

Д. В. Иртегов

ВВЕДЕНИЕ В ОПЕРАЦИОННЫЕ СИСТЕМЫ

2-е издание

- Управление памятью и загрузка программ
- Виртуальная память
- Внешние события
- Управление временем центрального процессора
- Межпроцессное взаимодействие
- Управление внешними устройствами
- Файловые системы
- Вопросы безопасности



УЧЕБНОЕ ПОСОБИЕ

bhv®

Д. В. Иртегов

ВВЕДЕНИЕ В ОПЕРАЦИОННЫЕ СИСТЕМЫ

2-е издание

Рекомендовано учебно-методическим объединением вузов
по университетскому политехническому образованию
в качестве учебного пособия для студентов, обучающихся по направлению
230100 "Информатика и вычислительная техника"

Санкт-Петербург
«БХВ-Петербург»

2008

УДК 681.3.066
ББК 32.973.26-018.2
И77

Иртегов Д. В.

И77 Введение в операционные системы. — 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2008. — 1040 с.: ил. — (Учебное пособие)

ISBN 978-5-94157-695-1

Описаны архитектуры современных операционных систем семейств UNIX, Win32, OS/2, VAX/VMS и др. Дано краткое введение в аппаратное обеспечение. Рассмотрены системы команд, загрузка программ, управление памятью, адресация, внешние события, многозадачность, синхронизация, обработка транзакций, внешние устройства и управление ими, файловые системы, безопасность и другие вопросы, обязательные для подготовки специалистов в области информатики и вычислительной техники.

Во втором издании во все разделы добавлены контрольные вопросы, а также включен ряд новых тем, в том числе глава "Введение в обработку транзакций".

Для студентов технических вузов, аспирантов и инженеров

УДК 681.3.066
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	Екатерина Кондукова
Зам. главного редактора	Евгений Рыбаков
Зав. редакцией	Григорий Добин
Редактор	Екатерина Капалыгина
Компьютерная верстка	Ольги Сергиенко
Корректор	Людмила Минина
Дизайн серии	Игоря Цырульникова
Оформление обложки	Елены Беляевой
Зав. производством	Николай Тверских

Рецензенты:

Пугачев Е. К., к.т.н., доцент кафедры "Компьютерные системы и сети" МГТУ им. Н. Э. Баумана

Тормасов А. Г., к.ф.-м.н., доцент, заместитель заведующего кафедрой информатики МФТИ

Кочеев А. А., к.ф.-м.н., доцент, проректор НГУ по информатизации

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 29.10.07.

Формат 70×100^{1/16}. Печать офсетная. Усл. печ. л. 83,85.

Тираж 2500 экз. Заказ № 4406

"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

Оглавление

Благодарности.....	11
Введение.....	13
Основные функции операционных систем.....	15
Какие бывают ОС.....	17
Системы с загрузчиком и без него.....	17
Управление оперативной памятью.....	19
Управление временем центрального процессора.....	27
Семейства операционных систем.....	38
Выбор операционной системы.....	39
Открытые системы.....	43
Как организована эта книга.....	44
Глава 1. Представление данных в вычислительных системах.....	47
1.1. Введение в двоичную арифметику.....	48
1.2. Представление рациональных чисел.....	53
1.3. Представление текстовых данных.....	58
1.4. Представление изображений.....	60
1.5. Представление звуков.....	64
1.6. Упаковка данных.....	64
1.7. Контрольные суммы.....	69
1.8. Введение в криптографию.....	72
Глава 2. Машинные языки.....	79
2.1. Системы команд.....	82
2.2. Форматы команд машинного языка.....	87
2.2.1. Стековые системы команд.....	92
2.3. Команды перехода.....	93
2.4. Регистры.....	95

2.5. Адресация оперативной памяти.....	102
2.6. Режимы адресации.....	106
2.6.1. Вырожденные режимы адресации.....	108
2.6.2. Абсолютная адресация.....	110
2.6.3. Косвенно-регистровый режим.....	111
2.6.4. Косвенно-регистровый режим со смещением.....	114
2.6.5. Базово-индексный режим.....	120
2.6.6. Сложные режимы адресации.....	122
2.6.7. Адресация с использованием счетчика команд.....	122
2.7. Банки памяти.....	124
2.8. CISC- и RISC-процессоры.....	129
2.9. Языки ассемблера.....	132
2.9.1. Многопроходное ассемблирование.....	137
2.10. Отладчики и деассемблеры.....	138
2.10.1. Листинги и отладочная информация.....	139
2.10.2. Отладка.....	146
2.10.3. Удаленная отладка.....	147
2.10.4. Деассемблирование.....	149
Вопросы для самопроверки.....	153
Вопросы, над которыми следует задуматься.....	153

Глава 3. Загрузка программ.....155

3.1. Абсолютная загрузка.....	160
3.2. Разделы памяти.....	161
3.3. Относительная загрузка.....	162
3.4. Базовая адресация.....	167
3.5. Позиционно-независимый код.....	168
3.6. Оверлеи (перекрытия).....	172
3.7. Сборка программ.....	174
3.8. Объектные библиотеки.....	186
3.9. Сборка в момент загрузки.....	188
3.10. Динамические библиотеки.....	193
3.11. Загрузка самой ОС.....	200
Вопросы для самопроверки.....	213

Глава 4. Управление оперативной памятью.....215

4.1. Открытая память.....	216
4.2. Алгоритмы динамического управления памятью.....	220
4.3. Сборка мусора.....	252
4.3.1. Подсчет ссылок.....	255
4.3.2. Просмотр ссылок.....	256
4.3.3. Генерационная сборка мусора.....	259
4.4. Открытая память (продолжение).....	265
4.4.1. Управление памятью в MacOS и Win16.....	270

4.5. Системы с базовой виртуальной адресацией.....	273
Вопросы для самопроверки.....	278
Глава 5. Сегментная и страничная виртуальная память.....	281
5.1. Сегменты, страницы и системные вызовы.....	289
5.2. Взаимно недоверяющие подсистемы.....	308
5.3. Сегменты, страницы и системные вызовы (продолжение).....	314
5.4. Разделяемые библиотеки.....	315
5.5. Страницочный обмен.....	327
5.5.1. Поиск жертвы.....	329
5.6. Управление swap-файлом.....	335
5.7. Одноуровневая память.....	338
Вопросы для самопроверки.....	345
Глава 6. Компьютер и внешние события.....	349
6.1. Опрос.....	350
6.2. Канальные процессоры и прямой доступ к памяти.....	352
6.3. Прерывания.....	356
6.4. Исключения.....	359
6.5. Многопроцессорные архитектуры.....	360
Глава 7. Параллелизм с точки зрения программиста.....	375
7.1. Формулировка задачи.....	378
7.2. Примитивы взаимоисключений.....	381
7.2.1. Мертвые и живые блокировки.....	385
7.3. Примитивы синхронизации с ожиданием.....	391
7.3.1. Семафоры.....	394
7.3.2. Блокировки чтения-записи.....	398
7.3.3. Копирование при записи.....	399
Введение в обработку транзакций.....	401
7.3.5. Захват участков файлов.....	402
7.3.6. Мониторы и серверы транзакций.....	403
7.4. Гармонически взаимодействующие последовательные потоки.....	405
7.5. Системы, управляемые событиями.....	416
Вопросы для самопроверки.....	428
Глава 8. Реализация многозадачности на однопроцессорных компьютерах.....	431
8.1. Кооперативная многозадачность.....	432
8.2. Вытесняющая многозадачность.....	437
8.2.1. Планировщики с приоритетами.....	450
8.2.2. Честное планирование.....	458
8.2.3. Инверсия приоритета.....	461

8.3. Пользовательские и ядерные нити.....	464
8.4. Монолитные системы и системы с микроядром.....	465
Вопросы для самопроверки.....	471
Глава 9. Внешние устройства.....	473
9.1. Доступ к внешним устройствам.....	475
9.2. Простые внешние устройства.....	478
9.3. Порты передачи данных.....	482
9.4. Шины.....	496
9.5. Устройства графического вывода.....	531
9.6. Запоминающие устройства прямого доступа.....	539
9.6.1. Производительность жестких дисков.....	553
9.6.2. Дисковые массивы.....	555
9.6.3. Сети доступа к дискам.....	558
Вопросы для самопроверки.....	561
Глава 10. Драйверы внешних устройств.....	563
10.1. Функции драйверов.....	569
10.2. Многоуровневые драйверы.....	575
10.3. Защита драйверов.....	581
10.4. Архитектура драйвера.....	586
10.4.1. Введение в конечные автоматы.....	596
10.4.2. Архитектура драйвера.....	602
10.5. Запросы к драйверу.....	608
10.5.1. Синхронный ввод/вывод.....	609
10.5.2. Асинхронный ввод/вывод.....	610
10.6. Сервисы ядра, доступные драйверам.....	612
10.6.1. Автоконфигурация.....	613
10.6.2. Выделение памяти.....	615
10.6.3. Таймеры.....	617
10.6.4. Обмен данными с пользовательским процессом.....	617
10.6.5. Сервисные функции.....	618
10.7. Асинхронная модель ввода/вывода с точки зрения приложений.....	619
10.8. Дисковый кэш.....	623
10.9. Спуллинг.....	627
Вопросы для самопроверки.....	629
Глава 11. Файловые системы.....	631
11.1. Файлы с точки зрения пользователя.....	633
11.1.1. Монтирование файловых систем.....	633
11.1.2. Формат имен файлов.....	637
11.1.3. Операции над файлами.....	639
11.1.4. Тип файла.....	642

11.2. Простые файловые системы.....	647
11.3. "Сложные" файловые системы.....	657
11.4. Устойчивость ФС к сбоям.....	668
11.4.1. Устойчивость к сбоям питания.....	668
11.4.2. Восстановление ФС после сбоя.....	672
11.4.3. Файловые системы с регистрацией намерений.....	677
11.4.4. Устойчивость ФС к сбоям диска.....	691
11.5. Файловые системы с копированием при записи.....	693
Вопросы для самопроверки.....	698

Глава 12. Обработка ошибок и исключений.....**701**

12.1. Типы ошибок, которые следует обрабатывать.....	704
12.2. Стратегии обработки ошибок.....	707
12.2.1. Автоматический перезапуск.....	708
12.2.2 Неявное освобождение ресурсов.....	710
12.2.3. Оповещение пользователя.....	713
12.2.3. Логи.....	716
12.3. Обнаружение ошибок.....	721
12.4. Передача информации об ошибках.....	726
12.4.1. Коды ошибок.....	730
12.4.2. Простые обработчики ошибок.....	733
12.4.3. Стековые обработчики ошибок и простая обработка исключений.....	738
12.4.4. Обработка исключений в стиле PL/I.....	747
12.4.5. Исключения C++/Java/C#.....	750
Вопросы для самопроверки.....	754

Глава 13. Безопасность.....**755**

13.1. Формулировка задачи.....	756
13.2. Сессии и идентификаторы пользователя.....	765
13.3. Аутентификация.....	767
13.4. Авторизация.....	772
13.4.1. Списки контроля доступа.....	773
13.4.2. Полномочия.....	783
13.4.3. Изменение идентификатора пользователя.....	787
13.5. Ресурсные квоты.....	793
13.6. Типичные уязвимые места.....	794
13.7. Ошибки программирования.....	797
13.7.1. Срыв буфера.....	797
13.7.2. Внедрение скриптов и SQL.....	805
13.7.3. Другие ошибки.....	807
13.8. Троянские программы.....	809
13.8.1. Троянские программы, внедряемые при разработке.....	813
13.8.2. Троянские программы, внедряемые при распространении.....	820

13.8.3. Троянские программы, внедряемые в уже установленную систему	821
13.9. Практические рекомендации.....	837
Вопросы для самопроверки.....	841
Приложение 1. Обзор современных ОС.....	843
П1.1. MVS, OS/390, z/OS.....	843
П1.2. ОС для компьютеров DEC.....	845
П1.2.1 PDP-6 MONITOR и TOPS-10.....	846
П1.2. BBN TENEX и TOPS-20.....	849
П1.2.3. RT-11.....	851
П1.2.4. RSX-11.....	854
П1.2.5. VAX/VMS.....	857
П1.2.6. OpenVMS.....	860
П1.3. Семейство Unix.....	862
П1.3.1. Распространение UNIX.....	866
П1.3.2. Микроядро.....	869
П1.3.3. Minix.....	869
П1.3.4. GNU Not Unix.....	870
П1.3.5. Open Software Foundation.....	871
П1.3.6. X/Open.....	871
П1.3.7. UNIX System V Release 4.....	872
П1.3.8. Linux.....	876
П1.3.9. MacOS X.....	877
П1.4. Семейство CP/M.....	881
П1.4.2 MS DOS.....	883
П1.4.2. Расширители DOS.....	885
П1.4.3. Win16.....	887
П1.4.4. OS/2 1.x.....	890
П1.4.5. IBM OS/2.....	893
П1.4.6. Windows NT/2000/XP.....	899
П1.4.7 Microsoft Xbox.....	911
П1.4.8. Windows 95/98/ME.....	913
П1.4.9. Windows CE.....	915
Приложение 2. Архитектура и язык ассемблера x86.....	917
П2.1. История.....	918
П2.2. Архитектура процессора.....	926
П2.2.1. Регистры.....	927
П2.2.2. Адресное пространство и диспетчер памяти.....	934
П2.2.3. Прерывания и исключения.....	947
П2.3. Система команд и режимы адресации.....	951
П2.4. Язык ассемблера.....	982
П2.4.1. Синтаксис.....	983

П2.4.2. Символы.....	983
П2.4.3. Секции.....	986
П2.4.4. Команды.....	987
П2.4.5. Директивы.....	989
Список источников информации.....	995
Документация.....	995
INMOS.....	995
National Bureau of Standards.....	995
NASA.....	995
US Army Ordnance Dept.....	995
Zortech.....	996
Книги и публикации.....	996
Интернет-ресурсы.....	1003
Adobe.....	1003
ANSI.....	1003
Atmel.....	1003
IBM.....	1003
IETF.....	1004
Intel.....	1005
Linux.....	1005
Microchip.....	1005
Microsoft.....	1006
National Semiconductors.....	1006
SPARC.....	1006
Sun Microsystems.....	1007
Другие ресурсы.....	1007
Предметный указатель.....	1015

*Посвящается
памяти моего
научного руководителя
И. В. Поттосина*

Благодарности

Жене Ирине за помощь в подготовке рукописи и моральную поддержку.

Большакову Т. Б., Непейводе Н. Н., Тупицину В. Г. и многим другим за обсуждение и ценные советы.

Студентам кафедры ФТИ физфака НГУ и студентам факультета информационных технологий НГУ, на которых в течение 12 лет "обкатывались" содержание и стиль изложения материала.

Введение

Есть хорошая традиция начинать книгу с определения предмета, которому она посвящена. К сожалению, в данном случае я не могу следовать этой традиции, потому что мне не известно хорошего, общепринятого и приемлемо краткого определения *операционной системы* (ОС). Далее я попытаюсь дать определение ОС через список функций, которые на нее обычно возлагаются, но необходимо помнить, что многие из программных продуктов, предлагаемых на рынке в качестве ОС, не выполняют (а иногда выполняют, но так, что лучше бы и не брались совсем) некоторые из этих функций.

В работе [Barron 1971] операционная система определяется так: "Я не знаю, что это такое, но всегда узнаю ее, если увижу". Эта фраза была сказана в первой половине 70-х, когда операционные системы действительно отличались большим разнообразием структуры и выполняемых функций.

С тех времен положение мало изменилось. С одной стороны, системы "общего назначения" — Unix, Windows XP, z/OS похожи друг на друга настолько, что доходит до анекдотов: например, в 1998 году OS/390 прошла тесты консорциума X-Open и отныне может на законных основаниях называться UNIX™.

С другой стороны, оставив в стороне "универсальные многопользовательские компьютеры (которые, в действительности, всего лишь специализируются в той же области, что и компьютеры первых поколений), мы обнаруживаем почти ту же картину, о которой писалось в [Barron 1971], и даже хуже. Только среди широко известных программных продуктов (как коммерческих, так и свободно распространяемых) можно насчитать не менее десятка программ, называемых ОС, и при этом, на первый взгляд, имеющих друг с другом довольно мало общего. В примерах кода для микроконтроллеров PIC есть ассемблерный листинг на полторы страницы, носящий (и не без оснований!) гордое наименование Операционной Системы Реального Времени [www.microchip.com AN585]. Те, кто постарше, могут вспомнить и еще один

программный продукт, на "обычные" ОС внешне и функционально мало похожий, но сообщающий при загрузке: "Loading Doom Operating System" (Загружается Операционная Система Doom).

Споры о том, является ли операционной системой Windows 95/98/ME, не утихают с 1993 года, времени появления первых бета-версий того, что тогда еще называлось Chicago (возможно, масла в огонь этих споров добавил тот факт, что прямой предок Chicago, Windows 3.x, никогда не назывался ОС ни в документации, ни в рекламных текстах), так что, как видим, не все могут узнать ОС, даже увидев ее.

Дополнительную путаницу вносит то обстоятельство, что в наше время все менее и менее понятным становится ответ на вопрос, что же такое компьютер. Определение ликбезовских курсов, согласно которому компьютер состоит из системного блока, монитора, клавиатуры и манипулятора "мышь", естественно, никуда не годится — под него не подходят даже портативные компьютеры или монтируемые в стойку серверы с IBM PC-совместимой архитектурой. Если же попытаться определить компьютер как устройство, имеющее где-то внутри полностью программируемый процессор фон-неймановской архитектуры, в число "компьютеров" попадет неожиданно много устройств.

