждая запись списка состоит из двух полей. В первом поле указывается некий идентификатор безопасности. Во втором поле располагается битовая маска доступа, описывающая, какие разрешения указаны в явном виде, какие не запрещены, и какие запрещены в явном виде для этого идентификатора. При использовании файловой системы NTFS список ACL реально представлен списком *DACL* (Discretionary Access Control List). Ранее (в главе 6) был изложен механизм работы

[1] На самом деле файловые объекты (файлы и каталоги с файлами) имеют списки управления доступом, только если они расположены на дисках с файловой системой NTFS.

разрешений NTFS, причем описаны разрешения как для прежней версии NTFS, использовавшиеся вплоть до Windows NT 4.0, так и для файловой системы NTFS5, которая появилась в Windows 2000[1]. Здесь мы лишь заметим, что рекомендуется составлять списки управления доступом, пользуясь не учетными записями пользователей, а учетными записями групп. Во-первых, это позволяет существенно сократить список управления доступом, поскольку групп обычно намного меньше, чем пользователей. Как результат, список будет намного короче, понятнее и удобнее для последующего редактирования. Во-вторых, в последующем можно будет создать нового пользователя (и не единожды) и добавить его в соответствующие группы, что практически автоматически определит его разрешения на те или иные объекты как члена определенных групп. Наконец, в-третьих, список будет быстрее обрабатываться операционной системой.

Каждый пользователь должен быть членом как минимум одной встроенной группы и может быть членом нескольких групп безопасности, создаваемых в процессе эксплуатации операционной системы. При регистрации пользователь получает так называемый *маркер доступа*, который содержит, помимо идентификатора безопасности учетной записи пользователя, все идентификаторы групп, в которые пользователь входит. Именно этот маркер сопровождает любой запрос на получение ресурса (объекта), который передается операционной системе.

Итоговые разрешения на доступ к объектам, имеющим списки управления доступом, вычисляются как сумма разрешений, определенных для каждой из групп. И только явный запрет на разрешение перечеркивает сумму разрешений, которая получается для данного пользователя.

Операционные системы Windows NT/2000/XP обеспечивают защиту на локальном уровне. Это означает, что механизмы защиты работают на каждом компьютере с такой операционной системой. Однако это имеет и обратную сторону. Дело в том, что учетные записи пользователей и групп локальны: они действуют только на том компьютере, где их создали. Если же есть необходимость обратиться к общим ресурсам компьютера через сеть, нужно, чтобы для пользователя, который выполняет такое обращение к удаленным объектам, была создана такая же учетная запись. Поскольку становится затруднительным обеспечить наличие учетных записей для каждого пользователя на всех тех компьютерах, с ресурсами которых ему необходимо работать, пользуясь вычислительной сетью, в свое время была предложена технология доменных сетей. В *домене*, который представляет собой множество компьютеров, должен быть выделен сервер со всеми учетными записями этого домена. Такой сервер называют *контроллером домена*. Учетные записи домена[2] в отличие от локальных учетных записей, имеющихся на каждом компью-

---

[1] В новых серверных операционных системах Windows 2003 Server используется новая версия системы управления файлами, которая обеспечивает существенное увеличение производительности при работе с файлами. Версия файловой системы в этих операционных системах осталась прежней — NTFS5.

[2] На контроллере домена, работающем под управлением Windows NT 4.0 Server, база с учетными записями домена по-прежнему представлена файлом SAM. На контроллерах домена, работающих под управлением Windows 2000/2003 Server, база с учетными записями домена находится в файле NTDS.DIT, поскольку организация доменов в этих операционных системах возможна только при установке службы Active Directory.

тере с операционной системой типа Windows NT, являются перемещаемыми: они могут перемещаться с контроллера домена на любой другой компьютер этого домена. В результате, имея множество компьютеров, объединенных в домен, и контроллер домена, на котором созданы все необходимые учетные записи, мы можем использовать эти учетные записи для управления доступом к различным ресурсам. Более того, мы можем контролировать использование этих ресурсов и регистрировать попытки несанкционированного доступа к тем или иным объектам. Контроль за использованием прав и разрешений, а также их регистрация называется *аудитом*.

Распределение оперативной памяти

В операционных системах типа Windows NT тоже используется плоская модель памяти. Однако схема распределения виртуального адресного пространства разительно отличается от модели памяти Windows 9x. Прежде всего, в отличие от Windows 9x, в гораздо большей степени задействуется ряд серьезных аппаратных средств защиты, имеющихся в микропроцессорах, а также применено принципиально другое логическое распределение адресного пространства.

Все системные программные модули находятся в своих собственных виртуальных адресных пространствах, и доступ к ним со стороны прикладных программ невозможен. Центральная супервизорная часть системы, состоящая, как мы теперь знаем, из микроядра, исполняющей системы (Win32 executive), модуля обеспечения аппаратной независимости, графического интерфейса устройств и оконного менеджера, а также низкоуровневых драйверов устройств, работает в нулевом кольце защиты в отдельном адресном пространстве.

Остальные программные модули самой операционной системы, которые выступают как серверные процессы по отношению к прикладным программам (клиентам), функционируют также в собственном системном виртуальном адресном пространстве, невидимом и недоступном для прикладных процессов. Правда, из-за того, что эти программные модули расположены в другом кольце защиты, они изолированы от ядра системы. Логическое распределение адресных пространств иллюстрирует рис. 11.3.

Прикладным программам выделяется 2 Гбайт[1] локального (собственного) линейного (неструктурированного) адресного пространства от границы 64 Кбайт до 2 Гбайт (первые 64 Кбайт полностью недоступны). Прикладные программы изолированы друг от друга, хотя могут общаться через буфер обмена (clipboard), механизмы DDE (Dynamic Data Exchange — динамический обмен данными) и OLE (Object Linking and Embedding — связывание и внедрение объектов).

В верхней части каждой области прикладной программы размером по 2 Гбайт размещен код системных библиотек DLL кольца защиты 3, который перенаправляет вызовы в совершенно изолированное адресное пространство, где содержится уже собственно системный код. Этот системный код, выступающий как серверный процесс (server process), проверяет значения параметров, исполняет запрошенную

[1] В серверных версиях Windows 2000 эта граница проходит выше, приложениям выделяется до 3 Гбайт.

функцию и пересылает результаты назад в адресное пространство прикладной программы. Хотя серверный процесс сам по себе остается процессом прикладного уровня, он полностью защищен от вызывающей его прикладной программы и изолирован от нее.

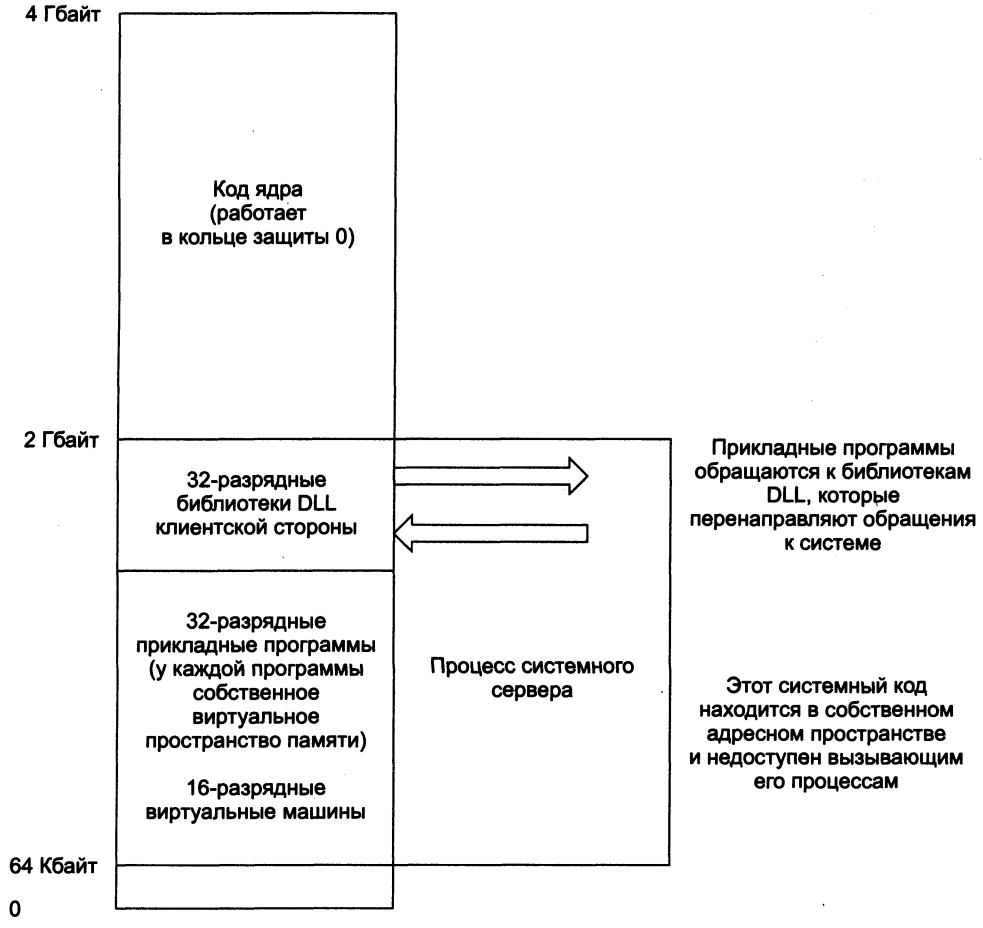


Рис. 11.3. Модель распределения виртуальной памяти в Windows NT

Между отметками 2 и 4 Гбайт расположены низкоуровневые системные компоненты Windows NT кольца защиты 0, в том числе ядро, планировщик потоков и диспетчер виртуальной памяти. Системные страницы в этой области наделены привилегиями супервизора, которые задаются физическими схемами колец защиты процессора. Это делает низкоуровневый системный код невидимым и недоступным по записи для программ прикладного уровня, но приводит к падению производительности из-за переходов между кольцами.

Для 16-разрядных прикладных Windows-программ операционные системы типа Windows NT реализуют сеансы Windows on Windows (WOW). В отличие от Win-

dows 9x, система Windows NT дает возможность выполнять 16-разрядные Windows-программы индивидуально в собственных пространствах памяти или совместно в разделяемом адресном пространстве. Почти во всех случаях 16- и 32-разрядные прикладные Windows-программы могут свободно взаимодействовать, используя механизм OLE, независимо от того, выполняются они в отдельной или общей памяти. Собственные прикладные программы и сеансы WOW выполняются в режиме вытесняющей многозадачности, основанной на управлении отдельными потоками. Несколько 16-разрядных прикладных Windows-программ в одном сеансе WOW выполняются в соответствии с кооперативной моделью многозадачности. Windows NT может также открыть в многозадачном режиме несколько сеансов DOS. Поскольку Windows NT имеет полностью 32-разрядную архитектуру, не существует теоретических ограничений на ресурсы компонентов GDI и User.

При запуске приложения создается *процесс* со своей информационной структурой. В рамках процесса запускается *поток выполнения*. При необходимости этот поток может инициировать запуск множества других потоков, которые будут выполняться параллельно в рамках одного процесса. Очевидно, что множество запущенных процессов также выполняются параллельно, и каждый из процессов может представлять собой мультизадачное (многопоточное) приложение. Задачи (потоки) в рамках одного процесса выполняются в едином виртуальном адресном пространстве, а процессы выполняются в различных виртуальных адресных пространствах. Короче, все это почти полностью напоминает Windows 9x.

Отображение различных виртуальных адресных пространств исполняющихся процессов на физическую память осуществляется по принципам страничной организации виртуальной памяти.

Процессами выделения памяти, ее резервирования, освобождения и замещения страниц управляет *диспетчер виртуальной памяти* (Virtual Memory Manager, VMM) Windows NT. В своей работе этот компонент реализует сложную стратегию учета требований к коду и данным процесса для минимизации обращений к диску, поскольку реализация виртуальной памяти часто приводит к большому количеству дисковых операций. Для взаимодействия между выполняющимися приложениями и между приложениями и кодом самой операционной системы используются соответствующие механизмы защиты памяти, поддерживаемые аппаратурой микропроцессора.

Каждая виртуальная страница памяти, отображаемая на физическую страницу, переносится в так называемый *страничный кадр* (page frame). Прежде чем код или данные можно будет переместить с диска в память, диспетчер виртуальной памяти должен найти или создать свободный или нулевой (заполненный нулями) страничный кадр. Заметим, что заполнение страниц нулями представляет собой одно из требований стандарта на системы безопасности уровня C2, принятого правительством США. Страничные кадры перед своим выделением должны заполняться нулями, чтобы исключить возможность использования их предыдущего содержимого другими процессами. Чтобы кадр можно было освободить, необходимо скопировать на диск изменения в его странице данных, и только после этого кадр можно будет повторно использовать. Программы, как правило, не меняют страниц кода. Такие страницы можно просто расформировать (удалить).

Диспетчер виртуальной памяти может быстро и относительно легко удовлетворить программные прерывания типа *страничной ошибки* (page fault). Что касается аппаратных прерываний типа страничной ошибки, то они приводят к необходимости *подкачки нужных страниц* (paging), что снижает производительность системы. Мы уже говорили (см. главу 3), что в Windows NT, к большому сожалению, для замещения страниц выбрана дисциплина FIFO, а не более эффективная дисциплина LRU или LFU, как это сделано в других операционных системах.

Когда процесс использует код или данные, находящиеся в физической памяти, система резервирует место для этой страницы в файле подкачки Pagefile.sys на диске. Это делается с расчетом на то, что данные потребуется выгрузить на диск. Файл Pagefile.sys представляет собой *зарезервированный* блок дискового пространства, который используется для выгрузки страниц, помеченных как «грязные», для освобождения физической памяти. Заметим, что этот файл может быть как непрерывным, так и фрагментированным; он может быть расположен на системном диске или на любом другом и даже на нескольких дисках. Размер этого страничного файла ограничивает объем данных, которые могут храниться во внешней памяти при использовании механизмов виртуальной памяти. По умолчанию размер файла подкачки в операционных системах Windows NT 4.0 устанавливается равным объему физической памяти плюс 12 Мбайт, однако пользователь имеет возможность изменить его размер по своему усмотрению. В следующих системах (Windows 2000/XP) начальный размер страничного файла подкачки берется равным полуторакратному объему физической оперативной памяти. То есть, например, для компьютера, имеющего 512 Мбайт оперативной памяти, по умолчанию размер файла Pagefile.sys равен 768 Мбайт. Проблема нехватки виртуальной памяти часто может быть решена за счет увеличения размера файла подкачки. Файл подкачки может быть не один — система поддерживает до 16 файлов подкачки, поэтому лучше создать их несколько и разместить на быстрых жестких дисках.

В системах Windows NT 4.0 объекты, создаваемые и используемые приложениями и операционной системой, хранятся в так называемых *пулах памяти* (memory pools). Доступ к этим пулам может быть получен только в привилегированном режиме работы процессора, в котором функционируют компоненты операционной системы. Поэтому для того чтобы объекты, хранящиеся в пулах, стали видимы потокам выполнения приложений, эти потоки должны переключиться в привилегированный режим.

Перемещаемый, или *нерезидентный*, *пул* (paged pool) содержит объекты, которые могут быть при необходимости выгружены на диск. *Неперемещаемый*, или *резидентный*, *пул* (nonpaged pool) содержит объекты, которые должны постоянно находиться в памяти. В частности, к такого рода объектам относятся структуры данных, используемые процедурами обработки прерываний, а также структуры, требуемые для предотвращения конфликтов в мультипроцессорных системах.