Во время частной дискуссии о том, следует ли учить студентов программированию на ассемблере, я предложил собеседнику посчитать находящиеся в комнате компьютеры: "телефизор — раз, часы какие? Электронные? — два..." Если бы в момент дискуссии у меня был сотовый телефон, я бы непременно предложил сосчитать и его. На самом деле, с телевизором и часами я отчасти лукавил, а отчасти рисковал ошибиться — современные программируемые логические матрицы тоже позволяют делать чудеса — но сотовый телефон, несомненно, относится к разряду полностью программируемых устройств.

В более сложном оборудовании этот аспект виден еще ярче. Разработка программного обеспечения бортовых компьютеров современного самолета уже давно стоит дороже разработки планера. Шутка, что истребитель — это специализированное периферийное устройство для программ управления огнем, конечно, содержит в себе долю щутки.

В наше время, без преувеличений, компьютером в указанном ранее смысле может оказаться практически любое устройство, включаемое в розетку и/или имеющее автономный источник электропитания. Но, повторюсь, может и не оказаться — о упомянутых ранее успехах технологии программируемых логических матриц забывать тоже не следует.

Возвращаясь к теме нашей книги, напомню, что многие из этих компьютеров работают под управлением того или иного продукта, позиционируемого на рынке как операционные системы: QNX, VxWorks и др. Существуют про-

граммы, разработанные полностью самостоятельно, без использования чужого кода, но некоторые из них модули также называются ОС. Бывает и так, что отдельные подсистемы программного комплекса, не называясь ОС, все-таки являются ею.

Возникают и совсем анекдотические ситуации, когда разработчик пишет свою собственную ОС, не подозревая об этом. Это случается не только при разработке встраиваемых приложений¹, но и при написании программ (особенно многопоточных и требовательных к ресурсам) для компьютеров и ОС общего назначения.

Ничего страшного в этом нет, слава богу, сейчас не советское время, когда постановлением ЦК была запрещена разработка любых компьютеров, не совместимых с сериями ЕС и СМ (разработчики цифровой аппаратуры, чья карьера пришла на конец 70-х и начало 80-х рассказывали, как по поводу каждого цифрового контроллера приходилось доказывать в *первом отделе*, что он не является ЭВМ).

Обидно только будет, если при написании такой "ОС" программист начнет изобретать велосипед и/или не воспользуется сервисами существующих систем по незнанию.

Основные функции операционных систем

Основные функции операционной системы сводятся к управлению ресурсами компьютера, в первую очередь оперативной и энергонезависимой памятью и временем центрального процессора, но также и периферийными устройствами. Управление состоит в том, что прикладная программа запрашивает у ОС ресурсы, а система находит и предоставляет их программе — либо сообщает, что данный ресурс недоступен.

Некоторые системы ограничиваются только распределением ресурсов, т. е. отслеживанием того, занят ресурс или свободен, и обработкой заявок на выделение и освобождение. Некоторые другие системы также гарантируют, что пользовательская задача обращается только к выделенным ей ресурсам, а не к чужим. Как мы увидим далее в книге, обеспечение этих гарантий полезно для защиты пользовательских программ друг от друга — как от ошибок в программах, так и от злонамеренных или ошибочных действий пользователей. Мы увидим также, что в большинстве случаев такие гарантии не могут быть

¹ Калька с *embedded application* — речь идет о микропроцессорах и микроконтроллерах, используемых в качестве управляющих устройств и включенных в конструкцию управляемого объекта. Бортовые компьютеры самолета, схема управления впрыском топлива у автомобиля, паруручные часы и т. д.

обеспечены только программными средствами и требуют специальной поддержки со стороны аппаратуры центрального процессора.

Разумеется, системы первого типа (по причинам, которые мы увидим далее в этом разделе, мы будем называть их ДОС) гораздо проще и требуют гораздо меньших ресурсов (тех же самых времени процессора и оперативной памяти) для своей работы. С другой стороны, такие системы применимы только в однопользовательских компьютерах, при работе на которых пользователь не сталкивается с возможными злонамеренными действиями других пользователей. Впрочем, даже в однопользовательской системе ошибки в прикладных программах могут приводить к порче кода и данных других задач, к потере данных, необходимости перегружать систему и понижению продуктивности работы пользователя, поэтому в последнее десятилетие ДОС постепенно вытесняются даже с однопользовательских настольных компьютеров.

Некоторые системы также предоставляют отдельным задачам или группам задач гарантии, что эта задача всегда будет получать определенное количество ресурсов. Потребность в таких гарантиях наиболее очевидна для многопользовательских систем, совместно используемых несколькими организациями — при этом доля ресурсов, предоставляемых задачам каждой из организаций, должна быть пропорциональна доле средств, которые эта организация внесла в приобретение системы и/или платит за ее эксплуатацию. Интерес к такому режиму использования вычислительных систем неоднократно и по разным причинам возникал и ослабевал.

Одним из важнейших типов гарантий, которые интересны далеко не только системам коллективного пользования, является так называемый режим реального времени, при котором пользовательская программа получает управление в течении гарантированного (и, как правило, достаточно небольшого) времени после возникновения того или иного внешнего события. Такой режим наиболее важен для вычислительных систем, управляющих промышленным или исследовательским оборудованием.

По современным представлениям ОС должна уметь делать следующее.

- Обеспечивать загрузку пользовательских программ в оперативную память и их исполнение (этот пункт не относится к ОС, предназначенным для прошивки в ПЗУ).
- Обеспечивать управление памятью. В простейшем случае это указание единственной загруженной программе адреса, на котором кончается память, доступная для использования, и начинается память, занятая системой. В многопроцессных системах это сложная задача управления системными ресурсами.
- Обеспечивать работу с устройствами долговременной памяти, такими как магнитные диски, ленты, оптические диски, флэш-память и т. д. Как пра-

вило, ОС управляет свободным пространством на этих носителях и структурирует пользовательские данные в виде файловых систем.

- Предоставлять более или менее стандартизованный доступ к различным периферийным устройствам, таким как терминалы, модемы, печатающие устройства или двигатели, поворачивающие рулевые плоскости истребителя.
- Предоставлять некоторый пользовательский интерфейс. Слово "некоторый" здесь сказано не случайно — часть систем ограничивается командной строкой, в то время как другие на 90% состоят из интерфейсной подсистемы. Встраиваемые системы часто не имеют никакого пользовательского интерфейса.

Существуют ОС, функции которых этим и исчерпываются. Одна из хорошо известных систем такого типа — *дисковая операционная система MS-DOS*.

Более развитые ОС предоставляют также следующие возможности:

- параллельное (или псевдопараллельное, если машина имеет только один процессор) исполнение нескольких задач;
- организацию взаимодействия задач друг с другом;
- организацию межмашинного взаимодействия и разделения ресурсов;
- защиту системных ресурсов, данных и программ пользователя, использующихся процессов и самой себя от ошибочных и зловредных действий пользователей и их программ;
- аутентификацию (проверку того, что пользователь является тем, за кого он себя выдает), авторизацию (проверка, что тот, за кого себя выдает пользователь, имеет право выполнять ту или иную операцию) и другие средства обеспечения безопасности.

Какие бывают ОС

В зависимости от используемых стратегий управления ресурсами, можно выделить несколько классов операционных систем. Впрочем, первая из перечисляемых далее классификаций, строго говоря, не относится к управлению ресурсами.

Системы с загрузчиком и без него

Большинство систем, предназначенных для работы на компьютерах, имеющих энергозависимую память и жесткий диск, имеют встроенный загрузчик для пользовательских программ. Функции этого загрузчика обычно не сво-

дятся к простому копированию программы с диска в ОЗУ, а несколько более сложны. Эти функции рассматриваются в главе 3.

Некоторые другие системы предназначены для работы на компьютерах с энергонезависимой программной памятью, обычно ПЗУ или флеш-памятью. При включении такого компьютера ОС и некоторая смесь приложений уже находится в памяти и может исполняться немедленно. Такие компьютеры широко используются во *встраиваемых приложениях* (*embedded application*), т. е. в качестве управляющих устройств в самых разнообразных ситуациях: в бытовой технике, промышленном оборудовании, транспортных средствах и т. д. В последние годы все более широкое распространение получают карманные компьютеры (PDA, смартфоны и т. д.), устроенные по такому принципу.

Перепрошивка ПЗУ подобных компьютеров обычно осуществляется внешними средствами, без участия собственного процессора устройства. Старые конструктивы предполагали изъятие ПЗУ из платы устройства — для этого ПЗУ монтировалось в специальный разъем, а не припаивалось к плате — и его перезапись ("программирование") в специальном устройстве — *программаторе*. Современные устройства все чаще и чаще допускают *внутрисхемную перепрошивку* (*in-circuit programming*).

При работе в таких условиях прошитая в ПЗУ часть ОС вовсе не нуждается в загрузчике. Впрочем такие ОС вынуждены иметь некий отдельный программный комплекс, который компонует образ системы вместе с приложениями и готовит его для программатора. Этот комплекс называется *средой кросс-разработки*, и обычно включает в себя не только компиляторы и компоновщик, но и эмулятор целевой системы, позволяющий отлаживать программу на рабочей станции разработчика, не перепрошивая тестовое устройство.

Производители некоторых маломощных процессоров и особенно микроконтроллеров предлагают также своеобразные устройства, так называемые *внутрисхемные эмуляторы*. Внутрисхемный эмулятор представляет собой устройство, соединяющее настольный компьютер разработчика с отладочной платой. Разъем, который подключается к плате, совпадает по размерам и распайке контактов с микросхемой процессора. На настольном компьютере работает эмулирующая программа, которая имитирует исполнение отлаживаемой программы и выставляет соответствующие напряжения на контактах. Таким образом, разработчик может сочетать возможности, предоставляемые эмулятором (например, пошаговую отладку), и одновременно смотреть, как программа будет себя вести в реальном, а не эмулируемом устройстве. Разумеется, внутрисхемная эмуляция требует, чтобы процессор настольного компьютера был намного мощнее процессора отлаживаемого устройства.

ОС для более производительных процессоров нередко включают в себя — или дают возможность включить в отладочную прошивку — средства для удаленной отладки приложения на тестовой плате. Средства для взаимодействия с управляющим компьютером при такой отладке отличаются большим разнообразием, от последовательных портов RS232 до Fast Ethernet и протокола TCP/IP.

В последнее время появился интерес к модульным ОС, которые могут работать как в качестве ОС общего назначения, так и прошиваться в ПЗУ. Разумеется, состав и конфигурация прошиваемого в ПЗУ ядра системы значительно отличается от того, что работает на настольном компьютере, но программные интерфейсы ОС остаются совместимыми. Это несколько упрощает процесс разработки: некоторые этапы отладки приложения можно осуществлять без всякого эмулятора. Кроме того, это позволяет переносить в ПЗУ некоторые приложения, первоначально созданные для работы на настольных компьютерах. Примерами модульных систем являются QNX (который поставляется как в виде станции разработчика с графическим интерфейсом, файловой системой и другими характерными атрибутами настольной ОС, так и в виде модулей ядра, из которых разработчик может собрать необходимый для его приложения комплект для прошивки в ПЗУ), Embedded Linux, Embedded Windows XP. Впрочем, продолжается и разработка специализированных ОС для встраиваемых приложений, таких как Symbian, Palm OS, Windows CE.

Далее в этой книге я буду называть ОС, имеющие встроенный загрузчик, *системами общего назначения*, а ОС (в том числе и специализированные конфигурации ОС общего назначения), рассчитанные на прошивку в ПЗУ, — *встраиваемыми системами* (калька с англ. *embedded system*). Эта терминология не вполне точна, например, в том отношении, что карманный компьютер или смартфон вряд ли можно назвать встраиваемым приложением, но она приблизительно соответствует современному маркетинговому делению ОС.

Управление оперативной памятью

Оперативная память представляет собой один из важнейших ресурсов вычислительной системы. В ней хранятся как данные, так и программы, потому неограниченный доступ задачи к памяти других задач позволяет этой задаче произвольно изменять поведение этих других задач. В частности, такой неограниченный доступ делает невозможным создание сколько-нибудь эффективной системы безопасности: злонамеренный пользователь может выдать себя за любого другого пользователя и прочитать любые данные, загруженные в ОЗУ. Кроме того, напрямую вызывая функции доступа к внешним устройствам, злонамеренный пользователь может прочитать любые данные, хранящиеся на устройствах внешней памяти.

Большую опасность также представляет непреднамеренный доступ к памяти других задач, прежде всего непреднамеренная их модификация, обусловленная ошибками программирования. В главе 12 мы рассмотрим примеры подобных ошибок и возможные причины их возникновения. Эти ошибки обычно приводят к нарушению работы задач и системы в целом и потере данных.

ДОС (Дисковые Операционные Системы)

Дисковые операционные системы работают с открытой памятью и не защищают код и данные загруженных задач друг от друга. Как правило, такая система представляет собой некий резидентный набор подпрограмм, совмещенный с загрузчиком программ. Обычно, но не всегда, эти подпрограммы содержат средства для работы с устройствами внешней памяти — дисками и лентами, откуда и происходит название систем данного типа. ДОС загружает пользовательскую программу в память и передает ей управление, после чего программа делает с системой все, что ей заблагорассудится, — в частности, ДОС не препятствует прямому доступу к контроллерам дисковых устройств в обход своих процедур. Доступ к некоторым периферийным устройствам вообще не контролируется. При завершении программы считается хорошим тоном оставлять машину в таком состоянии, чтобы ДОС могла продолжить работу. Если же программа приводит машину в какое-то другое состояние, ДОС ничем не может ей воспрепятствовать.

В простейшем случае ДОС является однозадачной системой. Примерами таких систем являются загрузочные мониторы для персональных компьютеров Spectrum, а также для игровых приставок.

Более сложные ДОС представляют собой многозадачные системы, порой даже реализующие многооконный графический пользовательский интерфейс. К таким системам относятся:

- MS/DR-DOS;
- Windows 3.x и (с некоторыми оговорками) Windows 95/98/ME;
- MacOS вплоть до версии 9;
- многие системы для карманных компьютеров, например PalmOS.

Большинство этих систем являются однопользовательскими и предназначены для работы на терминальных устройствах, персональных или карманных компьютерах; впрочем, есть и примеры серверных ОС такого типа — например, Novell NetWare.

Главным преимуществом ДОС является простота и низкие требования к вычислительным ресурсам, поэтому они особенно популярны на маломощных компьютерах. Развитие вычислительной техники делает ДОС все менее и менее конкурентоспособными, так что, например, с настольных персональных

компьютеров они уже практически вытеснены. Впрочем, желание пользователей продолжать работу с существующими программами, разработанными для ДОС, делает вытеснение старых систем длительным и болезненным процессом.

Распространение карманных компьютеров дало системам класса ДОС еще один шанс. Вычислительные ресурсы карманного компьютера ограничены не столько объемом и стоимостью микросхем процессора и памяти, сколько энергопотреблением, т. е. объемом, весом и стоимостью аккумулятора, способного эту систему прокормить. На время написания книги карманный компьютер под управлением Palm OS мог работать без подзарядки в течении нескольких недель или даже месяца, а время непрерывной работы компьютера под управлением Windows CE измерялось десятком или, в лучшем случае, полутора десятками часов и лишь немногого превосходило время автономной работы хорошего ноутбука.

На самом деле, качественный скачок в эргономике карманного устройства достигается при переходе от двух дней непрерывной работы на одной подзарядке ко многим дням работы. Если аккумулятора хватает только на один-два дня, а тем более на меньшее время, пользователь постоянно сталкивается с риском, что устройство окажется разряженным в тот момент, когда оно более всего необходимо. Если же срок непрерывной работы составляет хотя бы неделю (как у современных сотовых телефонов), пользователь всегда будет иметь полдня или больше на то, чтобы все-таки зарядить аккумулятор. Этот фактор значительно ограничивает применение устройств (как карманных компьютеров, так и смартфонов) на основе Windows CE.

ОС (Операционные системы)

Такие системы используют специальные аппаратные устройства, управляющие доступом к памяти, так называемые *диспетчеры памяти* (см. главу 5).

Как мы увидим в главе 5, такое устройство должно быть частью центрального процессора или, во всяком случае, очень тесно связано с ним.

Диспетчер памяти обеспечивает адресацию памяти, зависимую от контекста, так что разные задачи видят различные адресные пространства и логически не могут обращаться к памяти, которую им не выделяли. Поскольку адресное пространство задачи отображается на реальную память непрямым образом, такую схему доступа к памяти называют *виртуальной памятью*. Кроме разделения адресных пространств, диспетчеры памяти выполняют и ряд других полезных функций, например, управление доступом к адресам, на которые отображены порты периферийных устройств, защиту некоторых областей памяти от записи или исполнения (что позволяет в определенных пределах защитить программу от самой себя) и страничную подкачуку (имитациюope-

ративной памяти большого объема за счет подкачки с диска тех областей памяти, к которым происходит обращение). Реализация всех этих функций подробно обсуждается в главе 5.