Исходный размер пулов определяется объемом физической памяти, доступной Windows NT. Впоследствии размер пула устанавливается динамически и в зависимости от работающих в системе приложений и служб может изменяться в широком диапазоне значений.

Вся виртуальная память в Windows NT подразделяется на зарезервированную (reserved), выделенную (committed) и доступную (available).

- *Зарезервированная память* представляет собой набор непрерывных адресов, которые диспетчер виртуальной памяти (VMM) выделяет для процесса, но не учитывает в общей квоте памяти процесса до тех пор, пока она не будет фактически задействована. Когда процессу требуется выполнить запись в память, ему выделяется нужный объем из зарезервированной памяти. Если процессу потребуется больший объем памяти, то при наличии в системе доступной памяти дополнительная память может быть одновременно зарезервирована и использована.
- *Память выделена*, если диспетчер виртуальной памяти резервирует для нее место в файле Pagefile.sys на тот случай, когда потребуется выгрузить содержимое памяти на диск. Объем выделенной памяти процесса характеризует фактически потребляемый им объем памяти. Выделенная память ограничивается размером файла подкачки. Предельный объем выделенной памяти в системе (commit limit) определяется тем, какой объем памяти можно выделить процессам без увеличения размеров файла подкачки. Если в системе достаточно дискового пространства, то файл подкачки может быть увеличен, тем самым будет расширен предельный объем выделенной памяти.
- Вся память, которая не является ни выделенной, ни зарезервированной, является *доступной*. К доступной относится свободная память, обнуленная память (освобожденная и заполненная нулями), а также память, находящаяся в *списке ожидания* (standby list), то есть та, которая была удалена из рабочего набора процесса, но может быть затребована вновь.

Контрольные вопросы и задачи

Вопросы для проверки

1. Опишите основные архитектурные особенности операционных систем семейства Windows 9x.
2. Расскажите об организации мультизадачности в операционных системах Windows. Какие методы диспетчеризации используются в этих операционных системах?
3. Расскажите об управлении памятью в операционных системах семейства Windows 9x. Приведите карту распределения памяти и объясните причины невысокой надежности этих операционных систем.
4. Перечислите используемые планировщиком механизмы, которые обеспечивают бесперебойную работу системы и быструю реакцию на действия пользователя.
5. Опишите основные архитектурные особенности операционных систем семейства Windows NT.
6. Перечислите функции ядра (микроядра). Какова роль исполняющей системы (Win32 executive)? Какие основные компоненты входят в ее состав?

7. Какие функции выполняют компоненты Window Manager, GDI и драйверы графических устройств? Зачем их код получил нулевой уровень привилегий? Укажите положительные и отрицательные стороны этого решения.
8. Изложите основные идеи модели безопасности, принятой в системах Windows NT. Что следует понимать под терминами «права» и «разрешения»? Чем определяются права конкретного пользователя?
9. Что представляет собой список управления доступом? Расскажите о разрешениях файловой системы NTFS. Что такое SID?
10. Что означает локальность учетной записи? Бывают ли глобальные (перемещаемые) учетные записи? Что такое домен? Какую роль играет контроллер домена?
11. Расскажите об управлении памятью в операционных системах семейства Windows NT. Приведите карту распределения памяти и объясните причины высокой надежности этих операционных систем.

Задания

1. Изучите работу утилиты SysMon.exe (системный монитор, System monitor), входящей в состав операционных систем Windows 9x. Исследуйте загрузку центрального процессора и подсистемы управления памятью (использование оперативной памяти и файла подкачки) при запуске ресурсоемких приложений (например, Adobe Photoshop или аналогичных ему в плане расходования вычислительных ресурсов).
2. Изучите работу утилиты PerfMon.exe (системный монитор, Performance monitor), входящей в состав операционных систем Windows NT/2000/XP. Исследуйте загрузку центрального процессора и подсистемы управления памятью (использование оперативной памяти и файла подкачки) при запуске ресурсоемких приложений (например, Adobe Photoshop или аналогичных ему в плане расходования вычислительных ресурсов).

Список терминов

Термин	Аббревиатура	Перевод
Access		Обращение
Access Control Entry	ACE	Запись списка управления доступом
Access Control List	ACL	Список управления доступом
Access mask		Маска доступа
Account		Учетная запись
Account name		Входное, или учетное, имя
Active Directory		Активный каталог
Allocation block		Блок размещения
Application Program Interface	API	Интерфейс прикладного программирования
Auditing		Аудит
Available (memory)		Доступная (память)
Background session		Фоновый сеанс
Bad		Плохой (блок или кластер)
Bad block list		Список дефектных блоков
Band		Полоса
Base Input-Output System	BIOS	Базовая подсистема ввода-вывода
Batch		Пакет, пакетный (файл)
Binary Tree	B-Tree	Двоичное дерево
Bit map		Битовая карта
Bit Test and Reset	BTS	Проверка и установка бита
Bitmap block list		Список битовых карт
Boot block		Загрузочный блок
Boot manager		Менеджер загрузки
Boot Record	BR	Загрузочная запись
Bootstrap loader		Процедура начальной загрузки
Cache Manager		Диспетчер кэша
Capabilities		Полномочия

Термин	Аббревиатура	Перевод
Carry Flag	CF	Флаг переноса
Chain		Цепочка
Change permissions		Изменение разрешений
Change time		Время последнего изменения атрибутов
Chunk		Порция (данных)
Clean		Чистая (страница)
Code Segment	CS	Сегмент кода
Coder		Кодировщик
Committed		Выделенная (память)
Common Object Request Broker Architecture	CORBA	Общая архитектура посредника объектных запросов
Configuration Manager		Диспетчер конфигурации
Consumable Resource	CR	Расходуемый, или потребляемый, ресурс
Conventional memory		Основная, или стандартная, память
Cooperating processes		Взаимодействующие процессы
Cooperative multitasking		Кооперативная многозадачность
Critical Section	CS	Критическая секция
Current Privilege Level	CPL	Текущий уровень привилегий
Cylinder		Цилиндр
Cylinder-Head-Sector	C-H-S	Номера цилиндра, головки и сектора
Data Control Block	DCB	Блок управления данными
Deadlock		Тупик, клинч
Deny		Запрет
Descriptor Privilege Level	DPL	Уровень привилегии сегмента, определяемый его дескриптором
Desktop		Рабочий стол
Device Reference Table	DRT	Таблица описания виртуальных логических устройств
Direct Memory Access	DMA	Прямой доступ к памяти
Directory		Каталог
Directory band		Полоса каталогов
Directory emergency free block list		Список свободных запасных блоков каталогов
Dirty		Грязная (страница), грязный (бит)
Discretionary ACL	DACL	Дискреционный список ACL
Disk cache		Дисковый кэш
Disk Parameter Block	DPB	Блок параметров диска
Disks Operating System	DOS	Дисковая операционная система
Domain controller		Контроллер домена
Domain Name System	DNS	Система доменного именования
Dynamic Data Exchange	DDE	Динамический обмен данными