Виртуальная память позволяет защитить пользовательские задачи друг от друга и реализовать эффективные средства безопасности. Однако это сопровождается значительным ростом требований к ресурсам. Минимальные требования к оперативной памяти, предъявляемые типичной 32-разрядной ОС, составляют от 4 до 16 Мбайт. Впрочем, на 16-разрядных процессорах есть примеры ОС с гораздо более скромными потребностями. Так, ОС RSX/11 для 16-разрядных миникомпьютеров DEC PDP-11, в минимальной конфигурации могла работать в 32 Кбайтах ОЗУ, а в полной требовала всего 128 Кбайт.

Переключение между контекстами диспетчера памяти и особенно передача данных между такими контекстами (необходимая, например, при передаче параметров системного вызова) требуют больших затрат процессорного времени. Иногда накладные расходы ОС достигают 20% времени центрального процессора.

Диспетчер памяти представляет собой устройство, сравнимое по сложности с центральным процессором, поэтому процессоры с диспетчером памяти значительно превосходят по цене и энергопотреблению аналогичные процессоры без него. Большинство современных микропроцессоров общего назначения, такие как x86 (Intel 80386, 80486, Pentium, Pentium Pro/II, AMD Athlon и т. д.), имеют встроенный диспетчер памяти, реализованный на том же кристалле, что и сам процессор. Однако микропроцессоры, предназначенные для карманных и управляющих устройств, нередко не имеют диспетчера памяти совсем или, как некоторые модели PowerPC, имеют возможность выключать его.

Кроме того, работа диспетчера памяти приводит к замедлению доступа к ОЗУ; кэширование и некоторые другие приемы делают это замедление в среднем не очень-то большим, но в худшем случае виртуальная память ведет к снижению скорости доступа вдвое по сравнению с прямым доступом к памяти.

Так или иначе, но возможности, предоставляемые виртуальной памятью, оказываются очень велики. Теперь ошибка в одной задаче приводит к сбросу только этой задачи или, в крайнем случае, тех задач, которые с ней были тесно взаимосвязаны, но не к падению системы в целом. Это дает неоценимые преимущества для многопользовательских компьютеров, например для сетевых серверов, поэтому в системах коллективного пользования ДОС практически не используются с конца 60-х — начала 70-х годов. Само возникновение деления на ОС и ДОС относится примерно к этому же периоду и происходит от систем DOS/360 и OS/360 для компьютеров серии IBM System 360.

Для некоторых современных ОС существуют надстройки, дополнительно изолирующие группы задач друг от друга и обеспечивающие гарантированное качество обслуживания, независимое управление учетными записями пользователей (так что каждая группа задач работает со своей БД учетных записей) и другими общесистемными данными (например, реестром в системах семейства Win32) и даже независимую "перезагрузку" каждой из таких виртуализованных ОС. Такие надстройки находят применение в системах коллективного пользования, особенно в организациях, предоставляющих услуги Web-хостинга и других *провайдерах приложений* (Application Service Provider, ASP). Общепринятое родовое название для таких надстроек не существует; среди них следует упомянуть разделы z/OS и AIX, а также систему виртуализации сервисов ОС Virtuozzo компании SWsoft для Linux и Windows 2000, которую часто ошибочно называют системой виртуальных машин.

Во второй половине 80-х годов было осознано, что системы класса ОС дают значительные преимущества и для однопользовательских ("персональных") компьютеров. Это привело к появлению на рынке Intel 80286 и 80386 и других микропроцессоров, оснащенных диспетчерами памяти, переносу на настольные микрокомпьютеры миникомпьютерных ОС, таких как Unix, и разработке специализированных "персональных" ОС, таких как IBM/Microsoft OS/2, но вытеснение ДОС с настольных компьютеров не завершено и по сей день.

Как уже отмечалось, такая устойчивость старых систем обусловлена желанием пользователей продолжать работу со старыми прикладными программами.

Переход от ДОС к ОС сопровождается радикальной сменой подхода к работе с периферийными устройствами. ДОС позволяет прикладной программе работать с периферийным устройством (например, контроллером дисплея) непосредственно; даже если ДОС предоставляет сервисы для работы с такими устройствами, нередко оказывается, что эти сервисы недостаточно функциональны или потребляют слишком много ресурсов, так что разработчики приложений часто злоупотребляют предоставленной им свободой, а иногда и действительно оказываются вынуждены делать это. Нередко также случается, что разработчики прикладных программ работают с системными структурами данных и внутренними интерфейсами ДОС в обход документированных интерфейсов (это обычно также мотивируется неадекватностью, неудобством или низкой производительностью документированного API).

С другой стороны, ОС запрещают непосредственный доступ к аппаратуре, а все общение пользовательской программы с ядром ОС происходит через диспетчер системных вызовов. Произвольные обращения к данным ядра или вызов произвольных процедур ядра перехватываются диспетчером памяти и

приводят к ошибкам защиты. Таким образом, типичная программа, предназначенная для исполнения под ДОС, под управлением ОС работать не будет, даже если эти системы используют совместимые форматы загрузочных модулей и если интерфейс системных вызовов ОС похож на документированный интерфейс ДОС (как у DOS/360 и OS/360 или у MS DOS и OS/2). Иногда эта проблема может быть решена за счет использования виртуальных машин, которые будут рассматриваться в следующем разделе, но в общем случае она неразрешима.

Напротив, поскольку ОС вынуждает прикладные программы работать через определенные (но, надо отметить, не обязательно документированные) интерфейсы, то обеспечение совместимости между разными ОС представляет собой теоретически разрешимую задачу. На практике решение этой задачи сталкивается с трудностями, которые обычно обусловлены недостаточной документированностью интерфейса эмулируемой системы.

СВМ (Системы Виртуальных Машин)

Системы виртуальных машин используют промежуточные транслирующие слои для доступа не только к оперативной памяти, но и к внешним устройствам. Отличие от ОС в данном случае состоит в том, что пользовательская программа, работающая под управлением ОС, никогда не обращается к внешнему устройству сама, а использует сервисы ОС (системные вызовы). Таким образом, программа, рассчитанная на работу под управлением ОС, знает, что между ней и устройством находится транслирующий слой.

Напротив, программа, работающая под управлением СВМ, считает, что она работает с периферийными устройствами непосредственно. Транслирующий слой подсистемы ввода/вывода СВМ полностью прозрачен. Для этого процессор должен предоставлять специальные средства, позволяющие перехватывать все обращения к портам ввода/вывода, но такие средства присутствуют практически в любом центральном процессоре, реализующем виртуальную память.

Несколько более сложной задачей при реализации СВМ оказывается виртуализация обращений к диспетчеру памяти. Далеко не каждый диспетчер памяти допускает такую виртуализацию — так, в процессорах линии x86 соответствующие возможности появились только в Pentium Pro/Pentium II. Однако если такая виртуализация реализована на уровне процессора, система виртуальных машин позволяет запускать в качестве задач целые операционные системы.

Применения СВМ разнообразны.

- Запуск нескольких копий ОС, что может быть полезно, если необходимо запустить на одной машине смесь приложений, которая не очень-то ус-

тойчиво работает совместно или по другим причинам не может быть установлена под одним экземпляром ОС (причины мы разберем в разд. 5.4, проблемы такого рода характерны для OS/2 и Windows NT/2000/XP/2003). Поскольку СВМ отличаются довольно высокими накладными расходами, особенно при работе с внешними устройствами, подобные конфигурации редко находят промышленное применение; с точки зрения производительности в таких условиях гораздо привлекательнее системы виртуализации сервисов ОС, такие как Virtuozzo.

- Тестовые комплексы, позволяющие тестировать распределенное приложение, компоненты которого работают под разными ОС на одной машине. Так, разработчик может запустить в одной виртуальной машине Linux с Web-сервером Apache, а в другой — Windows XP с Web-браузером.
- Тестовые комплексы, позволяющие тестировать кросс-платформенное ПО под различными ОС.
- Тестовые комплексы, позволяющие отлаживать модули ядра ОС. Ошибки в модулях ядра обычно приводят к падению всей системы и — с высокой вероятностью — к потере данных, поэтому при разработке модулей ядра на одной машине обычно хранят исходные тексты и осуществляют компиляцию и сборку, а на другой — тестируют собранный модуль. Использование СВМ позволяет совместить эти функции на одной физической машине.
- Запуск унаследованных приложений, предназначенных для работы под устаревшими ОС и особенно ДОС. Например, OS/2 и Windows NT/2000/XP включают специализированные СВМ, позволяющие запускать особую версию MS-DOS и приложения для MS/DR-DOS, в том числе и такие приложения, которые напрямую пытаются работать с аппаратурой компьютера. Разумеется, эти обращения перехватываются СВМ и виртуализуются. DOS-эмулятор OS/2 даже позволяет запускать слегка модифицированную версию Windows 3.11 и большинство приложений Win16; в Windows NT это достигается специальной подсистемой WoW (Windows on Windows), которая не является СВМ в обсуждаемом смысле. Различные — правда, обычно, не очень качественные — эмуляторы MS-DOS доступны также для некоторых систем семейства Unix для процессоров x86 — SCO UnixWare, Linux.

Первая коммерчески успешная СВМ VM была реализована компанией IBM в первой половине 70-х годов для последней из перечисленных целей — для запуска нескольких приложений DOS/360 на машинах System/370. Эта система использовалась (и продолжает использоваться) также для разработки и отладки флагманской ОС для машин этого семейства, MVS и ее современных версий System/390 и z/OS. В соответствии с первоначальными планами VM

считалась переходной системой и должна была к концу 70-х годов обеспечить плавный вывод из эксплуатации приложений для DOS/360. Однако пользователи не захотели следовать этим планам — началась разработка новых приложений для DOS и даже специализированных оболочек, создававших по виртуальной машине для каждой терминальной сессии и обеспечивающих многопользовательскую интерактивную работу. В конце 70-х и в 80-е годы IBM несколько раз пыталась прекратить поддержку VM, но каждый раз это приводило к массовым протестам пользователей, так что поддержка VM под торговой маркой VM/ESA продолжается и по сей день.

После выхода на рынок процессоров Pentium II, предоставлявших возможность виртуализации диспетчера памяти, довольно быстро появился целый ряд универсальных СВМ для машин с этими процессорами: VMWare, Connectix Virtual PC (в 2003 году был приобретен Microsoft) и др. В отличие от VM/ESA, эти СВМ работают в качестве задач под управлением ОС "хозяина" (host operating system), обычно Windows NT и/или Linux. Вместо виртуализации доступа к реальной аппаратуре РС эти СВМ эмулируют наиболее распространенный адаптер жесткого диска (IDE/ATA), популярные видео- и звуковые устройства и т. д. Обращения к этим устройствам преобразуются в обращения к сервисам ОС "хозяина". Такой подход позволяет избежать разработки специализированных виртуализующих драйверов для всей номенклатуры периферийного оборудования, доступного для современных ПК — т. е. значительно удешевляет поддержку СВМ, но, разумеется, приводит к значительным накладным расходам при каждой операции ввода/вывода.

Гетерогенные вычислительные комплексы

В некотором родстве с СВМ находятся системы, позволяющие запускать несколько экземпляров ОС или даже различные ОС на разных процессорах многопроцессорного вычислительного комплекса. Такие системы обычно реализуются в виде модулей ядра соответствующих ОС. Эти модули осуществляют взаимодействие между копиями ОС, перераспределение ресурсов и виртуализацию некоторых наиболее важных ресурсов, таких как сетевая и дисковая подсистемы. Среди примеров подобных конфигураций необходимо упомянуть следующее.

- Использование РС-совместимых процессорных модулей под управлением OS/2 в машинах IBM System/370 и ранних версиях AS/400. Позднее на этих платах поддерживались также Novell Netware и Windows NT. Такие платы использовались для организации взаимодействия больших машин с сетями персональных компьютеров. Во второй половине 90-х годов поддержка соответствующих сетевых протоколов была реализована в OS/390

и OS/400, и необходимость в этих платах отпала, но такие конфигурации эксплуатируются и поддерживаются до сих пор.

- Процессорные модули System/370 и System/390, реализованные в виде плат расширения IBM PC. Этот модуль работает под управлением MVS или System/390 и рассматривает PC, в материнскую плату которой он вставлен, как канал ввода/вывода. Специальные модули ядра ОС, работающей на PC (обычно также OS/2), обеспечивали эмуляцию работы канального процессора. Различные типы подобных устройств выпускались небольшими сериями со второй половины 80-х годов и продолжают выпускаться по сей день. Применяются они, главным образом, для разработки ПО, но иногда и в качестве маломощных машин System/390.
- "Эмуляторы" PC для Apple Macintosh — PC-совместимые процессорные модули, оформленные в виде плат расширения Mac. Такие устройства выпускались многими небольшими компаниями.
- Кластеры Sun Solaris, позволяющие запустить несколько самостоятельных копий Solaris на многопроцессорных серверах Sun Fire. Каждому экземпляру выделяется определенное количество процессоров и ОЗУ; доступ к дисковой подсистеме и сетевым интерфейсам виртуализуется. Для взаимодействия между экземплярами ОС используется виртуальный сетевой интерфейс, реализованный в виде кольцевого буфера в разделенном ОЗУ. Допускается независимая перезагрузка ОС и перераспределение процессоров, ОЗУ и дискового пространства между системами без их перезагрузки — для этого администратор системы должен выполнить соответствующие команды, используя управляющую консоль кластера.
- IBM поддерживает запуск нескольких копий System/390 z/OS и Linux (в том числе и в смеси) на многопроцессорных машинах System/390 и z/9000. Как и в кластерах Sun, процессоры и ОЗУ распределяются, а каналы ввода/вывода виртуализуются; виртуализация обеспечивается программным обеспечением канальных процессоров практически без потери производительности. Допускается независимая перезагрузка экземпляров ОС и перераспределение ресурсов между экземплярами без их перезагрузки. Для связи между экземплярами ОС также используется виртуальный сетевой интерфейс на основе разделенного ОЗУ, обеспечивающий пропускную способность до нескольких сотен гигабит в секунду.

Управление временем центрального процессора

Обратите внимание на терминологическую тонкость — если сущность, запрашивавшую и получавшую оперативную память, в предыдущем разделе я называл задачей, то единица, которая запрашивает время ЦПУ, называется

процессом или нитью. Более формально эти понятия будут введены в *глазах 4 и 8* соответственно. В ряде старых ОС, в том числе в старых системах семейства Unix, понятия задачи и процесса совпадали. Большинство современных систем допускает создание нескольких нитей в рамках одной задачи.

Все известные ОС также можно разделить на два класса в зависимости от того, как они относятся к времени центрального процессора: некоторые системы только передают центральный процессор от задачи к задаче, а другие следят за тем, какое время (или какую долю времени) задача реально использует, и пресекают попытки "чрезмерного" использования ЦПУ. Это деление отчасти аналогично делению на ДОС и ОС при управлении оперативной памятью, но эта аналогия несколько натянута. Более удачная классификация ОС по отношению к времени ЦПУ сложнее и приводится далее в этом разделе.

Однопроцессные (однозадачные) системы

Такие системы вообще не распределяют время ЦПУ. Система передает управление пользовательской программе и получает его назад только во время системных вызовов. Если программа "плохо себя ведет", система может не получить управления никогда. В этом случае пользователю придется останавливать работу приложения каким-то другим способом — многие компьютеры для этого имеют специальную кнопку системного сброса (reset).

Например, у некоторых карманных компьютеров кнопка сброса спрятана на дне специальной дырки в корпусе, и ее надо нажимать разогнутой скрепкой или чем-то в этом роде, что позволяет избежать случайных нажатий. На компьютерах, которые такой кнопки не имеют, сброс обезумевшей задачи и перезагрузку системы приходится осуществлять выключением питания.

Наиболее известными примерами систем данного типа могут быть загрузочные мониторы для машин типа Spectrum. MS/DR-DOS — это однопроцессные, но, строго говоря, не однозадачные системы: они допускают последовательную загрузку нескольких программ и даже, при определенных обстоятельствах, получение управления задачами, отличными от текущей активной.

Кооперативные многопроцессные системы

Такие системы допускают поочередное исполнение нескольких пользовательских процессов, но переключение между процессами происходит по инициативе активного процесса, в результате выполнения специального системного вызова. Как и в однозадачных ОС, если активный процесс управления не отдает, система ничего не пытается (да и не может) сделать. Поэтому, как и в простых однозадачных системах, кнопка системного сброса остается важнейшим элементом пользовательского интерфейса кооперативных ОС.

Широкое распространение получили однопользовательские системы такого типа, реализующие многооконный графический пользовательский интерфейс (GUI, Graphical User Interface). Большинство современных реализаций GUI имеют событийно-ориентированную архитектуру. Выполнение пользователем различных действий, например, нажатие кнопки мыши в пределах окна, приводят к генерации события, которое затем передается для обработки задаче, ответственной за это окно. Хорошим тоном считается быстро обработать событие и дать возможность другим окнам обработать свои события. Программы, которые нарушают данное правило, приводят к задержкам при обработке событий всеми остальными окнами. Пользователь воспринимает это как "зависание" системы и может от расстройства нажать кнопку сброса; чтобы этого не происходило, рекомендуют как-то отображать "задумчивость" на экране, например, придавая курсору мыши форму песочных часов.

Разумеется, такое поведение сильно раздражает пользователей, поэтому единственной причиной, по которой такие ОС получали популярность, являлись их относительно низкие требования к системным ресурсам; причиной же их долголетия были сложности при переделке прикладных программ для работы под управлением более совершенных ОС (с ними мы вкратце познакомились в разд. "*ОС (Операционные системы)*").

Описываемую архитектуру имеют MS Windows до версии 3.11, MacOS до версии 9, Palm OS до версии 4.

Интересным примером многопользовательской кооперативно многозадачной системы является серверная ДОС Novell Netware. В первой половине 90-х годов эта система имела довольно большой успех в качестве выделенного файлового сервера для сетей персональных компьютеров. Во второй половине 90-х, по мере вытеснения файловых сервисов на вспомогательные роли и росту потребности в серверах приложений, эта система была вытеснена в довольно узкую рыночную нишу, в которой, однако, она до сих пор живет и процветает.

Вытесняющие многопроцессные системы

Системы с вытесняющей многозадачностью так же, как и кооперативные, допускают поочередное выполнение нескольких пользовательских процессов (на однопроцессорной машине по-другому невозможно), но переключение между процессами может происходить не только по инициативе активного процесса, но и из-за внешних событий. Например, если в системе есть две нити, одна из которых (низкоприоритетная) исполняется, а другая (высокоприоритетная) ждет пользовательского ввода, то когда этот ввод приходит, система отбирает управление у низкоприоритетной нити и отдает его высокоприоритетной. Кроме того, вытесняющие многопроцессные системы часто

ограничивают время, в течение которого нить может исполняться непрерывно; по истечении кванта времени управление у этой нити отбирают и, если в системе есть другая готовая к исполнению (т. е. не ждущая никакого события) нить, то управление отдается ей.

Разумеется, реализация такой системы требует решения двух задач.

- Система должна получать управление в ответ на внешние события и — при реализации квантов времени — в ответ на сигналы системного таймера. Почти все современные вычислительные системы реализуют это с помощью специального аппаратного механизма *прерываний* (см. разд. 6.3).
- Система должна отобрать управление у активной нити так, чтобы его потом можно было вернуть, т. е. не нарушив ее работу. На практике эта задача гораздо проще, чем она может показаться, потому что архитектуры подавляющего большинства современных процессоров специально оптимизированы для выполнения такой передачи; некоторые процессоры даже имеют специальные команды для такой передачи управления. Этот вопрос подробно обсуждается в разд. 8.2.

В действительности, написание многопоточных программ, рассчитанных на работу в такой среде, создает ряд дополнительных и довольно сложных проблем, обсуждению которых посвящена глава 7. Надо отметить, что многие из этих проблем возникают также при работе на многопроцессорных машинах.

Тем не менее преимущества, предоставляемые вытесняющими многопроцессорными системами, очень велики и настолько важны, что мы посвятим им еще два следующих раздела введения. Первые ОС (или, скорее, все-таки ДОС) с вытесняющей многозадачностью появились в начале 60-х годов вскоре после изобретения аппаратных прерываний; большинство современных и вновь разрабатываемых ОС также используют эту стратегию управления временем центрального процессора. Такие системы находят применение как в однопользовательских (настольных, карманных и др.) компьютерах, так и в многопользовательских серверах и во встраиваемых приложениях, в которых неинтерактивный компьютер управляет каким-то оборудованием, например впрыском топлива в двигатель автомобиля.

Приведенная классификация ОС различает их по тому, каким образом система распределяет время процессора. Возможна и другая классификация, по тому, каких целей пытается достичь ОС, распределяя это время, — данная классификация интересна, если мы определяем, какая ОС более адекватна данному конкретному приложению. С этой точки зрения, наиболее популярные системы можно разделить на три класса.

- Однопользовательские интерактивные системы, которые оптимизируют среднее время реакции на внешнее событие (чаще всего, ввод пользовате-

ля). В этом качестве используются все три перечисленных ранее класса ОС: однозадачные, кооперативные и вытесняющие многозадачные.

- *Многопользовательские интерактивные системы*, или, что то же самое, *системы разделенного времени* (*shared time*), которые максимизируют количество событий, обрабатываемых в единицу времени. Однозадачные системы этого обеспечить не могут, хотя есть примеры используемых таким образом кооперативных систем — примером служит уже упоминавшаяся Novell Netware. Впрочем, подавляющее большинство систем, претендующих на этот рынок, — вытесняющие многозадачные ОС. Отдельные ОС, например системы семейства Unix, могут использоваться без дополнительных настроек как в качестве многопользовательских, так и в качестве однопользовательских. Некоторые другие системы, например Windows NT/2000, в однопользовательской (Workstation) и многопользовательской (Server) конфигурациях используют разные настройки динамической приоритизации.
- *Системы реального времени* (*real time*) — системы, которые оптимизируют (или, точнее, гарантируют) максимальное время реакции на внешнее событие. Такие системы используются, главным образом, в неинтерактивных приложениях. Дело в том, что пользователя-человека неожиданная задержка при исполнении его запроса может раздражать, но непосредственной опасности эта задержка представлять не может. Напротив, задержка при обработке сигнала от установленного на летательном аппарате датчика опасности столкновения может привести к катастрофическим последствиям.

Системы разделенного времени

Переключение процессов по таймеру в сочетании с некоторыми другими приемами, такими как динамическая приоритизация (рассматривается в *разд. 8.2.1*), позволяет реализовать параллельную работу нескольких пользователей или пользовательских задач так, что каждый пользователь получает определенную долю процессорного времени. Со многих точек зрения каждый из этих пользователей может считать, что ему доступен свой собственный компьютер, производительность которого составляет определенную долю от реального.

Многотерминальные многопользовательские вычислительные системы появились в середине 60-х и получили большое распространение в 70-е годы. Еще в начале 80-х многотерминальные миникомпьютеры, такие как DEC VAX, считались магистральной линией развития вычислительной техники.

Распространение персональных компьютеров и рабочих станций в 80-е годы несколько изменило взгляд на роль многопользовательских систем — был даже период, когда казалось, что одноранговые сети РС могут решить все

задачи, кроме каких-то очень уж специализированных. Впрочем, рост сетей (как корпоративных, так и Интернет) и развитие прикладного программного обеспечения породили потребность в выделенных серверах, сначала файловых (IBM LAN Manager, Novell Netware), а затем и серверах приложений. Разумеется, нельзя также забывать о Web-серверах и сервисах, обеспечивающих сетевую инфраструктуру: маршрутизацию, передачу почты и т. д. Как мы увидим далее, все эти задачи естественное всего возложить на системы разделения времени.

Для сервера приложений выгоды от многозадачности не совсем очевидны. Действительно, исполнение каждого запроса прикладного уровня (например, запроса SQL) требует определенных затрат времени сетевого интерфейса, центрального процессора и дискового контроллера. Поскольку многие сервера имеют один сетевой интерфейс, один процессор и один диск, неясно, за счет чего мы можем получить выигрыш, исполняя несколько запросов квазипараллельно — однако выигрыш есть, и он может быть значительным. Чтобы понять, в чем он состоит, давайте рассмотрим процесс исполнения запроса.

Сначала запрос принимается сетевым интерфейсом.

Затем он анализируется центральным процессором. На самом деле, этот анализ многоэтапный и частично выполняется модулями ядра ОС, а частично самим приложением, — для простоты мы не будем в этом разбираться и, поскольку обработкой запроса заняты именно ЦПУ и оперативная память, мы объединим этот этап процесса в один блок. Также для простоты мы предположим, что время приема запроса и время его первичного анализа одинаковы. Это предположение не очень реалистично, но упрощает мне рисование картинки, а картинка позволит нам понять главную идею.

Так или иначе, обработав запрос, система приходит к выводу, что его исполнение требует обращения к диску. Диск также некоторое время обрабатывает запрос и передает данные в ОЗУ.

Процессор обрабатывает полученные данные, формирует ответ и возвращает его пользователю.

Затем сетевой адаптер некоторое время осуществляет передачу ответа.

Все эти этапы изображены на рис. В1.

Рассмотрим теперь результат квазипараллельного исполнения трех запросов, приходящих друг за другом (рис. В2). Видно, что пока диск передает данные, относящиеся к первому запросу, центральный процессор может анализировать второй запрос, а сетевой интерфейс принимать третий. Впрочем, если соотношение между временами этапов исполнения каждого из запросов именно таковы, как я предположил, то центральный процессор оказывается перегружен — ему следовало бы одновременно обработать входящий третий запрос и сформировать ответ на первый. Возможно, в этой ситуации было бы

целесообразно поставить второй процессор. На практике нередко так и поступают; двухпроцессорные серверы x86 довольно распространены, но PC-совместимые машины с большим количеством процессоров редки и используются лишь в некоторых специфических ситуациях; количество процессоров в серверах приложений достигает 16 штук у машин z/9000, а у серверов Sun Fire — 32 и более.



Рис. В.1. Исполнение одиночного запроса

Следовательно, хотя многопоточность обычно не может улучшить время обработки одного запроса, она вполне способна в несколько раз увеличить количество запросов, обрабатываемых в единицу времени, — а это именно то, что требуется от сервера приложений.

Многотерминальные системы в чистом виде также не ушли со сцены. До сих пор сохраняются унаследованные приложения, работающие на больших компьютерах IBM System/390 и z/9000, рассчитанные на работу с терминалами IBM 3270; в наше время часто используются эмуляторы этих терминалов, работающие на персональных компьютерах и взаимодействующие с центральным компьютером по сети. Основная среда графического пользователь-

ского интерфейса для ОС семейства Unix, X Window с самого начала разрабатывалась как распределенная; компания Sun в настоящее время предлагает многотерминальные программно-аппаратные комплексы Sun Ray, состоящие из "тонких клиентов"-Х-терминалов и центрального сервера Sun SPARC под управлением Solaris. Среди многотерминальных решений на основе более молодых систем семейства Win32 следует назвать Citrix ICA и Microsoft Terminal Server.

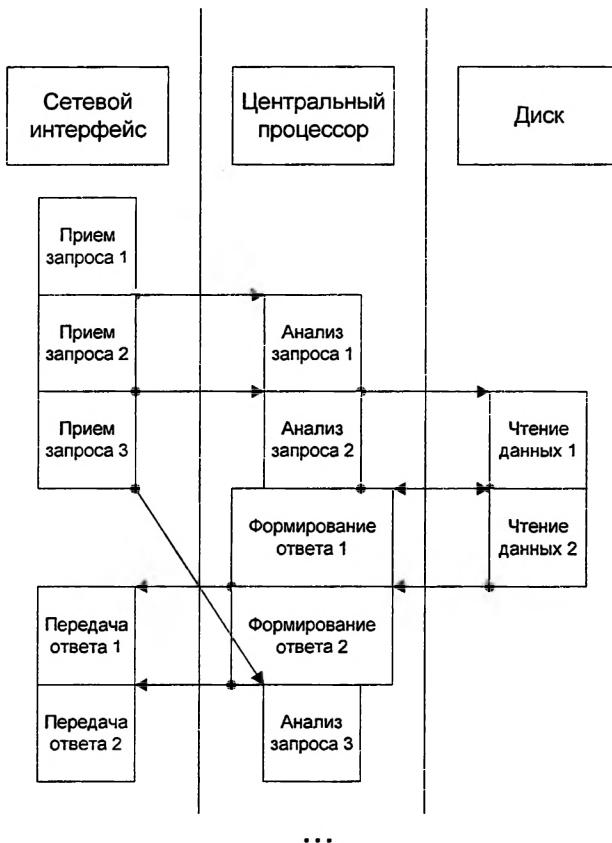


Рис. В.2. Процесс исполнения трех запросов

По сравнению с "толстыми клиентами", т. е. персональными компьютерами, многотерминальные системы имеют как ряд преимуществ, так и ряд недостатков. Перечислим преимущества терминального сервера.

- Централизованный контроль за программным обеспечением и, как следствие, более низкая совокупная стоимость владения (Total Cost of Ownership),

TCO) — вместо того чтобы устанавливать обновленную версию ОС на десятки ПК, достаточно установить ее на один центральный компьютер. Это относится и к установке обновлений ПО, и к его перенастройке.

- Большая, по сравнению с персональными компьютерами, эффективность использования вычислительных ресурсов. Действительно, типичный персональный компьютер большую часть времени простояивает, ожидая действий пользователя. С другой стороны, если поставить на ПК маломощный процессор, это приведет к медленной реакции на действия пользователя, что будет этого самого пользователя сильно раздражать. При нынешних ценах на аппаратное обеспечение нервы пользователя оказываются дороже мощного процессора, поэтому целесообразнее устанавливать на ПК едва ли не самые мощные из доступных на рынке процессоров. Кроме того, каждый ПК должен иметь в памяти свою копию программного обеспечения, с которым работает пользователь. Напротив, в условиях терминального сервера пользовательские запросы оказываются в среднем равномерно распределены по времени, так что процессор того же класса, что стоял бы на ПК, вполне может обеспечить приемлемое среднее время реакции, хотя и при гораздо более высокой средней загрузке. Также, поскольку все пользователи работают с одной и той же ОС и, во многих случаях, с одним и тем же прикладным ПО, можно довольно много сэкономить на оперативной памяти — как за счет памяти, занятой программами, так и за счет разного рода кэшей, буферов и резервов "на всякий случай".
- Относительная простота разработки приложений. В этой книге разработка распределенных сетевых приложений почти не рассматривается, но так или иначе очевидно, что разработать многопоточное приложение, обеспечивающее взаимодействие в пределах одной машины, намного проще, чем распределенное. Поэтому почти все без исключения приложения, обеспечивающие коллективную работу, созданные в 60-е и 70-е годы, рассчитаны именно на многотерминальную работу в режиме разделения времени. Многие из таких приложений эксплуатируются до сих пор.

Недостатки терминального сервера.

- Плохая масштабируемость: даже если машина, мало отличающаяся от серийного ПК, может обслуживать несколько десятков пользователей, она не сможет обслужить сотню. Да, для сотни пользователей можно поставить многопроцессорный сервер с высокопроизводительной дисковой подсистемой и большим объемом ОЗУ, но уже для нескольких сотен пользователей на рынке может просто не оказаться системы, обеспечивающей приемлемую производительность. Разумеется, можно поставить несколько серверов, но при этом потеряется последнее из перечисленных преиму-

ществ — от одномашинного приложения придется переходить к распределенному.

- Высокие требования к сетевой инфраструктуре как по производительности, так и по надежности. В итоге это приводит к повышению стоимости сетевого оборудования и его обслуживания и даже может съесть выигрыш от более эффективного использования аппаратуры и поддержки ПО.
- Полная неприменимость в некоторых ситуациях, например при большом количестве мобильных пользователей с ноутбуками (laptop) и/или пользователей, работающих из дома.

Традиционные системы разделенного времени обеспечивают в среднем справедливое распределение процессорного времени, но не гарантируют точного соблюдения этой справедливости в каждый конкретный момент. Это приемлемо для корпоративных серверов, в которых вычислительные ресурсы оплачиваются непосредственно пользователями, а компанией, на которую они работают. Однако для провайдеров приложений эта ситуация нетерпима: заказчик, обнаружив случайную флуктуацию в распределении ресурсов, может возмутиться и спросить, за что он платил деньги. Это приводит к появлению специальных надстроек над базовой ОС, таких как Swsoft Virtuozzo, обеспечивающих гарантированное качество обслуживания, в том числе и строго гарантированную долю процессорного времени.

Системы реального времени

Многие системы разделенного времени обеспечивают удовлетворительно среднее время реакции на внешние события, однако есть приложения, для которых необходимо гарантированное максимальное время такой реакции. Иными словами, ОС, которая в среднем обеспечивает реакцию на событие в течении 0,1 миллисекунды, но допускает — пусть даже изредка — увеличение этого времени до 0,1 секунды, для таких приложений неприемлема. Большинство этих приложений связаны с управлением промышленным и исследовательским оборудованием, а также с управлением бортовым оборудованием транспортных средств и космических аппаратов.

Важно подчеркнуть, что гарантированное время реакции на событие — это не совсем то же самое, что высокая производительность (под которой обычно подразумеваю именно среднее время обработки события). Из дискретной математики и теории алгоритмов известно много примеров алгоритмов, которые обеспечивают хорошие средние показатели, но очень плохо себя ведут в некоторых, пусть даже маловероятных, случаях. Так, классический алгоритм быстрой сортировки очень медленно работает на уже отсортированном массиве; хэш-таблица с низким коэффициентом заполнения обеспечивает константное (независимое от размера таблицы) среднее время поиска, но в

худшем случае, когда все включенные в хэш элементы попали в одну ячейку, время поиска оказывается пропорционально количеству элементов, включенных в таблицу.

Впрочем, при разработке системы реального времени главной проблемой оказываются не "плохие" в указанном ранее смысле алгоритмы, а тот факт, что многие операции ввода/вывода занимают очень большое время. Ядро системы реального времени должно уметь вытеснить само себя и передавать управление пользовательским задачам во время исполнения операций ввода/вывода. Проблемы, к которым это приводит, и решения, применяемые на практике, обсуждаются в разделе 8.2.3. Так или иначе, архитектура наиболее успешных систем РВ, таких как QNX и VxWorks, достаточно сильно отличается от наиболее распространенных систем разделенного времени.

До недавнего времени при разработке приложений РВ часто вообще обходились без ОС, работая на "голой" машине, или использовали какую-либо примитивную однозадачную ДОС в качестве загрузчика и средства управления вспомогательными ресурсами. Нередко такое приложение содержало внутри себя собственный планировщик и собственные средства межпроцессного взаимодействия, т. е. значительную часть полноценной ОС. При разработке приложений для 8-разрядных микроконтроллеров этот подход используется и по сей день.