продолжение

Продолжение таблицы

Термин	Аббревиатура	Перевод
Dynamic Host Control Protocol	DHCP	Протокол управления динамической адресацией компьютеров
Dynamic Link Library	DLL	Динамически связываемые библиотеки
Dynamic priority variation		Динамическое изменение приоритета
Effective Group ID	EGID	Эффективный идентификатор группы
Effective Privilege Level	EPL	Эффективный уровень привилегий
Effective User ID	EUID	Эффективный идентификатор пользователя
Effective Performance		Эффективная производительность
Embedded		Внедренная (система)
Equipment table	EQT	Таблица оборудования
Execution priority		Приоритет выполнения
Expanded Memory Specification	EMS	Дополнительная (отображаемая) память
Extended		Расширенный (раздел)
Extended Attributes	EAs	Расширенные атрибуты
Extended Instruction Pointer	EIP	Расширенный указатель команд
Extended Memory Specification	XMS	Расширенная память
Extensable Architecture		Расширяемая архитектура
Extent		Фрагмент (файла)
Fast File System	FFS	Быстродействующая файловая система
Fault-Tolerant Networking		Надежная работа в сети
File Allocation Table	FAT	Таблица размещения файлов
File node	F-node	Файловый узел
File reference		Файловая ссылка
File System access Group ID	FSGID	Идентификатор доступа группы к файловой системе
File System access User ID	FSUID	Идентификатор доступа пользователя к файловой системе
First Come First Served	FCFS	Первым пришел, первым обслужен
First In First Out	FIFO	Первый пришедший первым и выбывает
First-class delivery		Первый класс доставки
FixPak		Пакет исправлений и обновлений
FLEET Transport Layer	FTL	Транспортный уровень FLEET
Folder		Папка
Foreground boost		Повышение приоритета активной задачи
Foreground session		Активный сеанс
Foreground task		Активная задача (с которой сейчас работает пользователь)
Form Disk	FDisk	Формирование диска

Термин	Аббревиатура	Перевод
Gate		Шлюз
Generic		Родовые (права доступа)
Global Descriptor Table	GDT	Глобальная таблица дескрипторов
Global Descriptor Table Register	GDTR	Регистр глобальной таблицы дескрипторов
Graphical Device Interface	GDI	Графический интерфейс устройства
Graphical User Interface	GUI	Графический интерфейс пользователя
Group Identifier	GID	Идентификатор группы
Hardware Abstraction Layer	HAL	Уровень абстракции аппаратных средств
Hardware Emulation Layer	HEL	Уровень эмуляции аппаратных средств
Head		Головка (чтения/записи данных)
Hidden		Невидимый (раздел)
High Memory Area	HMA	Область памяти с большими адресами (выше 1 Мбайт)
High Performance File System	HPFS	Высокопроизводительная файловая система
Hint		Подсказка
Home		Домашний (каталог)
Host		Хост
HotFix		Аварийное замещение
HotFix areas		Области аварийного замещения
HotFix map		Карта аварийного замещения
I/O Request Packet	IRP	Пакет запросов на ввод-вывод
Independed processes		Независимые процессы
Input/Output boost		Повышение приоритета ввода-вывода
Input/Output Manager		Диспетчер ввода-вывода
Input/Output Supervisor	IOS	Супервизор ввода-вывода
Installable File System	IFS	Устанавливаемая (монтируемая) файловая система
Institute of Electrical and Electronics Engineers	IEEE	Институт инженеров по электротехнике и радиоэлектронике
Instruction Pointer	IP	Указатель команд
Interface		Интерфейс
Inter-Process Communication	IPC	Взаимодействие между процессами
Interrupt		Прерывание
Interrupt Descriptor Table	IDT	Таблица дескрипторов прерываний
Interrupt Descriptor Table Register	IDTR	Регистр таблицы дескрипторов прерываний
Interrupt gate		Коммутатор прерываний

продолжение ⇩

Продолжение таблицы

Термин	Аббревиатура	Перевод
Interrupt if overflow	INTO	Прерывание по переполнению
Interrupt Request	INTR	Запрос на прерывание
Interrupt Request	IRQ	Линия запроса на прерывание
Job Control Language	JCL	Язык управления заданиями
Journaling File System	JFS	Файловая система с протоколированием
Kernel		Ядро
Kernel mode		Привилегированный режим, режим ядра, или режим супервизора
Last Come First Served		Последним пришел, первым обслужен
Lazy write		Отложенная запись
Least Frequently Used	LFU	Реже других используемый
Least Recently Used	LRU	Дольше других неиспользуемый
Least Recently Used	LRU	Дольше других неиспользуемый
Load Balancing on the Fly		Балансировка нагрузки на лету
Local Descriptor Table Register	LDTR	Регистр локальной таблицы дескрипторов
Local Procedure Call	LPC	Вызов локальных процедур
Log in		Регистрация, вход в систему
Logical block addressing	LBA	Логическая адресация блоков
Logical disk		Логический диск
Logical Disk Table	LDT	Таблица логических дисков
Logical Volume Manager	LVM	Менеджер логических дисков
Login		Входное, или учетное, имя
Long File Name	LFN	Длинное имя файла
Mailbox		Почтовый ящик
Mailslot		Гнездо почтового ящика
Main		Главная (часть, функция)
Marshalling		Сборка
Master		Главный (накопитель)
Master Boot Record	MBR	Главная загрузочная запись
Master File Table	MFT	Главная таблица файлов
Memory pool		Пул памяти
Modification time		Время последнего изменения
Module		Модуль
Module database		База данных модулей
MultiProcess Executing		Мультипрограммное выполнение вычислений
Multiprogramming with a Variable number of Tasks	MVT	Мультипрограммирование с переменным числом задач
Mutex		Мьютекс

Термин	Аббревиатура	Перевод
Named pipe		Именованный канал
National Computer Security Center		Национальный центр компьютерной безопасности
Native		Основная, естественная, или нативная (среда)
New Technology File System	NTFS	Файловая система новой технологии
Nice number, или nice		Относительный приоритет
No Mask Interrupt	NMI	Немаскируемое прерывание
Nonpaged pool		Неперемещаемый, или резидентный, пул
Non-preemptive multitasking		Не вытесняющая многозадачность
Non-System Bootstrap	NSB	Внесистемный загрузчик
Object Linking and Embedding	OLE	Связывание и внедрение объектов
Object Manager		Диспетчер объектов
Overlay		Оверлейная (структура)
Page		Страница
Page Descriptor	PD	Страничный дескриптор
Page Directory Entry	PDE	Таблица каталога таблиц страниц
Page fault		Страничная ошибка
Page frame		Страничный кадр, физическая страница
Page Table Entry	PTE	Таблица страниц
Paged pool		Перемещаемый, или нерезидентный, пул
Paging		Подкачка страниц
Paging file		Страничный файл
Partition		Раздел, часть (памяти или диска)
Partition Table	PT	Таблица разделов
Password		Пароль
Permission		Разрешение
Pipe		Канал связи, конвейер, транспортер
Platform independent		Аппаратная независимость
Pool		Пул
Portable Operating System Interface for Computer Environments	POSIX	Не зависимый от платформы системный интерфейс для компьютерного окружения
Power On Self Test	POST	Самотестирование при включении компьютера
Preemptive multitasking		Вытесняющая многозадачность
Primary		Первичный (раздел)
Primary scheduler		Основной планировщик
Priority		Приоритет

продолжение

Продолжение таблицы

Термин	Аббревиатура	Перевод
Priority boost		Повышение приоритета
Privilege Level	PL	Уровень привилегий
Process		Процесс
Process Identifier	PID	Идентификатор процесса
Process Manager		Диспетчер процессов
Producer-consumer		Производитель-потребитель
Profile		Профиль
Program Status Word	PSW	Слово состояния программы
Protected mode		Защищенный режим (работы процессора)
Proxy		Представитель
Queue		Очередь
Random		Случайный (выбор)
Read ahead		Упреждающее чтение
Real mode		Реальный режим (работы процессора)
Redirector		Редиректор
Re-enterable		Реентерабельный (программный модуль)
Re-entrance		Повторно входимый (программный модуль)
Region		Раздел, область (памяти)
Registry		Реестр
Regular		Регулярный (класс задач)
Release		Освободить
Remote Procedure Call	RPC	Вызов удаленных процедур
Requested Privilege Level	RPL	Запрашиваемый уровень привилегий
Reserved		Зарезервированная (память)
Reserved Sector	ResSecs	Зарезервированный сектор
Response time		Время отклика
Reusable Resource	RR	Многократно используемый ресурс
Right		Право
Root		Корень
Root Directory	RDir	Корневой каталог
Rotational latency		Время ожидания
Round Robin	RR	Карусельная (дисциплина обслуживания)
Run Time Library	RTL	Библиотека времени выполнения
Scheduler		Планировщик
Secondary MBR	SMBR	Вторичная запись MBR
Second-class delivery		Второй класс доставки
Sector		Сектор

Термин	Аббревиатура	Перевод
Security Account Management	SAM	База данных системы управления учетными записями
Security Identifier	SID	Идентификатор безопасности
Security Reference Monitor		Монитор безопасности
Seek time		Время на позиционирование (поиск) цилиндра
Segment		Сегмент
Semaphore		Семафор
Server Message Blocks	SMB	Блоки сообщений сервера (сетевая технология)
Server process		Серверный процесс
Set Interrupt Flag	STI	Установить флаг прерываний
Shell		Оболочка, интерпретатор команд
Shell script		Командный файл
Shortcut		Ярлык
Shortest Job Next	SJN	Следующим выполняется самое короткое задание
Shortest Remaining Time	SRT	Время выделяется заданию, которому осталось выполняться меньше всего времени
Shortest Seek Time First	SSTF	Запрос с наименьшим временем поиска выполняется первым
Simultaneous Peripheral Operation On-Line	Spooling	Имитация работы устройства в режиме подключения
Slave		Вспомогательный (накопитель)
Small		Малая (модель памяти)
Socket		Сокет
Spare block		Резервный блок
Specific		Специфичные (права доступа)
Spool-file		Спул-файл
Spool-reader		Спулер чтения
Spool-writer		Спулер записи
Standard		Стандартные (права доступа)
Standby list		Список ожидания
Starvation boost		Повышение приоритета «забытой» задачи
Stream		Поток данных
Stub		Заглушка
Subdirectory		Вложенный каталог, подкаталог
Super block		Дополнительный блок

продолжение ⇩

Продолжение таблицы

Термин	Аббревиатура	Перевод
Superuser		Суперпользователь
Supervisor		Супервизор
Swap partition		Раздел подкачки
Swap-file		Файл подкачки
Swapping		Замещение, подкачка, свопинг
System ACL	SACL	Системный список ACL
System Bootstrap	SB	Системный загрузчик
System Object Model	SOM	Модель системных объектов
System Resource	SR	Системный ресурс
System stack		Системный стек
Tainted		Испорченная (страница)
Task		Задача
Task gate		Коммутатор задачи
Task Register	TR	Регистр задачи
Task State Segment	TSS	Сегмент состояния задачи
Taskbar		Панель задач
Terminate and Stay Resident	TSR	Завершиться и остаться резидентным в памяти
Test and Set	TS	Проверка и установка
Thin		Легковесный (процесс)
Thread		Поток выполнения, тред, нить
Throughput		Пропускная способность
Time critical		Критические по времени (задачи)
Time sharing		Разделение времени
Time slice		Квант времени
Timeslice scheduler		Планировщик квантования
Track		Дорожка
Trap Flag	TF	Флаг трассировки
Trap gate		Коммутатор перехвата
Turnaround time		Время оборота
Unit Control Block	UCB	Блок управления устройством ввода-вывода
Universal Serial Bus	USB	Универсальная последовательная шина
Unmarshalling		Разборка
Upper Memory Areas	UMA	Области верхней памяти
User file-creation mask	umask	Пользовательская маска создания файла
User Identifier	UID	Идентификатор пользователя
User mode		Пользовательский режим

Термин	Аббревиатура	Перевод
Utilization		Загрузка
Variable priority		Переменный приоритет
Virgin		Абсолютно чистая (страница)
Virtual Device	VD	Виртуально устройство
Virtual DOS Machine	VDM	Виртуальная DOS-машина
Virtual FAT	VFAT	Виртуальная система FAT
Virtual Memory Manager	VMM	Диспетчер виртуальной памяти
Virtual Memory System	VMS	Система виртуальной памяти
Virtual Mode	VM	Виртуальный режим
Volume		Том
Waiting time		Время ожидания
Windows Internet Name Service	WINS	Служба именования Windows для Интернета
Word Counter	WC	Счетчик слов
Workplace Shell	WPS	Среда рабочего места, обеспечивающая графический режим работы в OS/2

Список литературы

1. *Александров Е. К., Рудня Ю. Л.* Микропроцессор 80386: как он работает и как работают с ним: Учеб. пособие / Под ред. проф. Д. В. Пузанкова. — СПб.: Элмор, 1994. 274 с.
2. *Богумирский Б. С.* Руководство пользователя ПЭВМ: В 2 ч. — СПб.: Ассоциация OILCO, 1992. 357с.
3. *Гордеев А. В., Кучин Н. В.* Проектирование взаимодействующих процессов в операционных системах: Учеб. пособие. Л.: ЛИАП, 1991. 72 с.
4. *Гордеев А. В., Молчанов А. Ю.* Системное программное обеспечение: Учебник. — СПб.: Питер, 2002. 736 с.
5. *Гордеев А. В., Молчанов А. Ю.* Применение сетей Петри для анализа вычислительных процессов и проектирования вычислительных систем: Учеб. пособие. Л.: ЛИАП, 1993. 80 с.
6. *Гордеев А. В., Никитин А. В., Фильчаков В. В.* Организация пакетов прикладных программ: Учеб. пособие. Л.: ЛИАП, 1988. 78 с.
7. *Гордеев А. В., Штепен В. А.* Управление процессами в операционных системах реального времени: Учеб. пособие. Л.: ЛИАП, 1988. 76 с.
8. *Григорьев В. Л.* Микропроцессор i486. Архитектура и программирование: В 4 кн. — М.: Гранал, 1993.
9. *Гудмэн Дж.* Секреты жесткого диска. — Киев: Диалектика, 1994. 256 с.
10. *Дейкстра Е.* Взаимодействующие последовательные процессы // Языки программирования / Под ред. Ф. Женюи. — М.: Мир, 1972.
11. *Дейтел Г.* Введение в операционные системы: В 2 т. / Пер. с англ. Л. А. Теплицкого, А. Б. Ходулева, В. С. Штаркмана; Под ред. В. С. Штаркмана. — М.: Мир, 1987.
12. *Джордейн Р.* Справочник программиста персональных компьютеров типа IBM PC, XT и AT / Пер. с англ. — М.: Финансы и статистика, 1991. 544 с.
13. *Дунаев С.* UNIX system v. 4.2: Общее руководство. — М.: Диалог-наука, 1995. 287 с.
14. *Коваленко И. Н.* QNX: Золушка в семье UNIX http://www.lgg.ru/~nigl/QNX/doc/Kovalenko_cinderella.html. 1995.
15. *Иртегов Д. В.* Введение в операционные системы. — СПб.: БХВ-Петербург, 2002. 624 с.