Любопытно, что новомодное течение в компьютерной технике — multimedia — при качественной реализации предъявляет к системе те же требования, что и промышленные задачи реального времени. В multimedia основной проблемой является синхронизация изображения на экране со звуком. Именно в таком порядке. Звук обычно генерируется внешним аппаратным устройством с собственным таймером, и изображение синхронизируется с ним. Человек способен заметить довольно малые временные неоднородности в звуковом потоке, а пропуск кадров в визуальном потоке не так заметен. Расхождение же звука и изображения фиксируется человеком уже при задержках около 30 мс. Поэтому системы высококачественного multimedia должны обеспечивать синхронизацию с такой же или более высокой точностью, что мало отличается от реального времени.

Так называемое "мягкое реальное время" (soft real time), предоставляемое современными платформами Win32, не является реальным временем вообще, это что-то вроде "осетрины второй свежести". Система "мягкого РВ" обеспечивает не гарантированное максимальное, а всего лишь хорошее среднее время реакции, которое меньше интервала смены кадров видеоизображения или проигрывания блоков аудиопотока. Для мультимедийных приложений и игр различие между "средним" и "гарантированным" не очень критично — нудрится картинка или поплынет звук. Но для промышленных приложений, где необходимо настоящее реальное время, это обычно неприемлемо.

Семейства операционных систем

Часто можно проследить преемственность между различными ОС, необязательно разработанными одной компанией. Отчасти такая преемственность обусловлена требованиями совместимости или хотя бы переносимости прикладного программного обеспечения, отчасти — заимствованием отдельных удачных концепций.

На основании такой преемственности можно выстроить "генеалогические деревья" операционных систем и — с той или иной обоснованностью — объединять их в семейства. Впрочем, в отличие от древа происхождения биологических видов, граф родства ОС не является деревом и нередко содержит циклы, поэтому бесспорной многоуровневой классификации, охватывающей всю техносферу и похожей на линнеевскую классификацию видов, выстроить не удается.

Тем не менее мы с достаточно большой уверенностью можем выделить минимум три семейства ныне эксплуатирующихся ОС и еще несколько — вымерших или близких к тому.

- Системы для больших компьютеров фирмы IBM — MVS, OS/390 и z/OS; несмотря на разницу в названиях, это версии одной и той же ОС.
- Обширное, бурно развивающееся и имеющее трудно определимые границы семейство Unix. В этой книге под системами данного семейства мы будем подразумевать прежде всего ОС трех основных родов:
 - Unix System V Release 4.x: Sun Solaris, SCO Unixware, SGI Irix;
 - Berkeley Software Distribution Unix: BSDI, FreeBSD и другие ветви BSD, а также построенная на ядре FreeBSD MacOS X;
 - Linux.

Кроме этих ОС, в семейство Unix также входят Minix, IBM AIX, HP/UX, Tru64 и ряд менее известных ОС. Некоторые из этих систем (как Minix) разработаны с нуля, другие в той или иной мере основаны на исходных текстах Unix System V Release 3 и/или BSD Unix. Принадлежность к семейству еще некоторых ОС, таких, как QNX — дискуссионный вопрос, обсуждение которого увело бы нас далеко в сторону от темы книги.

- Семейство прямых и косвенных потомков Control Program/Monitor (CP/M) фирмы Digital Research. В этом семействе можно выделить также весьма широко известное подсемейство Win32-платформ (рис. В.3). Сюда относятся три существенно различные по архитектуре линии систем:
 - полноценная ОС с защищенной виртуальной памятью Windows NT/2000/XP/2003 (из маркетинговых соображений во второй половине

90-х годов Microsoft перешел от простой нумерации версий к бессистемным обозначениям; в действительности все перечисленные ОС суть версии одной и той же системы, первоначально называвшейся OS/2 NT; далее в этой книге я буду называть ОС этой линии просто Windows NT, используя коммерческое обозначение только когда речь идет об отличиях между версиями);

- своеобразная переходная система Windows 95/98/ME (как и в предыдущем случае, это тоже версии одной и той же системы);
- разработанная с нуля ОС для карманных компьютеров Windows CE.

В *приложении 1* будет приведен краткий обзор историй этих семейств и краткая характеристика наиболее известных или этапных систем, принадлежащих данным семействам. В *приложении 2* описана архитектура процессора x86, являющегося основой наиболее распространенного семейства систем.

Еще одно практически вымершее к настоящему моменту, но оставившее в наследство ряд важных и интересных концепций семейство, — это операционные системы для мини- и микрокомпьютеров фирмы DEC: RT-11, RSX-11 и VAX/VMS.

Замечание

Ряд коммерчески успешных ОС, такие как OS/400 или PalmOS, а также ряд маргинальных и экспериментальных систем, этой классификацией не охватываются.

Выбор операционной системы

Выбор типа операционной системы часто представляет собой нетривиальную задачу. Некоторые приложения накладывают жесткие требования, которым удовлетворяет только небольшое количество систем. Например, задачи управления промышленным или исследовательским оборудованием в режиме жесткого реального времени вынуждают нас делать выбор между специализированными ОС реального времени и некоторыми ОС общего назначения, такими как Unix System V Release 4 (хотя Unix SVR4 теоретически способна обеспечивать гарантированное время реакции, системы этого семейства имеют ряд недостатков с точки зрения задач РВ, поэтому чаще всего предпочтительными оказываются специализированные ОС: QNX, VxWorks, OS-9 и т. д.). Другие приложения, например серверы баз данных, просто требуют высокой надежности и производительности, что отсекает системы класса DOS и MS Windows.

Наконец, некоторые задачи, такие как автоматизация конторской работы в небольших организациях, не предъявляют высоких требований к надежности,

производительности и времени реакции системы, что предоставляет широкий выбор между различными ДОС, MS Windows, Mac OS и многими системами общего назначения. При этом технические параметры системы перестают быть значимыми, и в игру вступают другие факторы. На заре развития персональной техники таким фактором была стоимость аппаратного обеспечения, вынуждавшая делать выбор в пользу ДОС и, позднее, MS Windows.

Нужно отметить, впрочем, что современные версии Windows, несмотря на низкую надежность, сложность конфигурации и поддержки, а также ряд функциональных недостатков, вполне адекватны большинству задач конторской автоматизации. Проблемы возникают, когда задачи, стоящие перед организацией, выходят за пределы распечатки прайс-листов из MS Excel и набора писем в MS Word. Лучше всего проблемы этого рода выражены в следующей притче.

Проблема

Организация имеет двенадцать велосипедов. Стоит задача: перевезти рояль. Что делать? Грузовик не предлагать...

Основная проблема MS Windows состоит вовсе не в том, что это не "настоящая" операционная система — "велосипед", в терминах процитированной притчи, а в том, что она не обеспечивает путей плавного и безболезненного перехода к другим платформам, даже если возникнет необходимость такого перехода. Строго говоря, тот же недостаток свойственен многим другим за крытым (closed) платформам, поставляемым одной фирмой и использующим нестандартные "фирменные" интерфейсы. Пока "закрытое" решение соответствует вашим требованиям, все хорошо, но когда вы выходите за пределы технологических возможностей данного решения, вы оказываетесь в тупике.

Часто основным критерием выбора ОС считают поддержку конкретного приложения или некоторого набора прикладных программ. Действительно, если необходимая вам программа работает только под одной ОС, то ваш выбор оказывается жестко обусловлен этим обстоятельством. Работоспособность приложения определяется не только, а иногда и не столько архитектурой системы, сколько интерфейсом между ОС и прикладной программой — форматом загрузочного модуля, интерфейсом системных вызовов, наличием и интерфейсом дополнительных сервисов. При этом одни и те же или совместимые сервисы могут предоставляться системами разных типов; так, интерфейс Win32 предоставляют как Windows NT, так и системы линии Windows 95.

На практике подобная жесткая обусловленность встречается реже, чем принято думать: существует множество решений, позволяющих с той или иной степенью успеха запустить приложение, предназначенное для одной систем-

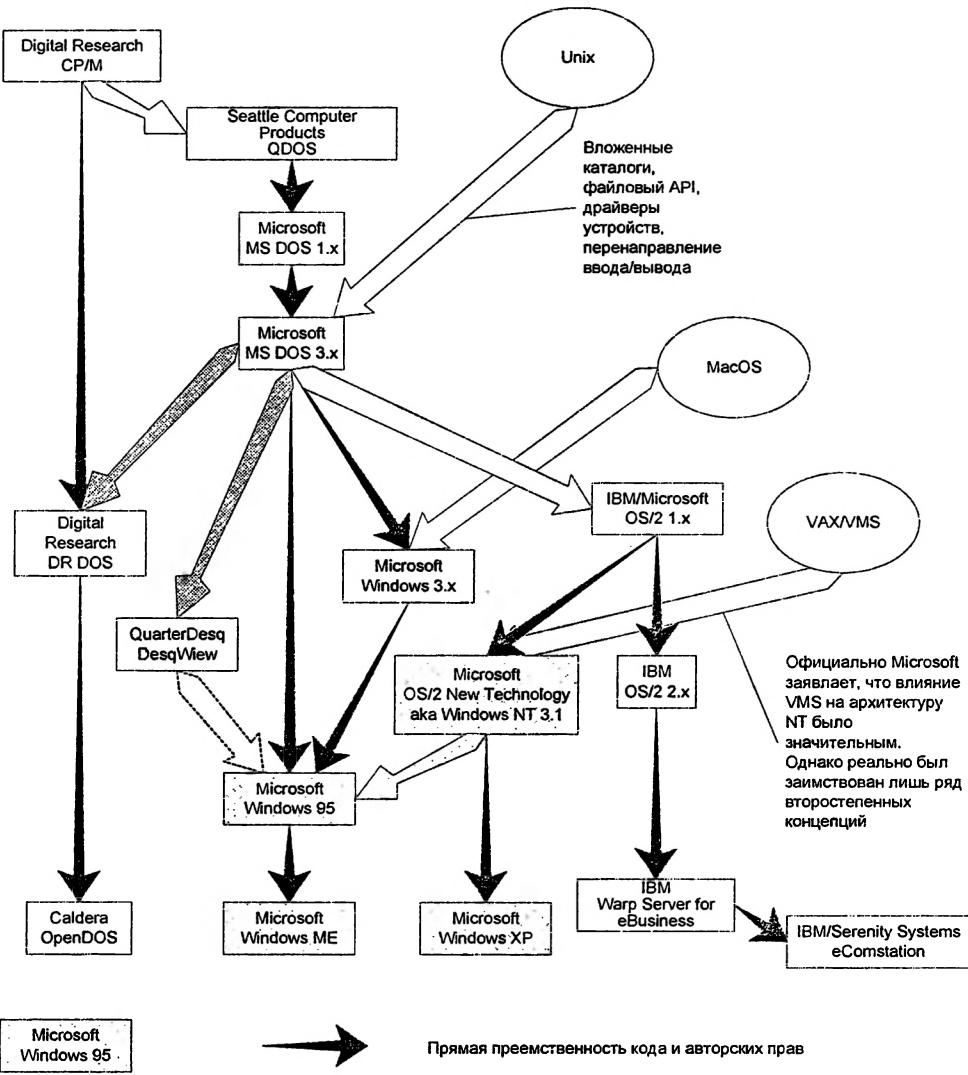


Рис. В.3. Генеалогия ОС семейства CP/M

мы, под совсем другой ОС — это не только системы виртуальных машин, но и различного рода эмулирующие слои, например:

- эмуляторы System V (iBSC и SPARC Solaris) для Linux;
- эмулятор Linux для Solaris 10;
- подсистема WoW (Windows on Windows), позволяющая запускать приложения Win16 на Windows NT;
- проекты (до настоящего времени не завершенные) Wine и Odin, позволяющие запускать приложения Win32, соответственно, на Linux и OS/2;
- библиотеки Bristol Technologies, позволяющие перекомпилировать приложения Win32 для работы под Unix-системами и IBM System 390;
- библиотеки Cygwin и GNU/Win32, реализующие подмножество системных интерфейсов Unix для Win32;
- библиотека EMX, обеспечивающая возможность переноса приложений Unix под OS/2.

Понятно, что такая эмуляция редко бывает вполне совершенной. Впрочем, скептики, критически оценивающие перспективы эмуляционных проектов, часто упускают из вида один своеобразный фактор: действительно, полная bug-for-bug независимая эмуляция большинства современных ОС невозможна или, во всяком случае, чрезвычайно дорога. Однако производителям эмулятора не всегда нужно достигать идеальной совместимости. Им всего лишь необходимо набрать такую критическую массу пользователей, чтобы разработчики хотя бы некоторых приложений были вынуждены сделать этот эмулятор поддерживаемой платформой — тогда разработчики этих приложений сами найдут средства обхода отличий между базовой и эмулирующей средами исполнения.

На рынке приложений Win32 это несколько затрудняется тем, что основное Win32-приложение (Microsoft Office) поставляется той же компанией, что и сами Win32-системы. Разумеется, эта компания активно не заинтересована в поддержке альтернативных Win32-платформ. К счастью, в последние годы появились работоспособные функциональные аналоги MS Office как для Win32, так и для других систем, например OpenOffice.

В большинстве случаев, важным фактором при выборе ОС является доступность специалистов, занимающихся ее поддержкой. Разумеется, здесь существует довольно сильный сетевой эффект: чем более распространена система, тем более распространены и люди, которые в ней разбираются. С другой стороны, здесь есть и своеобразная петля отрицательной обратной связи: большинство специалистов заинтересовано в повышении своих доходов, а для этого полезно быть специалистом в чем-то таком, в чем специалистов мало, например в z/OS, QNX или Cisco IOS. В результате может получиться, что

специалисты по распространенной ОС — это люди малоквалифицированные и неспособные к самообучению, а все хорошие эксперты давно уже специализируются в чем-то другом. Такая ситуация возникает, когда расцвет популярной системы уже позади, и, разумеется, это вносит свой вклад в ее вытеснении с рынка.

Открытые системы

Альтернативой закрытым решениям является концепция открытых систем. Идея открытых систем исходит из того, что для разных задач необходимы разные системы — как специализированные, так и системы общего назначения, просто по-разному настроенные и сбалансированные. Сложность состоит в том, чтобы обеспечить:

- взаимодействие разнородных систем в гетерогенной сети;
- обмен данными между различными приложениями на разных платформах;
- переносимость прикладного ПО с одной платформы на другую, хотя бы путем перекомпиляции исходных текстов;
- по возможности однородный пользовательский интерфейс.

Эти задачи предполагается решать с помощью открытых стандартов — стандартных сетевых протоколов, стандартных форматов данных, стандартизации программных интерфейсов — API (Application Program Interface, интерфейс прикладных программ) и ABI (Application Binary Interface, бинарный интерфейс прикладных программ) — и, наконец, стандартизации пользовательского интерфейса.

В качестве стандартного сетевого протокола предлагалась семиуровневая модель OSI, но прежде чем на основе этой модели было разработано что-то полезное, получило широкое распространение семейство протоколов TCP/IP. Документация по протоколам этого семейства имеет статус общественной собственности (*public domain*); кроме того, есть по крайней мере одна программная реализация данного протокола, также имеющая статус *public domain* — сетевое ПО системы BSD Unix, что стало вполне приемлемым основанием для применения TCP/IP в открытых системах.

Обсуждение стандартных форматов данных увело бы нас далеко от основной темы, но нужно отметить следующее: в настоящее время существует много общепризнанных стандартов представления изображений (особенно растровых) и звуковых данных, но некоторые типы данных так и не имеют признанной стандартной формы. Например, есть несколько открытых форматов представления форматированного текста: troff, L_AT_EX и другие пакеты макросов для системы T_EX, и, наконец, стандарт SGML (Standard Generalized

Markup Language), но ни один из этих стандартов не пользуется популярностью среди разработчиков коммерческих текстовых процессоров (справедливо ради нужно отметить, что форматы troff и L_AT_EX очень неудобны для текстовых процессоров WYSIWYG, но SGML разрабатывался специально для них). Причины такого отношения понятны: предоставление пользователю возможности без проблем обмениваться данными с текстовым процессором конкурента означает дать пользователю возможность выбирать между твоим процессором и его. Впрочем, пользователю от понимания не легче.

Для того чтобы как-то обеспечить переносимость программ между системами различных типов, принимались различные стандарты интерфейса между пользовательской (обычно говорят — прикладной, но это не всегда правильно) программой и ОС. Одним из первых таких стандартов был стандарт библиотек ANSI C. Он основан на системных вызовах ОС Unix, но функции MS-DOS для работы с файлами (использующие *file handle*) тоже достаточно близки к этому стандарту.

Позднее делалось еще несколько попыток стандартизировать интерфейс системных вызовов. Одной из относительно удачных попыток такого рода был POSIX (Portable Operating System Interface [based on] unX, переносимый интерфейс операционной системы, основанный на Unix), который в той или иной форме поддерживается всеми системами семейства Unix и некоторыми ОС, не входящими в это семейство, например Windows NT. Но наибольший успех имела деятельность консорциума X/Open, который в 1998 году сертифицировал операционную систему OS/390 фирмы IBM как соответствующую спецификациям Unix/95 [[www.opengroup.org xu007](http://www.opengroup.org/xu007)] и, таким образом, дал представителю самого древнего из современных родов операционных систем право называться UNIX™.

Впрочем, данная книга не претендует на исчерпывающее руководство по выбору операционной среды, а тем более на справочник по программированию или поддержке конкретной операционной системы. Наша задача — дать обзор основных задач, которые встают перед разработчиками операционной системы, и путей их решения, которые были найдены в ходе развития вычислительных технологий.