16. *Кастер Х.* Основы Windows NT и NTFS / Пер. с англ. — М.: Изд. отдел «Русская редакция» ТОО «Channel Trading Ltd.», 1996. 440 с.
17. *Кейлингерт П.* Элементы операционных систем. Введение для пользователей / Пер. с англ. Б. Л. Лисса и С. П. Тресковой. — М.: Мир, 1985. 295 с.
18. *Кейслер С.* Проектирование операционных систем для малых ЭВМ. — М.: Мир, 1986. 680 с.
19. *Краковяк С.* Основы организации и функционирования ОС ЭВМ. — М.: Мир, 1988. 480 с.
20. *Михальчук В. М., Ровдо А. А., Рыжиков С. В.* Микропроцессоры 80x86, Pentium: Архитектура, функционирование, программирование, оптимизация кода. — Минск: Битрикс, 1994. 400 с.
21. *Мурата Т.* Сети Петри: Свойства, анализ, приложения (обзор) // ТИИЭР, 1989. № 4. С. 41–85.
22. *Мэдник С., Донован Дж.* Операционные системы. — М.: Мир, 1978. 792 с.
23. *Немет Э., Снайдер Г., Сибасс С., Хейн Т.* UNIX: руководство системного администратора. Для профессионалов / Пер. с англ. — СПб.: Питер; Киев: Издательская группа BHV, 2002. 928 с.
24. *Нортон П.* Персональный компьютер фирмы IBM и операционная система MS-DOS / Пер. с англ. — М.: Радио и связь, 1992. 416 с.
25. *Нортон П., Гудмен Дж.* Внутренний мир персональных компьютеров. 8-е изд. Избранное от Питера Нортона / Пер. с англ.; Питер Нортон, Джон Гудмен. — К.: Диасофт, 1999. 584 с.
26. *Минаси М., Камарда Б. и др.* OS/2 Warp изнутри: В 2 т. / Пер. с англ. С. Сокорновой. — СПб.: Питер, 1996. Т. 1: 528 с.; Т. 2: 512 с.
27. *Обухов И.* QNX: Как надо делать операционные системы / PC Week RE. 1998. № 7. С. 58–59.
28. *Озеров В.* Советы по Дельфи (Версия 1.3.1 от 1.07.2000) — http://www.webmachine.ru/delphi.
29. *Олифер В. Г., Олифер Н. А.* Сетевые операционные системы.:Учебник. — СПб.: Питер, 2001. 544 с.
30. *Олифер Н. А., Олифер В. Г.* Сетевые операционные системы / Публ. Центра информационных технологий — www.citmgu.ru.
31. *Фодор Ж., Бонифас Д., Танги Ж.* Операционные системы — от PC до PS/2 / Пер. с франц. — М.: Мир, 1992. 319 с.
32. *Орловский Г. В.* Введение в архитектуру микропроцессора 80386. — СПб: Сеанс-Пресс Ltd; Инфокон, 1992. 240 с.
33. ОС QNX: Обзор системы / http://www.lgg.ru/~nigl/QNX/doc/about_qnx.html.
34. *Петерсен Р.* Linux: руководство по операционной системе: В 2 т. / Пер. с англ. — Киев: Издательская группа BHV, 1998.
35. *Петзолд Ч.* Программирование для Windows 95: В 2 т. / Пер. с англ. — СПб.: BHV — Санкт-Петербург, 1997.
36. *Питерсон Дж.* Теория сетей Петри и моделирование систем / Пер. с англ. — М.: Мир, 1984. 264 с.

37. Ресурсы Microsoft Windows 98 / Пер. с англ. — М.: Издательско-торговый дом «Русская редакция», 1999. 1288 с.
38. Ресурсы Microsoft Windows NT Workstation 4.0 / Пер. с англ. — СПб.: BHV — Санкт-Петербург, 1998. 800 с.
39. *Робачевский А. М.* Операционная система UNIX. — СПб.: BHV — Санкт-Петербург, 1997. 528 с.
40. *Рудаков П. И., Финогенов К. Г.* Программируем на языке ассемблера IBM PC. Ч. 3: Защищенный режим. — М.: Энтроп, 1996. 320 с.
41. *Соловьев Г. Н., Никитин В. Д.* Операционные системы ЭВМ: Учеб. пособие. — М.: Высшая школа, 1989. 255 с.
42. *Соломон Д., Руссинович М.* Внутреннее устройство Microsoft Windows 2000. Мастер-класс / Пер. с англ. — СПб.: Питер; М.: Издательско-торговый дом «Русская редакция», 2001. 752 с.
43. *Стивенс У.* UNIX: взаимодействие процессов. — СПб.: Питер, 2002. 576 с.
44. *Столлингс В.* Операционные системы. 4-е изд. / Пер. с англ. — М.: Издательский дом «Вильямс», 2002. 848 с.
45. *Студнев А.* Boot-менеджеры — кто они и откуда? // Byte Россия. 1998. № 4. С. 70–75.
46. *Таненбаум Э.* Современные операционные системы. 2-е изд. — СПб.: Питер, 2002. 1040 с.
47. *Тревеннор А.* Операционные системы малых ЭВМ / Пер. с англ. А. Г. Васильева. — М.: Финансы и статистика, 1987. 188 с.
48. *Фролов А. В., Фролов Г. В.* Защищенный режим процессоров Intel 80286, 80386, 80486. Практическое руководство по использованию защищенного режима. — М.: Диалог-МИФИ, 1993. 240 с.
49. *Фролов А. В., Фролов Г. В.* Операционная система OS/2 Warp. — М.: Диалог-МИФИ, 1995. 272 с. (Библиотека системного программиста; т. 20)
50. *Фролов А. В., Фролов Г. В.* Программирование для IBM OS/2 Warp: Ч. 1. — М.: Диалог-МИФИ, 1996. 288 с.
51. *Фролов А. В., Фролов Г. В.* Программирование для Windows NT. — М.: Диалог-МИФИ, 1996. (Библиотека системного программиста; т. 26, 27)
52. *Хоар Ч.* Взаимодействующие последовательные процессы. — М.: Мир, 1989. 264 с.
53. *Цикритзис Д., Бернстайн Ф.* Операционные системы / Пер. с англ. В. Л. Ушковой и Н. Б. Фейгельсон. — М.: Мир, 1977. 336 с.
54. *Шоу А.* Логическое проектирование операционных систем / Пер. с англ. В. В. Макарова и В. Д. Никитина. — М.: Мир, 1981. 360 с.
55. *Ющенко С. В.* ОС QNX — реальное время, реальные возможности // Мир ПК. 1995. № 5–6.
56. Microsoft Windows 2000: Server и Professional. Русские версии / Под общ. ред. А. Н. Чекмарева и Д. Б. Вишнякова. — СПб.: BHV, 2000. 1056 с.
57. Understanding Windows NT POSIX Compatibility» by Ray Cort Microsoft Corporate Technology Team, Created: May-June 1993.

Алфавитный указатель

A

access mask — маска доступа 196
account — учетная запись 388
ACE (Access Control Entry) 201, 388
ACL (Access Control List) 180, 201, 388
API (Application Program Interface) 14, 296, 298
auditing — аудит 288

B

B-tree (Binary Tree) 182
background — фоновый режим 386
background session — фоновый сеанс 187
bad block list — список дефектных блоков 180
BIOS (Base Input-Output System) 79
bitmap block list — список битовых карт 179
boot block — загрузочный блок 179
bootstrap loader — начальный загрузчик 151
BTS (Bit Test and Reset) 222

C

C-Scan — циклическое сканирование 161
chain — цепочка 168
clean page — чистая страница 374
conventional memory — основная, или стандартная, память 81
cooperating processes — взаимодействующие процессы 210
cooperative multitasking — кооперативная многозадачность 370
CPL (Current Privilege Level) 116
CPU throughput — пропускная способность процессора 64
CPU utilization — загрузка процессора 64
CR (Consumable Resource) 248
CS (Critical Section) 214
cylinder — цилиндр диска 146

D

DACL (Discretionary Access Control List) 196, 388
deadlock — тупик 247
desktop — рабочий стол 365
directory — каталог 166
directory band — полоса каталогов 180
dirty page — грязная страница 374
DMA (Direct Memory Access) 356
DOS (Disk Operating System) 78
DPB (Disk Parameter Block) 170
DPL (Descriptor Privilege Level) 116
DRT (Device Reference Table) 140
dynamic priority variation — динамическое изменение 65

E

EAs (Extended Attributes) 177
EMS (Expanded Memory Specification) 81
EPL (Effective Privilege Level) 117
equipment table — таблица оборудования 139
extended partition — расширенный раздел 150
extent — экстент 178

F

FAT (File Allocation Table) 164, 166
FCFS (First Come First Served) 56
file reference — файловая ссылка 193
FLEET 346
folder — папка 366
foreground — передний план 386
foreground session — активный сеанс 187
foreground task — задача переднего плана 54
FreeBSD 339
FSGID (File System access Group ID) 327
FSUID (File System access User ID) 327

G

gate — шлюз 118
GDI (Grafical Device Interface) 366

GDT (Global Descriptor Table) 106
GDTR (Global Descriptor Table Register) 105
GID (Group Identifier) 315
GUI (Graphical User Interface) 297, 361

H

HAL (Hardware Abstraction Layer) 283
head — головка чтения/записи 146
HMA (High Memory Area) 81
HotFix map — карта аварийного замещения 180
HPFS (High Performance File System) 177

I

IDT (Interrupt Descriptor Table) 125
IDTR (Interrupt Descriptor Table Register) 105, 122
IEEE (Institute of Electrical and Electronics Engi 304
IFS (Installable File System) 185, 188
independed processes — независимые процессы 210
interface — интерфейс 13
interrupt gate — коммутатор прерывания 125
INTR (Interrupt Request) 123
IPC (Inter Process Communication) 342
IRP (I/O Request Packet) 385

J

JCL (Job Control Language) 58
JFS (Journaling File System) 359

K

kernel mode — режим ядра 382

L

lazy write — отложенная запись 157, 185
LBA (Logical Block Addressing) 147
LDT (Local Descriptor Table) 106
LDT (Logical Disks Table) 151
LDTR (Local Descriptor Table Register) 104
LFU (Least Frequently Used) 90
Linux 336
logical disk — логический диск 151
login — входное, или учетное, имя 15
LPC (Local Procedure Call) 385
LRU (Least Recently Used) 90, 373
LVM (Logical Volume Manager) 359

M

MBR (Master Boot Record) 147
memory pool — пул памяти 393
MFT (Master File Table) 191
module — модуль 377
module database — база данных модулей 377
MPE (MultiProcess Executing) 305
multi-threaded — многопоточный 294
MULTICS (MULTiplexed Information and Computing Sys 312
mutex — мьютекс 229

N

native — основная, естественная, нативная 16
nonpaged pool — неперемещаемый пул 393
NSB (Non-System Bootstrap) 147
NTFS permissions — разрешения NTFS 194

P

page fault — страничная ошибка 393
page frame — страничный кадр 392
paged pool — перемещаемый пул 393
paging — подкачка страниц 393
partition — раздел диска 82, 146
PDE (Page Directory Entry) 111
permission — разрешение 388
PID (Process Identifier) 34
pipe — канал связи 242
PL (Privilege Level) 116
pool — пул 213
POSIX (Portable Operating System Interface for Computer Environments) 284, 304
POST (Power On Self Test) 79
primary partition — первичный раздел 150
primary scheduler — основной планировщик 372
priority boost — повышение приоритета 70
process — процесс 37
produces-consumer — производитель-потребитель 213
profile — профиль 366
protected mode — защищенный режим 102
PT (Partition Table) 147
PTE (Page Table Entry) 111

Q

QNX 341
queue — очередь 244

R

read ahead — упреждающее чтение 158
real mode — реальный режим 102
region — раздел диска 82
registry — реестр 367
release — освободить 227
release — освободить 239
request — запрос 239
reserved memory — зарезервированная память 394
response time — время отклика 64
REXX 354
right — право 387
rotational latency — время ожидания 178
RPC (Remote Procedure Call) 296, 335

RPL (Requested Privilege Level) 107, 117
RR (Reusable Resource) 248
RR (Round Robin) 59
RTL (Run Time Library) 298, 300

S

SACL (System Access Control List) 196
SAM (Security Account Management) 388
scheduler — планировщик 370
sector — сектор диска 146
seek time — время позиционирования 178
shell — оболочка, командный интерпретатор 316, 317
SID (Security Identifier) 196, 388
SJN (Shortest Job Next) 58
SOM (System Object Model) 355, 358
spare block — резервный блок 179
spooling — спулинг 136
SRT (Shortest Remaining Time) 58
SSTF (Shortest Seek Time First) 160
stream — поток данных 193, 322
subdirectory — подкаталог 166
super block — дополнительный блок 179
supervisor — супервизор 29
swapping — свопинг 84

T

tainted page — испорченная страница 374
task — задача 25, 50
task gate — коммутатор задачи 125
thread — тред, поток выполнения, нить 37, 348, 369
time critical — критическая по времени (задача) 69
time sharing — разделение времени 29, 320
time slice — квант времени 60
timeslice scheduler планировщик квантования 372
TR (Task Register) 36, 104, 107
track — дорожка диска 146
trap gate — коммутатор перехвата 125
TSR (Terminate and Stay Resident) 78
TSS (Task State Segment) 36, 51, 105, 107
turnaround time — время оборота 64

U

UCB (Unit Control Block) 139
UID (User Identifier) 315
UMA (Upper Memory Area) 81
user mode — пользовательский режим 382

V

variable priority — переменный приоритет 69
virgin page — абсолютно чистая страница 374
VMM (Virtual Memory Manager) 392
VMS (Virtual Memory System) 305
volume — том 170, 179, 191

W

wait — ожидать 227, 239
waiting time — время ожидания 64
WinAPI 304
Windows 361
WPS (WorkPlace Shell) 357

X

X-Window 358
XMS (Extended Memory Specification) 81

А

аварийное замещение 180, 184, 189
авторизация 14, 287, 388
адрес
  виртуальный 73
  линейный 108
адресное пространство
  виртуальное 73
  глобальное 106
  логическое 73
  локальное 104, 106
алгоритм
  банкира 266
  Деккера 219
атрибут файла 174, 194
аудит 288, 390
аутентификация 14, 287, 366, 388

Б

библиотека времени выполнения 298
бит
  обращения 90, 111
  присутствия 87, 111
блокировка памяти 215
буфер
  системный 143
  сообщений 240
  страниц 96
буферизация 143, 144

В

ввод
  асинхронный 143, 144
  синхронный 143
вектор прерывания 122
взаимное исключение 213, 214
виртуальная задача 350
виртуальная машина 113, 281, 314
виртуальное устройство 136
виртуальный терминал 337

время
оборота 64
ожидания 64
отклика 64
вывод
асинхронный 143, 144
буферизованный 143
синхронный 143
вызов
локальных процедур 385
удаленных процедур 335
вычислительные процессы
взаимодействующие 210
конкурирующие 211
независимые 210
параллельные 210
последовательные 25
сотрудничающие 213

Г

гарантия обслуживания 63
главная загрузочная запись 147
головка чтения/записи 146
граф повторно используемых
ресурсов 249, 268
группа 388
безопасности 388
встроенная 388

Д

двоичное дерево 182
двойная буферизация 157
дескриптор
задачи 36, 50, 51, 107
прерывания 125
процесса 32, 34, 50
сегмента 87, 106
страницы 94, 110
файла 166
шлюза 118, 125
динамическое присоединение 93
диск
динамический 191
логический 146, 147, 151
диспетчер 344
виртуальной памяти 392
задач 50
памяти 85
диспетчер
окон 317
диспетчеризация 55
дисциплина обслуживания 30, 55
FCFS 56
RR 59
SJN 58
дисциплина обслуживания (*продолжение*)
SRT 58
вытесняющая 57, 61
не вытесняющая 57, 61
с несколькими очередями 61
домен 389
дорожка диска 146
драйвер 135
виртуального устройства 367
мини-драйвер 367
секция
завершения 136
запуска 135
продолжения 135
универсальный 367

З

загрузка процессора 64
загрузочная запись 79, 147, 170, 172, 176
загрузочный блок 179
загрузчик
внесистемный 147
главный 151
начальный 151
системный 155, 170
задача 25, 38, 51
диск-резидентная 35
ОЗУ-резидентная 36, 83
реального времени 69
резидентная 36, 78, 83
запрос
к операционной системе 16, 281
на ввод-вывод 133, 141, 385
на прерывание 135