Как организована эта книга

Поиск информации о деталях внутреннего устройства конкретных ОС не представляет большой сложности. Книги по архитектуре популярных систем (Win32, Linux, Unix SVR4) издаются большими тиражами и легко доступны. Ряд систем (Linux, FreeBSD, OpenVMS) попросту поставляются с исходными текстами; значительная часть исходных текстов Sun Solaris доступна на сайте

<http://opensolaris.org>. Поставщики коммерческих ОС, как правило, также предоставляют разработчикам весьма полную документацию. Бумажная документация стоит дорого, но практически все приличные поставщики программного обеспечения публикуют документацию в Internet. Так, внутренняя структура ОС Sun Solaris весьма подробно описана в документах, которые могут быть найдены по адресу <http://docs.sun.com>; IBM предоставляет очень много технической информации об ОС, поставляемых этой компанией (MVS/zOS, OS/400, AIX, OS/2), на сайте www.redbooks.ibm.com. Документация по OpenVMS доступна на сайте <http://h71000.www7.hp.com/DOC/>. Довольно много отсканированных документов по компьютерам компаний DEC и программному обеспечению для них, в том числе по упоминающимся в этой книге ОС RT-11 и RSX-11, можно найти на сайте <http://bitsavers.org>.

Даже Microsoft, известная низким качеством и неполнотой документации, предлагает немало сведений — хотя и плохо структурированных — в составе Microsoft Developer Network.

Я не ставил перед собой цели детально описать архитектуру всех упоминаемых систем. Вместо этого, обсуждая те или иные общие концепции, обратился к воплощениям данных концепций в конкретных системах как к примерам. Выбор примеров зачастую определяется весьма субъективными соображениями. Некоторые из них приведены всего лишь потому, что по какой-либо причине поразили мое воображение.

В книге принят используемый в зарубежной литературе стиль ссылок. З квадратных скобках указываются фамилии авторов и год издания книги или публикации статьи. Для ресурсов Internet указывается имя сервера и, для нескольких ссылок на один сервер, дополнительные ключевые слова. Я старался выбирать эти дополнения так, чтобы их можно было использовать для поиска на сервере, если он предоставляет такие средства. Полные библиографические данные или полный URL приводятся в списке источников информации.



ГЛАВА 1

Представление данных в вычислительных системах

А для низкой жизни были числа,
Как домашний, подъяремный скот,
Потому, что все оттенки смысла
Умное число передает.

Н. Гумилев

Из курсов компьютерного ликбеза известно, что современные компьютеры оперируют числовыми данными в двоичной системе счисления, а нечисловые данные (текст, звук, изображение) так или иначе переводят в цифровую форму (оцифровывают).

В силу аппаратных ограничений процессор оперирует числами фиксированной разрядности. Количество двоичных разрядов (битов) основного арифметико-логического устройства (АЛУ) называют *разрядностью процессора* (впрочем, позже мы увидим примеры, когда под разрядностью процессора подразумевается и нечто другое). Процессоры современных систем коллективного пользования (z90, UltraSPARC, Alpha) имеют 64-разрядные АЛУ, хотя в эксплуатации остается еще довольно много 32-разрядных систем, таких как System/390. Персональные компьютеры (x86, PowerPC) и серверы рабочих групп имеют 32-разрядные процессоры. Процессоры меньшей разрядности — 16-, 8- и даже 4-разрядные — широко используются во встраиваемых приложениях.

В прошлом встречались и процессоры, разрядность которых не являлась степенью числа 2, например 36- или 48-разрядные, но из-за сложностей переноса программного обеспечения и обмена данными между системами с некратной разрядностью такие машины постепенно вымерли.

1.1. Введение в двоичную арифметику

Арифметические операции над двоичными числами осуществляются с помощью алгоритма, который в школе изучают под названием "сложение в столбик".

Таблица 1.1. Таблица сложения одноразрядных двоичных чисел

$0 + 0 = 00$
$0 + 1 = 01$
$1 + 0 = 01$
$1 + 1 = 10$

Из табл. 1.1 видно, что результат сложения двух одноразрядных чисел является двухразрядным (двухзначным) числом, а результат сложения двух N-разрядных — ($N+1$)-разрядным. Образующийся дополнительный бит называется *битом переноса* (carry bit).

При сложении двоичных чисел в столбик мы выписываем их друг под другом (пример 1.1). Два младших разряда мы складываем в соответствии с табл. 1.1. При сложении последующих двух разрядов мы должны учитывать не только эти разряды, но и бит переноса из младшего разряда, т. е. производить сложение в соответствии с табл. 1.2. Из этой таблицы видно, что для трех одноразрядных слагаемых все равно получается только два бита суммы, так что для работы со всеми последующими разрядами мы можем обойтись только одним битом переноса (рис. 1.1).

Таблица 1.2. Таблица сложения с учетом переноса

$0 + 0 + 0 = 00$
$0 + 1 + 0 = 01$
$1 + 0 + 0 = 01$
$0 + 0 + 1 = 01$
$1 + 1 + 0 = 10$
$1 + 0 + 1 = 10$
$0 + 1 + 1 = 10$
$1 + 1 + 1 = 11$

Пример 1.1 Сложение двух 8-разрядных чисел ($83 + 56 = 139$)

+	1	1							
	0	1	0	1	0	0	1	1	перенос
	0	0	1	1	1	0	0	0	1-е слагаемое
	1	0	0	0	1	0	1	1	2-е слагаемое
	1	0	0	0	1	0	1	1	результат

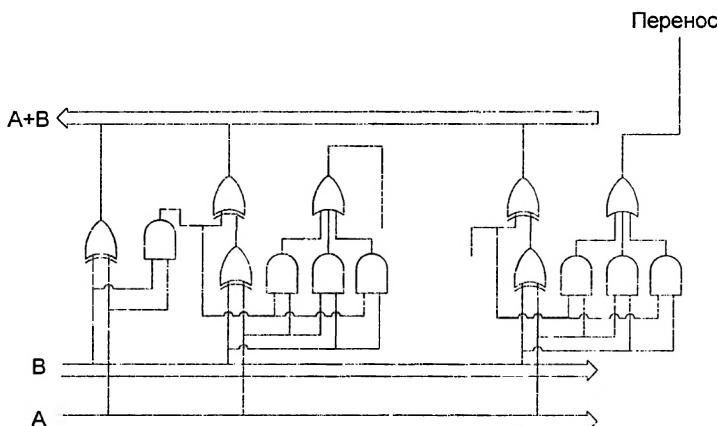


Рис. 1.1. 8-разрядный двоичный сумматор

Дополнительный, 9-й разряд, образующийся при сложении 8-разрядных чисел, переносится в специальный бит слова состояния процессора, который также называется битом переноса и обычно обозначается буквой С (от англ. *carry* — перенос). Этот бит можно использовать как условие в командах условного перехода, а также для реализации 16-разрядной операции сложения или одноименной операции большей разрядности на 8-разрядном АЛУ.

При операциях над беззнаковыми (неотрицательными) числами бит переноса можно интерпретировать как признак *переполнения*: т. е. того, что результат нельзя представить числом с разрядностью АЛУ. Игнорирование этого бита может приводить к неприятным последствиям: например, складывали мы $164 + 95$, а получили в результате 3.

Иногда этот эффект, называемый "обращиванием счетчиков", имеет и полезное применение. Например, используя "часовой" кварцевый генератор с частотой 32 768 Гц и 15-разрядный двоичный счетчик, мы можем отмерять секунды по появлению бита переноса в 16-м разряде и избавляемся от необходимости сбрасывать сам счетчик.

Двоичное вычитание может выполняться аналогичным образом, только необходимо использовать не таблицы сложения, а таблицы вычитания для двух и трех слагаемых. Не утомляя себя и читателя выписыванием этих таблиц, ска-

жем сразу, что операция двоичного вычитания эквивалентна операции двоичного сложения уменьшаемого с *двоичным дополнением* вычитаемого. Двоичное дополнение строится таким образом: все биты числа инвертируются (нули заменяются на единицы, и наоборот), а затем к результату добавляется единица. Доказательство этого утверждения мы оставляем любопытному читателю, а сами просто рассмотрим пример 1.2.

Пример 1.2. Вычитание чисел $(83 - 56 = 27)$

Построение двоичного дополнения $\overline{56}$:

0	0	1	1	1	0	0	0	число
1	1	0	0	0	1	1	1	побитовое отрицание
1	1	0	0	1	0	0	0	побитовое отрицание+1

Сложение с двоичным дополнением $(83 + \overline{56}) \bmod 256 = 27$:

$$\begin{array}{r}
 & & 0 \\
 & + & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
 & & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
 \hline
 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1
 \end{array}$$

Из примера видно, что эквивалентность между операциями неполная: сложение с дополнением сопровождается переносом в 9-й разряд, которого нет при прямом вычитании. Этот факт приводит к тому, что мы уже не можем считать перенос в 9-й разряд критерием того, что результат сложения не может быть представлен 8-ю битами. Точный критерий переполнения для целочисленных операций сложения и вычитания теперь звучит так: переполнение произошло, если перенос в 9-й бит (для 8-разрядного АЛУ) не равен переносу в 10-й бит.

Но в остальном двоичное дополнение сильно упрощает жизнь проектировщикам процессоров: вместо двух устройств, сумматора и дифференциатора (по-русски, сложителя и вычитателя), нам достаточно иметь только сумматор. Кроме того, можно представлять отрицательные числа в двоично-дополнительном коде (табл. 1.3). При таком представлении признак переполнения называют также признаком потери знака.

Видно, что четыре бита позволяют нам представить либо ноль и натуральные числа от 1 до 15, либо целые числа от -8 до 7. Во втором случае, старший бит может интерпретироваться как знаковый: если он равен 1, число отрицательное, если 0 — положительное. Для манипулирования числами в обоих представлениях можно использовать одни и те же команды сложения и вычитания, различие возникает только, когда мы начинаем интерпретировать ре-

зультаты сравнения таких чисел или сами эти числа (например, переводить их в десятичный формат).

Для команд умножения и деления трюк с двоичным дополнением не проходит, поэтому процессоры, использующие такое представление данных, вынуждены иметь по две пары команд умножения и деления, знаковые и беззнаковые. Любознательному читателю предлагается самостоятельно разработать алгоритмы умножения и деления двоичных чисел в двоично-дополнительном представлении. За основу для этих алгоритмов опять-таки рекомендуется взять школьные методы умножения и деления многозначных чисел "в столбик".

Таблица 1.3. Двоичное представление знаковых и беззнаковых чисел

Беззнаковое	Знаковое	Двоичное
7	+7	0111
6	+6	0110
5	+5	0101
4	+4	0100
3	+3	0011
2	+2	0010
1	+1	0001
0	0	0000
15	-1	1111
14	-2	1110
13	-3	1101
12	-4	1100
11	-5	1011
10	-6	1010
9	-7	1001
8	-8	1000

Подавляющее большинство современных процессоров использует двоично-дополнительное представление для целых чисел. В те времена, когда компьютеры были большими, встречались системы, применяющие для этой цели дополнительный, знаковый бит: число -1 представлялось так же, как +1, но с установленным знаковым битом. Такие процессоры должны были иметь от-

дельные команды для беззнаковых и знаковых арифметических операций и более сложное АЛУ. Кроме того, при подобном представлении возникает специфическая проблема "отрицательного нуля".

Иногда наравне с двоичным используется и специфическое, так называемое *двоично-десятичное представление* чисел (рис. 1.2). Это представление особенно удобно для приложений, которые постоянно вынуждены использовать десятичный ввод и вывод (микрокалькуляторы, часы, телефоны с автоопределителем номера и т. д.) и имеют небольшой объем программной памяти, в который нецелесообразно помещать универсальную процедуру преобразования чисел из двоичного представления в десятичное и обратно. В таком представлении десятичная цифра обозначается тетрадой, четырьмя битами. Цифры от 0 до 9 представляются своими двоичными эквивалентами, а комбинации битов 1010, 1011, 1100, 1101, 1110, 1111 недопустимы.

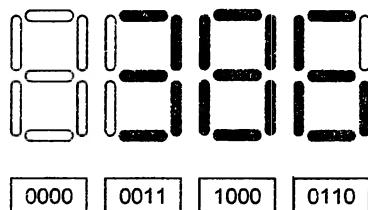


Рис. 1.2. Двоично-десятичное представление чисел

Вместо арифметических операций над такими числами большинство современных микропроцессоров предлагают использовать для их сложения и вычитания обычные бинарные операции, а потом исправлять возникающие при этом недопустимые значения с помощью специальной команды *двоично-десятичной коррекции*. Алгоритм работы данной команды читателю предлагается разработать самостоятельно. Для него потребуется информация о том, происходил ли при сложении двоичный перенос из младшей тетрады. Процессоры, предоставляющие такую команду, имеют и бит межтетрадного переноса.

Если операция производится над числами, имеющими 16 или более двоичных разрядов (4 и более двоично-десятичных), для коррекции нам недостаточно одного межтетрадного переноса — надо иметь по биту переноса на каждую из пар последовательных тетрад. Так далеко ни один из существующих процессоров не заходит. Например, 32-разрядные процессоры x86 имеют команды двоично-десятичной коррекции только для операций над числами с 8-ю двоичными (двумя двоично-десятичными) разрядами.

1.2. Представление рациональных чисел

<desruptor> Куда я попал учиться?
 <desruptor> математик спросил у одногруппника:
 "Егор ты когда-нибудь считал тангенс 15 градусов?"
 <desruptor> Егор: " Нет. Пальцев не хватает"

По материалам bash.org.ru

Представить произвольное вещественное число с помощью конечного числа элементов, способных принимать лишь ограниченный набор значений (а именно таковы все цифровые представления данных), разумеется, невозможно. Максимум, что можно сделать, — это найти то или иное рациональное приближение для такого числа и оперировать им.

Примечание

На самом деле, возможно точное, а не приближенное представление вещественных чисел рациональными — не одиночной дробью, а сходящейся бесконечной последовательностью дробей, так называемое Гильбертова сечение. Конечным представлением служит не сама последовательность, а правило ее формирования. Из-за своей сложности такое представление крайне редко используется в вычислительных системах и никогда не реализуется аппаратно.

Два основных представления рациональных чисел, используемых в компьютерах, — это представления с фиксированной и плавающей точкой. Интерпретирующие системы (например, MathCAD) иногда реализуют и собственно рациональные числа, представляемые в виде целых числителя и знаменателя, но процессоры, умеющие работать с такими числами на уровне системы команд, мне неизвестны.

Представление с *фиксированной точкой* (*fixed point*) (рис. 1.3) концептуально самое простое: мы берем обычное двоичное число и объявляем, что определенное количество его младших разрядов представляет собой дробную часть в позиционной записи. Сложение и вычитание таких чисел может выполняться с помощью обычных целочисленных команд, а вот после умножения и перед делением нам надо, так или иначе, передвинуть двоичную запятую на место.

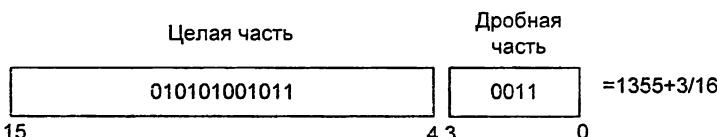


Рис. 1.3. Число с двоичной фиксированной точкой

Примечание

В русском языке принято называть разделитель целой и дробной частей позиционной дроби запятой, а в английском — точкой. Современная вычислительная терминология формировалась на английском языке, и английские словосочетания *fixed-point* и *floating-point* часто переводились на русский язык буквально, поэтому словосочетания "фиксированная точка" и "плавающая точка" прочно вошли в русскую компьютерную лексику, и мы будем использовать именно их.

Современные процессоры обычно не предоставляют арифметических операций с фиксированной точкой, однако никто не запрещает программисту или разработчику компилятора реализовать их на основе стандартных целочисленных операций и команд битового сдвига.

Если нам необходима точность вычислений, определенная количеством десятичных знаков (например, при подсчете рублей с точностью до копеек), нужно помнить, что большинство десятичных дробей в позиционной двоичной записи представляют собой периодические (с бесконечным числом знаков) дроби. Для обеспечения требуемой точности нам следует либо иметь дополнительные двоичные позиции, либо вместо фиксированной двоичной использовать фиксированную десятичную точку: складывать такие числа по-прежнему можно с помощью целочисленных операций, но коррекцию умножения и деления необходимо выполнять с помощью умножения (деления) на степень десяти, а не на степень двойки.

В научных и инженерных вычислениях и цифровой обработке сигналов шире применяются числа с *плавающей двоичной точкой* (*floating point*). Дело в том, что исходные данные для таких вычислений обычно являются результатами измерений физических величин. Все физически реализуемые способы измерений сопровождаются ошибками: для объяснения этого печального факта часто ссылаются на принцип неопределенности Гейзенberга [Карнап 1971], но при практических измерениях гораздо большую роль играют следующие виды ошибок.

- Термальные шумы — температурные колебания измеряемого объекта и измерительного инструмента.
- Инструментальные погрешности, обусловленные различными ошибками при изготовлении или калибровке инструмента или изменением его параметров с момента последней калибровки, например, за счет механического износа.
- Методологические ошибки, обусловленные не только неумением пользоваться инструментом, но и тем фактом, что наши инструменты не имеют непосредственного контакта с явлением, которое мы хотим измерить: например, измеряя динамику валового внутреннего продукта, мы, в действительности, измеряем также и эффекты, обусловленные динамикой денеж-

ной массы. Возможна и обратная ситуация, когда наш инструмент имеет слишком тесный контакт с измеряемым объектом и искажает его поведение. Из школьного курса физики известны такие источники методологических ошибок, как падение напряжения на вольтметре или внутреннее сопротивление амперметра.