И

идентификатор
безопасности 196
процесса 34
инициативное устройство 33, 133
интерфейс 13
графический 297
пользовательский 17
прикладного программирования 281, 298
программный 17
исключение 131
исполняющая система 384

К

канал
ввода-вывода
программный 134
прямой доступ к памяти 134
связи 242

каталог
корневой 166, 167, 169, 173
понятие 166
кластер 167, 191
кольцо защиты 118
коммутатор
задачи 125
перехвата 125
прерывания 125
контекст задачи 34
контроллер домена 389
коэффициент мультипрограммирования 83
критическая секция 214
кэширование 157, 158, 185

М

маска доступа 196
менеджер загрузки 152, 155
метафайл 192
механизм
взаимодействия между процессами 342
шлюзов 118
микроядро
OS/2 357
QNX 291, 342, 345
Windows NT 383
определение 289
многозадачность 369
вытесняющая 61
кооперативная 61, 370
многопоточность 38
модель
клиент-сервер 292
Холта 248
модуль 279, 377
монитор Хоара 237
мультипрограммирование 27, 29, 47
мьютекс 229

О

образ процесса 314
обслуживание
бесприоритетное 55
приоритетное 55
оверлейная структура 77
ожидание
активное 223
пассивное 224
операционная система 13
макроядерная 280, 292
микроядерная 280, 289
монолитная 280, 292
операционная среда 16, 114
ОСРВ (операционная система реального времени) 48
ОСРВ QNX 291
очередь сообщений 244, 349

П

пакет запросов на ввод-вывод 385
память
виртуальная 73
сегментная 87, 106
сегментно-страничная 97
страничная 93, 110
выделенная 394
зарезервированная 394
оперативная 72
реальная 73
физическая 73
пейджер 355
передача сообщений 348
перенаправление ввода-вывода 322
планирование
вычислительных процессов 52
упреждающее 372
планировщик 370
квантования 372
основной 372
плоская модель памяти 113, 374
подкаталог 166
полоса каталогов 180
порт 349, 356
поток
выполнения 37, 348
данных 193, 322, 352
почтовый ящик 240
правила использования шлюзов 119, 121
прерывание 18, 122
асинхронное 19, 123
внешнее 19, 123
внутреннее 19, 123
маскируемое 21, 123
немаскируемое 123
программное 21
синхронное 19, 123
принтер 137
локальный 137
сетевой 137
приоритет
абсолютный 56
динамический 55, 65
постоянный 55
сна 66
пробуксовка 95
программа
абсолютная двоичная 75
нативная 114
программный канал 331
программный модуль 279
диск-резидентный 280
повторно входимый 46
привилегированный 44
реентерабельный 44, 139, 279
транзитный 280

пропускная способность процессора 64
профиль 366
процесс 37, 369, 392
  внешний 144
  внутренний 144
  вычислительный 25
  легковесный 37
  писатель 232
  последовательный 25
  читатель 232
пул 213

P

рабочее множество 96
раздел
  диска 146
    активный 147, 150
    первичный 150
    расширенный 150
  памяти 82
разделение времени 320
разрешения NTFS 194
  индивидуальные 197, 203
  основные 204
  специальные 199, 205
  стандартные 198
распределение памяти
  неразрывное 82
  разрывное 86
расширенные атрибуты 177
регистр
  задачи 36
  флагов 123
редиректор 342, 343, 385
режим
  ввода-вывода 134
    обмен с прерываниями 134, 135
    опрос готовности 134, 135
  виртуальный 113
  защищенный 102, 105, 106, 124
  мультизадачный 27, 36, 37, 47
  мультипрограммный 27, 36, 47
  отложенной записи 157, 185
  пользователя 116, 131
  привилегированный 29, 131, 280
  работы процессора 29, 280
  разделения времени 29
  реальный 102, 105, 122
  супервизора 116, 131, 280
ресурс 26
  виртуальный 72
  критический 210
  повторно используемый 248
  потребляемый 248
  программный 43
  расходуемый 248
  системный 248

C

сбалансированное двоичное
    дерево 178, 182, 190
свопинг 84, 89
сегмент
  кода 102, 108
  логический 87
  подчиненный 118
  состояния задачи 36, 51, 105, 107, 127
сегментная адресация 102
сектор диска 146
селектор
  дескриптора 118
  сегмента 107
семафор 224
семафорный примитив
  P 225, 228
  V 225, 228
сервер 385
сети Петри 255
  графическое представление 257
  теоретико-множественное представление 255
сигнал готовности 135
система
  многопользовательская 29
  мультитерминальная 48
  реального времени (СРВ) 293
  управления файлами 164, 165
  файловая 163
    FAT 164, 167
    FAT12 165
    FAT16 167
    FAT32 172, 174
    HPFS 177
    NTFS 188
    super-FAT 165
    VFAT 172, 173
    монтируемая 164, 185, 188
системный вызов 16, 78
  exec 319
  fork 319
системный вызов 320
скрипт 362
состояние
  активное 31, 210
  безопасное 261
  блокирования 32
  выполнения 31
  готовности к выполнению 32
  ненадежное 265
  ожидания 32
  пассивное 31
  пользовательское 320
  системное 320
  тупика 247, 261, 268
  фиксированное 272

список
битовых карт 179
дефектных блоков 180
управления доступом 180, 190
дискреционный 196
системный 196
спул-файл 137
спулинг 136
страница
абсолютно чистая 374
виртуальная 93
грязная 374
испорченная 374
физическая 93
чистая 374
страничный кадр 392
страничный файл 93
стратегия
обслуживания 53
планирования 53
супервизор 16, 29, 77
ввода-вывода 132, 142
задач 132, 142, 281
прерываний 22, 23
суперпользователь 316

Т

таблица
векторов прерывания 79, 122
виртуальных логических устройств 140
дескрипторов
прерываний 125
сегментов 87, 104
каталога таблиц страниц 105, 111
логических дисков 151
оборудования 139
прерываний 140
разделов 147
размещения файлов 164, 166, 167
сегментов 87
страниц 94
тег 99
теорема о тупике 269
том 170, 179, 191
тупиковая ситуация 226, 235, 247, 257

У

уплотнение памяти 86
упреждающее чтение 158
уровень
безопасности 288, 392
уровень (*продолжение*)
привилегий 116, 382
задачи 117
запрашиваемый 117
сегмента 116
текущий 116
эффективный 117
условие
взаимного исключения 254, 264
кругового ожидания 254, 264
ожидания 254, 264
отсутствия перераспределения 254, 264
установка тайм-аута 135
учетная запись 388
группы 388
пользователя 388

Ф

файл 163
каталог 166
подкачки 93, 393
страничный 93
файловая система 165
файловая ссылка 193
файловый объект 167, 195
файловый узел 180, 185
фрагментация
памяти 84
файлов 169, 183
функции
API 300
библиотечные 286
микроядра 290
системные 14, 16, 290

Ц

циклическое сканирование 161, 186
цилиндр диска 146

Ш

шлюз 118
задачи 125
перехвата 125
прерывания 125

Э

экстент 178, 181

Я

ядро операционной системы 77, 279

Гордеев Александр Владимирович

Операционные системы: Учебник для вузов

2-е издание

Заведующий редакцией	*А. Кривцов*
Ведущий редактор	*Ю. Суркис*
Литературный редактор	*А. Жданов*
Художник	*Н. Биржаков*
Иллюстрации	*М. Шендерова*
Корректоры	*С. Беляева, И. Тимофеева*
Верстка	*А. Келле-Пелле*

Подписано в печать 16.03.07. Формат 70×100/16. Усл. п. л. 33,54.
Доп. тираж 3000 экз. Заказ № 490.
ООО «Питер Пресс». 198206, Санкт-Петербург, Петергофское шоссе, д. 73, лит. А29.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 953005 — литература учебная.
Отпечатано по технологии CtP в ОАО «Печатный двор» им. А. М. Горького.
197110, Санкт-Петербург, Чкаловский пр., 15.