Если с принципом Гейзенберга бороться невозможно, то тепловые и инструментальные ошибки можно уменьшать, первые — снижением температуры, а вторые — большей тщательностью изготовления и калибровки, ужесточением условий эксплуатации и хранения или совершенствованием самого инструмента (например, интерферометрический лазерный дальномер при равной тщательности изготовления будет намного точнее рулетки). Однако ни абсолютный нуль температуры, ни абсолютная точность изготовления физически недостижимы, и даже приближение к ним в большинстве ситуаций недопустимо дорого.

Сомневающемуся в этом утверждении читателю предлагается представить, во что бы превратилась его жизнь, если бы при взвешивании продуктов в магазине сами продукты и весы охлаждались хотя бы до температуры жидкого азота, а плотницкий метр надо было бы хранить с теми же предосторожностями, что и метеорологический эталон.

Поэтому на практике целесообразно смириться с погрешностями измерений, а на приборе указать точность, которую его изготовитель реально может гарантировать при соблюдении пользователем условий хранения и эксплуатации. Полному искоренению подлежат только методологические ошибки, да и с ними во многих ситуациях приходится смириться из-за неприемлемо высокой стоимости прямых измерений.

Относительно дешевый способ повышения точности — многократные измерения и усреднение результата, но этот метод повышает стоимость измерений и, если измерения производятся одним и тем же инструментом, не позволяет устранить инструментальные и методологические ошибки.

И тепловые шумы, и инструменты порождают ошибки, которые при прочих равных условиях пропорциональны измеряемой величине. Десятиметровая рулетка имеет инструментальную ошибку, измеряемую миллиметрами, а десятисантиметровый штангенциркуль — микронами. Поэтому ошибку измерений часто указывают не в абсолютных единицах, а в процентах.

При записи результатов измерений хорошим тоном считается указывать точность этих измерений и не выписывать десятичные цифры (или знаки после запятой), значения которых лежат внутри границ ошибки. Цифры перед десятичной запятой в этом случае заменяются нулями, а после нее — просто отбрасываются. Для сокращения записи больших значений, измеренных с относительно небольшой точностью, в научной и инженерной литературе исполь-

зуется экспоненциальная запись чисел: незначащие (в данном случае — лежащие внутри границ ошибки) младшие десятичные цифры отбрасываются, и после числа добавляется 10^N , где N — количество отброшенных цифр. Обычно рекомендуют нормализовать такую запись, перемещая десятичную запятую на место после старшей цифры (смещение запятой на одну позицию влево соответствует увеличению показателя степени множителя на единицу). В соответствии с этими правилами величина $2\ 128\ 506 \pm 20$ преобразуется к виду $2,12850 \times 10^6$ (обратите внимание, что младший ноль в данном случае является значащим).

При бытовых измерениях обычно обходятся двумя-тремя значащими десятичными цифрами. В научных и инженерных измерениях используются и большие точности, но на практике измерения с точностью выше шестого десятичного знака встречаются разве что при разработке и калибровке метрологических эталонов. Для сравнения, цифровая телефония обходится 8 битами, а аналого-цифровой преобразователь бытовой звуковой карты имеет 12, реже 14 значащих двоичных разрядов. Большинство современных карт считаются 16-битными, но на практике младшие разряды их АЦП оцифровывают только тепловой шум усилителя и собственных старших разрядов, а также ошибки калибровки того и другого. Честные (т. е. такие, у которых все разряды значащие) 16-разрядные АЦП используются в профессиональной звукозаписывающей и измерительной аппаратуре, 24-разрядные АЦП относятся к прецизионной аппаратуре, а 32-разрядные на практике не применяют.

Вычислительные системы широко используют представления чисел с плавающей точкой, только не десятичной, а двоичной (рис. 1.4). Идея этого представления состоит в том, чтобы нормализовать позиционную двоичную дробь, избавившись от незначащих старших нулевых битов и освободив место для (возможно) значащих младших разрядов. Сдвиг, который нужен для нормализации, записывается в битовое поле, называемое *порядком (exponent)*. Само же число называется *мантиссой (mantissa, significand)*.

Порядок	Мантисса	
01010100	100000001100011100111110	=8439614*2**84
31	24 23	0

Рис. 1.4. Число с плавающей двоичной точкой

Число с плавающей точкой, таким образом, состоит из двух битовых полей: мантиссы M и порядка E . Число, представленное двумя такими полями, равно $M \times 2^E$. Нормализация состоит в отбрасывании всех старших нулей, поэтому старший бит нормализованной двоичной мантиссы всегда равен 1. Большин-

ство современных реализаций чисел с плавающей точкой используют этот факт для того, чтобы объявить незначащими не только старшие нули, но и эту единицу, и, таким образом, выигрывают дополнительный бит точности мантиссы.

Сложение двух чисел с плавающей точкой состоит в *депонормализации* мантисс (совмещении двоичных точек), их сложении и нормализации результата. Перемножение таких чисел, соответственно, выполняется перемножением мантисс, сложением порядков и опять-таки нормализацией результата.

В некоторых старых архитектурах, например БЭСМ-6, все арифметические операции выполнялись над числами с плавающей точкой, однако существовала возможность выключить нормализацию мантиссы. Ненормализованные числа с плавающей точкой использовались для представления значений с фиксированной точкой, в том числе и целочисленных.

До начала 1980-х разные архитектуры использовали различные форматы чисел с плавающей точкой и различные подходы к нормализации и округлению значений. Это порождало проблемы при обмене данными между вычислительными системами и при переносе ПО, особенно вычислительных программ, чувствительных к ошибкам округления. В 1981 году было принято соглашение IEEE 754, стандартизующее представления чисел и точную семантику операций над ними, в том числе все детали округления результата. Большинство современных процессоров реализуют операции с плавающей точкой в соответствии со стандартом IEEE 754.

Стандарт языка ANSI C требует наличия 32-битового (8-разрядный порядок и 24-разрядная мантисса) и 64-битового (16-разрядный порядок и 48-разрядная мантисса) представлений чисел с плавающей точкой, которые называются, соответственно, *числами одинарной и двойной точности* (*float* и *double float* или просто *double*). Числа двойной точности, конечно же, не могут быть результатом прямых измерений физических величин, но позволяют избежать накопления ошибок округления при вычислениях.

Большинство современных процессоров общего назначения и ориентированных на приложения цифровой обработки сигналов (ЦОС) предоставляет операции над такими числами, а зачастую и над числами большей разрядности. Операции практически всегда включают сложение, вычитание, умножение и деление. Часто на уровне системы команд реализуются и элементарные функции: экспонента, логарифм, квадратный корень, синус, косинус и т. д. Процессоры ЦОС нередко предоставляют и отдельные шаги дискретного преобразования Фурье.

Количество операций с плавающей точкой в секунду (Floating Operations Per Second, FLOPS, в наше время чаще говорят о Mflops — миллионах операций

в секунду), которые может исполнять процессор, является одной из важных его характеристик, хотя и не для всех приложений эта характеристика критична.

1.3. Представление текстовых данных

— Вы можете прочитать третью снизу строку таблицы?
— Н, К, И, М... Доктор, у вас криво настроена кодировка!

Все используемые способы представления текстовых данных, так или иначе, сводятся к нумерации символов алфавита (или множества символов системы письменности интересующего нас языка, которое используется вместо алфавита — слоговой азбуки, иероглифов и т. д.) и хранения полученных целых чисел наравне с обычными числами. Способ нумерации называется *кодировкой* (*character set*), а числа — *кодами символов* (*character code*).

Для большинства кодировок языков, использующих алфавитную письменность (латиница, кириллица, арабский алфавит, еврейский и греческий языки), достаточно 127 символов. Самая распространенная система кодирования латиницы — ASCII — использует 7 бит на символ. Другие алфавиты обычно кодируются более сложным образом: символы алфавита получают коды в диапазоне от 128 до 255, а коды от 0 до 127 соответствуют кодам ASCII. Таким образом, любой символ этих алфавитов, в том числе и в многоязычных текстах, использующих сочетание национального алфавита и латиницы, может быть представлен 8 битами или одним байтом. Но для японских слоговых азбук, а тем более для китайской иероглифики, 255 кодов явно недостаточно, и приходится использовать многобайтовые кодировки. Распространенное обозначение таких кодировок — DBCS (Double Byte Character Set, набор символов, кодируемый двумя байтами).

Двух байтов, в принципе, достаточно, чтобы сформировать единую кодировку для всех современных алфавитов и основных подмножеств иероглифики. Попытка стандартизовать такое представление — Unicode — пока что не имеет полного успеха. Отчасти это можно объяснить тем, что потребность в представлении разноязычных текстов в пределах одного документа ограничена, кроме того, слишком много старого программного обеспечения использует предположение о том, что символ занимает не более байта. Такие программы не могут быть легко преобразованы для работы с Unicode.

Используются две основные кодировки латиницы: ASCII и EBCDIC (Extended Binary Coded Decimal Information Code), применяемая системами AS/400, System/370, System/390 и z90 фирмы IBM. Для представления русского варианта кириллицы существует три основных кодировки: альтернативная (из-

вестная также как cp866), cp1251 и KOI-8, а также ряд менее широко используемых (ISO 8892-5 и др.).

Арифметические операции над такими "числами" обычно бессмысленны, зато большой смысл имеют операции сравнения. Операции сравнения в современных процессорах реализованы как *перазрушающее вычитание* — мы производим те же действия, что и при обычном двоичном вычитании, но запоминаем не сам результат, а лишь флаги знака, переноса и равенства результата нулю. На основании значений этих флагов определяем результат сравнения: если разность равна нулю, сравниваемые символы одинаковы, если она положительна или отрицательна, один из символов больше или меньше другого.

Естественно, чаще всего мы хотим интерпретировать результаты посимвольного сравнения как лексикографическое (алфавитное) "больше" или "меньше" (для русского алфавита, "а" меньше, чем "б"). Проще всего это делать, если нумерация символов совпадает с их порядком в алфавите, но далеко не для всех распространенных кодировок это справедливо.

В кодировке ASCII (American Standard Code for Information Interchange, американский стандартный код обмена информацией), например, все символы латиницы, цифры и большинство распространенных знаков препинания обозначаются кодами от 0 до 127, при этом коды букв расставлены в соответствии с латинским алфавитом. В США, как и в других англоязычных странах, латинский алфавит используется в неизмененном виде, а для передачи звуков, отсутствовавших в оригинальном латинском языке, применяется причудливая орфография. Большинство других европейских алфавитов обходит проблему несоответствия фонетик путем расширения набора символов латиницы — например, в немецком языке добавлены буквы ö, ä, ÿ и ß. Другие языки имеют множество различных "акцентов" и "диакритических символов", расставляемых над буквами для указания особенностей произношения. Некоторые языки, например французский, используют одновременно и расширения алфавита, и причудливую орфографию. Нередко встречаются и дополнительные знаки препинания, например, ¿ и ¡ в испанском языке.

Все символы-расширения в каждом из национальных алфавитов находятся на определенных местах, но при использовании кодировки ASCII для представления этих символов сохранить данный порядок невозможно — соответствующие коды уже заняты. Так, в кодировке ISO 8895-1 все символы латиницы кодируются в соответствии с ASCII, а коды расширений более или менее произвольно раскиданы между 128 и 255. Более яркий пример той же проблемы — кодировки кириллицы семейства КОИ (Код Обмена Информацией), в которых символы кириллицы сопоставлены фонетически соответствующим им символам латиницы ("филе нот фоунд" или, наоборот, "esli wy ne movete

программисты говорят о том, что это «русский, смените кодировку»). Естественно, совместить такое со-поставление и алфавитную сортировку невозможно.

Стандартным решением в таких случаях является использование для сравнения и лексикографической сортировки промежуточных таблиц, в которых для каждого допустимого кода указан его номер в лексикографическом порядке. На уровне системы команд процессоры этого обычно не делают, но на уровне библиотек языков высокого уровня это осуществляется очень часто.

При обмене данными между системами, использующими разные кодировки, необходимо учитывать этот факт. Стандартный способ такого учета, применяемый во многих кросс-платформенных форматах документов (HTML, MIME), — это сообщение где-то в теле документа (обычно в его начале) или в передаваемой вместе с документом метаинформации об используемых языке и кодировке. Большинство средств просмотра почты и документов HTML умеют интерпретировать эту метаинформацию, поэтому конечный пользователь все реже и реже сталкивается с необходимостью самостоятельно разбираться в различных кодировках.

Проблемы возникают, когда метаинформация не указана или, что еще хуже, указана неправильно. Последнее особенно часто встречается при неправильной настройке почтового клиента у отправителя. Для борьбы с такими сообщениями разработаны даже специальные программы, применяющие частотный анализ текста для определения реально использованной кодировки.

Два основных подхода к представлению форматированного текста — языки разметки и сложные структуры данных, используемые в текстовых процессорах. Примерами языков разметки являются HTML, L^AT_EX, troff. В этих языках обычный текст снабжается командами, указывающими на то, каким шрифтом следует отображать конкретный фрагмент текста и как его следует форматировать (например, какова ширина параграфа, следует ли делать переносы в словах, надо ли выравнивать текст по левому или правому краю). Команды представляют собой специальные последовательности символов той же кодировки, в которой набран и основной текст.

Отформатированные таким образом тексты могут генерироваться как вручную (например, L^AT_EX ориентирован именно на ручной набор текста), так и различными автоматизированными средствами.

1.4. Представление изображений

Все известные форматы представления изображений (как неподвижных, так и движущихся) можно разделить на *растровые* и *векторные*.

В векторном формате изображение разделяется на примитивы — прямые линии, многоугольники, окружности и сегменты окружностей, параметрические кривые, заливые определенным цветом или шаблоном, связные области, набранные определенным шрифтом отрывки текста и т. д. (рис. 1.5). Для пересекающихся примитивов задается порядок, в котором один из них перекрывает другой. Некоторые форматы, например, PostScript, позволяют задавать собственные примитивы, аналогично тому, как в языках программирования можно описывать подпрограммы. Такие форматы часто имеют переменные и условные операторы и представляют собой полнофункциональный (хотя и специализированный) язык программирования.



Рис. 1.5. Двухмерное векторное изображение



Рис. 1.6. Трехмерное векторное изображение

Каждый примитив описывается своими геометрическими координатами. Точность описания в разных форматах различна, нередко используются числа с плавающей точкой двойной точности или с фиксированной точкой и точностью до 16-го двоичного знака.

Координаты примитивов бывают как двух-, так и трехмерными. Для трехмерных изображений, естественно, набор примитивов расширяется, в него включаются и различные поверхности — сферы, эллипсоиды и их сегменты, параметрические многообразия и др. (рис. 1.6).

Двухмерные векторные форматы очень хороши для представления чертежей, диаграмм, шрифтов (или, если угодно, отдельных букв шрифта) и отформатированных текстов. Такие изображения удобно редактировать — изображения и их отдельные элементы легко поддаются масштабированию и другим преобразованиям. Примеры двухмерных векторных форматов: PostScript (язык управления лазерными печатающими устройствами, разработанный

компанией Adobe), PDF (Portable Document Format, специализированное подмножество PostScript), WMF (Windows MetaFile), PCL (Printer Control Language, система команд принтеров, поддерживаемая большинством современных лазерных и струйных печатающих устройств). Примером векторного представления движущихся изображений является MacroMedia Flash. Трехмерные векторные форматы широко используются в системах автоматизированного проектирования и для генерации фотoreалистичных изображений методами трассировки лучей, а в последние годы — в игровых программах.

Однако преобразование реальной сцены (например, полученной оцифровкой видеоизображения или сканированием фотографии) в векторный формат представляет собой сложную и, в общем случае, неразрешимую задачу. Программы-векторизаторы существуют, но потребляют очень много ресурсов, а качество изображения во многих случаях получается низким. Самое же главное — создание фотoreалистичных (фотографических или имитирующих фотографию) изображений в векторном формате, хотя теоретически и возможно, на практике требует большого числа очень сложных примитивов. Гораздо более практичным для этих целей оказался другой подход к оцифровке изображений, который использует большинство современных устройств визуализации: растровые дисплеи и многие печатающие устройства.

В растровом формате изображение разбивается на прямоугольную матрицу элементов, называемых *пикселями* (слегка искаженное PICture ELeMent — элемент картинки). Матрица называется *растром* (raster, предположительно от лат. raster, rastrum — мотыга). Для каждого пикселя определяется его яркость и, если изображение цветное, цвет. Если, как это часто бывает при оцифровке реальных сцен или преобразовании в растровый формат (растеризации) векторных изображений, в один пикセル попали несколько элементов, их яркость и цвет усредняются с учетом занимаемой площади. При оцифровке усреднение выполняется аналоговыми контурами аналого-цифрового преобразователя, при растеризации — алгоритмами *анти-алиасинга*.

Размер матрицы называется *разрешением растрового изображения*. Для печатающих устройств (и при растеризации изображений, предназначенных для таких устройств) обычно задается неполный размер матрицы, соответствующей всему печатному листу, и количество пикселов, приходящихся на вертикальный или горизонтальный отрезок длиной 1 дюйм; соответствующая единица так и называется — *точки на дюйм* (Dots Per Inch, DPI).

Для черно-белой печати обычно достаточно 300 или 600 DPI. Однако принтеры, в отличие от растровых терминалов, не умеют манипулировать яркостью отдельной точки, поэтому изменения яркости приходится имитировать, разбивая изображение на квадратные участки и регулируя яркость относительным количеством черных и белых (или цветных и белых при цветной печати)

точек в этом участке. Для получения таким способом приемлемого качества фотопротеалистичных изображений 300 DPI заведомо недостаточно, и даже бытовым принтерам приходится использовать гораздо более высокие разрешения, вплоть до 2400 DPI.

Вторым параметром растрового изображения является разрядность одного пикселя, которую называют *цветовой глубиной*. Для черно-белых изображений достаточно одного бита на пикセル, для градаций яркости серого или цветовых составляющих изображения необходимо несколько битов (рис. 1.7). В цветных изображениях пиксель разбивается на три или четыре составляющие, соответствующие разным цветам спектра. В промежуточных данных, используемых при оцифровке и редактировании растровых изображений, цветовая глубина достигает 48 или 64 бит (16 бит на цветовую составляющую). Яркостный диапазон современных мониторов, впрочем, позволяет ограничиться 8 битами, т. е. 256 градациями, на одну цветовую составляющую: большее количество градаций просто незаметно глазу.



Рис. 1.7. Растровое изображение

Наиболее широко используемые *цветовые модели* — это RGB (Red, Green, Blue, красный, зеленый, синий, соответствующие максимумам частотной характеристики светочувствительных пигментов человеческого глаза), CMY (Cyan, Magenta, Yellow, голубой, пурпурный, желтый, дополнительные к RGB) и CMYG — те же цвета, но с добавлением градаций серого. Цветовая модель RGB используется в цветных кинескопах и видеоадаптерах, CMYG — в цветной полиграфии.

В различных графических форматах используются разные способы хранения пикселов. Два основных подхода: хранить числа, соответствующие пикселям,

одно за другим, или разбивать изображение на битовые плоскости — сначала хранятся младшие биты всех пикселов, потом — вторые и т. д. Обычно растровое изображение снабжается заголовком, в котором указано его разрешение, цветовая глубина и, нередко, используемая цветовая модель.

1.5. Представление звуков

Два основных подхода к хранению звуковых данных можно сопоставить с векторным и растровым способами хранения изображений: это MIDI и подобные ему форматы и оцифрованный звук.

В формате MIDI (Musical Instrument Digital Interface — цифровой интерфейс музыкальных инструментов) звук генерируется синтезатором, который умеет порождать звуки различного тембра, высоты, длительности и громкости. Тембры этих звуков обычно более или менее соответствуют звукам распространенных музыкальных инструментов. Вместо собственно звука хранится последовательность команд этого синтезатора. Используя в качестве звуковых примитивов фонемы человеческого языка, этот подход можно применить и для синтеза речи.

MIDI-файлы имеют малый объем и, при наличии аппаратного синтезатора, не требуют ресурсов центрального процессора для воспроизведения, поэтому их часто используют в качестве фонового озвучивания игровых программ и Web-страниц. К недостаткам этого формата следует отнести тот факт, что качество его воспроизведения определяется качеством синтезатора, которое у дешевых звуковых карт оставляет желать лучшего, и то, что далеко не всякий звук можно воспроизвести таким способом.

Задача преобразования реального звука в MIDI сродни задаче векторизации растрового изображения и другим задачам распознавания образов, и в общем виде не разрешима.

Оцифрованный звук, напротив, является результатом простого осуществления аналого-цифрового преобразования реального звука. Характеристиками такого звука являются частота дискретизации, разрешение АЦП и количество каналов — моно или стерео.

1.6. Упаковка данных

Квипто: Пакуйте
Ю. Могильский, к.ф. "Ва-банк"

Данные многих форматов имеют значительный объем, поэтому их хранение и передача зачастую требуют значительных ресурсов. Одним из спо-

собов решения этой проблемы является повышение емкости запоминающих устройств и пропускной способности каналов связи. Однако во многих случаях применима и более дешевая альтернатива этим методам — *упаковка данных (compression)*.

Научной основой всех методов упаковки является теория информации: данные, в которых имеются статистические автокорреляции, называются избыточными или имеющими низкую энтропию. Устранив эти автокорреляции, т. е. повысив энтропию, объем данных можно уменьшить без потери смысла, а зачастую и с возможностью однозначно восстановить исходные данные. Методы повышения энтропии, которые не позволяют по упакованному потоку восстановить исходный, называются *необратимыми, приблизительными* или *сжимающими с потерями (losing compression)*. Соответственно, методы, которые позволяют это сделать, называются *обратимыми, точными*, или *сжимающими без потерь (lossless compression)*.

Один из первых методов упаковки был предложен задолго до разработки современной теории информации; в 1844 году Сэмюэл Морзе построил первую линию проволочного телеграфа. Система кодировки букв, известная как Азбука Морзе (табл. 1.4), использовала для представления различных букв алфавита посылки различной длины, при этом длина посыпалки зависела от частоты использования соответствующего символа в английском языке. Часто встречающиеся символы кодировались более короткими последовательностями.

Таблица 1.4. Русская азбука Морзе¹

Буква	Символы Морзе	Слово	Буква	Символы Морзе	Слово
А	-	а-том	И	..	
Б	-...	бес-са-раб-ка	К	-.-	конс-тан-тин
В	--	ва-ви-лон	Л	.-..	ла-до-жан-ка
Г	--.	го-ло-ва	М	--	ми-нин
Д	-..	до-бав-ка	Н	-.	но-га
Е	.		О	---	о-ло-во
Ж	...-	жат-ва-зла-ков	П	.--.	па-ни-хи-да
З	--..	звон-бу-ла-та	Р	-.	ра-ду-га

¹ Во вспомогательных словах для запоминания слоги, содержащие букву "а", обозначают точкой, не содержащие — тире.

Таблица 1.4 (окончание)

Буква	Символы Морзе	Слово	Буква	Символы Морзе	Слово
С	...	са-ма-ра	Ш	----	ше-ре-ме-тев
Т	-	ток	Щ	--..	щу-ро-гла-зый
У	..-		Ю	..--	
Ф	..-.	фа-на-тич-ка	Я	.-..	
Х	...	ха-на-ан-ка	Ы	-.--	ы-ка-ни-е
Ц	-.-.	цы-га-ноч-ка	Ь-Ь	-...-	
Ч	---.	че-ре-му-ха			

В конце сороковых годов XX века основателем современной теории информации Шенном, и независимо от него, Фано был разработан универсальный алгоритм построения оптимальных кодов. Более известный аналог этого алгоритма был предложен несколько позже Дэвидом Хаффманом [Huffman 1952]. Принцип построения этих кодов в целом соответствует логике, которой руководствовался Морзе, — кодировать значения, которые часто повторяются в потоке, более короткими последовательностями битов.

Коды Хаффмана и Шеннона-Фано устраниют автокорреляции, соответствующие неравномерности встречаемости символов, но сохраняют без изменений часто встречающиеся последовательности символов, а они ответственны за значительную часть избыточности текстов на естественных и синтетических языках. Для упаковки данных такого рода в конце 70-х годов Лемпелем и Зиффом было предложено семейство алгоритмов, наиболее известные из которых — LZ77 и LZW [Lempel-Ziv 1978].

Все эти алгоритмы сводятся к поиску в потоке повторяющихся последовательностей и замене этих последовательностей на их номер в динамически формируемом словаре. Различие состоит в способах кодирования номера и формирования словаря. Номер последовательности в словаре должен содержать больше битов, чем символы исходного потока, хотя бы уже для того, чтобы его можно было отличить от символа, поэтому алгоритмы Лемпеля-Зиффа предполагают дальнейшее перекодирование преобразованного потока кодом Хаффмана. Большинство современных архиваторов, такие как PkZip, GNU Zip, RAR, основаны на вариациях и аналогах алгоритмов Лемпеля-Зиффа.

При упаковке нетекстовых данных могут применяться и другие способы удаления повторений. Например, при упаковке растровых изображений широко

используется метод RLE (Run-Length Encoding — кодирование длины повторяющихся последовательностей), когда повторяющиеся пиксели заменяются счетчиком повторений и значением пикселя.

Наиболее универсальны так называемые арифметические алгоритмы, которые находят и устраниют все автокорреляции, присутствующие во входных данных. Математическое обоснование и принципы работы этих алгоритмов заслуживают отдельной книги и увидели бы нас далеко в сторону от темы. К сожалению, из-за больших вычислительных затрат эти алгоритмы и реализующие их программы не получают широкого распространения.

Все перечисленные алгоритмы способны только устранять автокорреляции, уже существующие во входном потоке. Понятно, что если автокорреляций не было, то упаковки не произойдет, поэтому гарантировать уровень упаковки эти алгоритмы не могут.

Для упаковки данных, полученных оцифровкой реальных сигналов, прежде всего изображений и звука, точные алгоритмы не подходят совершенно. Дело в том, что реальный сигнал всегда сопровождается тепловым, так называемым белым (равномерно содержащим все частоты) шумом. Этот шум искашает наличествующие в сигнале автокорреляции, сам же автокорреляций не имеет, поэтому обратимые алгоритмы с зашумленным сигналом справиться не могут.

Чтобы убедиться в этом, можно попробовать упаковать любым, или даже несколькими из распространенных архиваторов трек аудио-CD или цифровую фотографию... впрочем, чтобы с цифровой фотографией фокус получился, необходимо, чтобы кадр был снят без обработки встроенным упаковщиком камеры.

Идея обширного семейства алгоритмов, пригодных для сжатия зашумленных сигналов, была позаимствована из принципа работы цифровых фильтров "шумодавов". "Шумодав" работает следующим образом: он осуществляет над сигналом преобразование Фурье и удаляет из полученного спектрального образа самые слабые частоты, которые ниже порога подавления. Сигнал при этом, конечно, искашается, но сильнее всего при этом страдает равномерно распределенный по спектру шум, что и требуется (рис. 1.8).

Алгоритмы JFIF (лежащий в основе распространенного формата хранения растровых изображений JPG), MPEG2-4, MPEG3 [www.jpeg.org] тоже начинаются с выполнения над входным потоком преобразования Фурье. Но, в отличие от "шумодава", JFIF удаляет из полученного спектра не частоты, которые ниже заданного порога, а фиксированное количество частот — конечно же, стараясь отобрать самые слабые. Количество частот, которые надо выкинуть, определяется параметром настройки упаковщика. У JFIF этот параметр так и называется — коэффициентом упаковки, у MP3 — битрейтом.

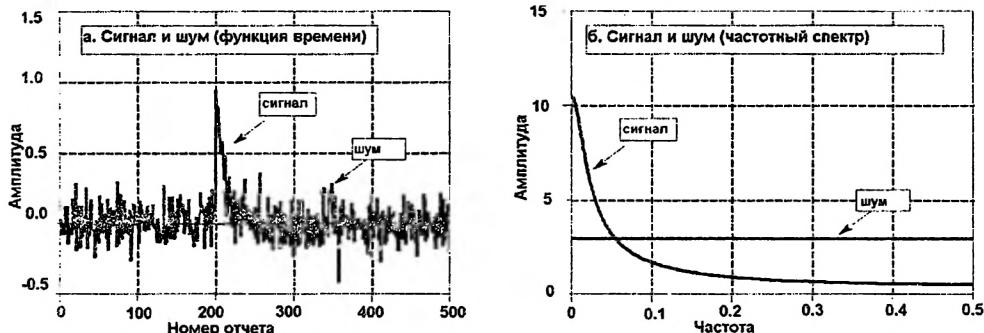


Рис. 1.8. Зашумленный сигнал и его спектральный образ (результат преобразования Фурье), цит. по [Smith 1997]

Профильтрованный сигнал заведомо содержит автокорреляции — даже если исходный незашумленный сигнал их и не содержал, такая фильтрация их создаст — и потому легко поддается упаковке. Благодаря этому, все перечисленные алгоритмы обеспечивают гарантированный уровень упаковки. Они способны сжать в заданное число раз даже чистый белый шум. Понятно, что точно восстановить по подвергнутому такому преобразованию потоку исходный сигнал невозможно, но такой цели и не ставится, поэтому все перечисленные методы относятся к разряду необратимых.

При разумно выбранном уровне упаковки результат — фотoreалистичное изображение или музыкальное произведение — на взгляд (или, соответственно, на слух) практически неотличим от оригинала. Различие может показать только спектральный анализ данных. Но если распаковать сжатое с потерями изображение, подвергнуть его редактированию (например, отмасштабировать или пририсовать какой-нибудь логотип), а потом упаковать снова, результат будет удручающим: редактирование привнесет в изображение новые частоты, поэтому велика опасность, что повторная упаковка даже с более низким коэффициентом сжатия "откусит" какие-то из "полезных" частот изображения. После нескольких таких перепаковок от изображения остается только сюжет, а от музыки — только основной ритм. Поэтому всерьез предлагается использовать приблизительные алгоритмы упаковки в качестве механизма защиты авторских прав: в спорной ситуации только настоящий автор произведения может предъявить его исходный, неупакованный вариант.

Экспериментальные варианты приблизительных алгоритмов вместо классического разложения по взвешенной сумме синусов и косинусов используют разложение по специальным функциям, так называемым *вэйвлетам* (wavelet), или, что то же самое, компактным волнам. Утверждается, что вэйвлетная фильтрация при том же уровне сжатия (понятно, что сравнивать по уровню сжатия алгоритмы, которые сжимают что угодно в заданное число раз, со-

вершенно бессмысленно) дает меньший уровень субъективно обнаружимых искажений. Но субъективное восприятие, как известно, сильно подвержено эффекту плацебо (человек склонен видеть улучшение или вообще изменение там, где его нет, если имеет основания предполагать, что изменение должно произойти), зато вейвлетные алгоритмы сильно уступают обычным вариациям JFIF по производительности, поэтому до сих пор они не вышли из экспериментального состояния.

1.7. Контрольные суммы

Хранение данных и их передача часто сопровождается или может сопровождаться ошибками. Приемнику и передатчику информации необходимо знать, что данные в потоке должны соответствовать определенным правилам. Приводя реальный поток в соответствие с этими правилами, приемник может восстановить его исходное содержание. Количество и типы практически восстанавливаемых ошибок определяются применяемыми правилами кодирования. Понятно, что всегда существует (и во многих случаях может быть теоретически оценен) порог количества ошибок в сообщении, после которого сообщение не поддается даже частичному восстановлению.

Соответствие потока данных тем или иным правилам теория информации описывает как наличие статистических автокорреляций или информационной избыточности в потоке. Такие данные всегда будут иметь больший объем, чем эквивалентные, но не соответствующие никаким правилам (например, упакованные), т. е. помехозащищенность достигается не бесплатно. Существование "бесплатных" средств повышения помехозащищенности каналов противоречит, ни много, ни мало, Второму Началу термодинамики (доказательство этого утверждения требует глубоких знаний в области теории информации и термодинамики, и поэтому здесь не приводится).

Естественные языки обеспечивают очень высокую (в письменной форме двух- и трехкратную, а в звуковой еще большую) избыточность за счет применения сложных фонетических, лексических и синтаксических правил. Остроумным способом дополнительного повышения избыточности человеческой речи являются стихи (белые и тем более рифмованные), широко использовавшиеся до изобретения письменности для повышения надежности хранения в человеческих же головах исторических сведений и священных текстов.

К сожалению, с задачей восстановления искаженных сообщений на естественных языках в общем случае может справиться лишь человеческий мозг. Правила кодирования, применимые в вычислительных системах, должны удовлетворять не только требованиям теоретико-информационной оптимальности, но и быть достаточно просты для программной или аппаратной реализации.

Простейшим способом внесения избыточности является полное дублирование данных. Благодаря своей простоте, этот способ иногда применяется на практике, но обладает многочисленными недостатками. Во-первых, избыточность этого метода чрезмерно высока для многих практических применений. Во-вторых, он позволяет только обнаруживать ошибки, но не исправлять их: при отсутствии других правил кодирования, мы не можем знать, какая из копий верна, а какая ошибочна.

Троекратное копирование обеспечивает еще более высокую избыточность, зато при его использовании для каждого расходящегося бита мы можем проводить голосование: считать правильным то значение, которое присутствует минимум в двух копиях данных (в данном случае мы исходим из того, что вероятность ошибки в одном и том же бите двух копий достаточно мала).

Трехкратное копирование, таким образом, позволяет восстанавливать данные, но имеет слишком уж высокую избыточность. Эти примеры, кроме простоты, любопытны тем, что демонстрируют нам практически важную классификацию избыточных кодов: бывают коды, которые только обнаруживают ошибки, а бывают и такие, которые позволяют их восстанавливать. Далеко не всегда коды второго типа могут быть построены на основе кодов первого типа. Во многих случаях, например при передаче данных по сети, целесообразно запросить повтор испорченного пакета, поэтому коды, способные только обнаруживать ошибки, практически полезны и широко применяются.

Все данные, с которыми могут работать современные вычислительные системы, представляют собой последовательности битов, поэтому все правила, которые мы далее будем рассматривать, распространяются только на такие последовательности.

Простейший из применяемых способов кодирования с обнаружением ошибок — это *бит четности* (*parity bit*). Блок данных снабжается дополнительным битом, значение которого выбирается так, чтобы общее количество битов, равных единице, в блоке было четным или, что в данном случае то же самое, значение которого равно сумме всех остальных бит по модулю 2. Такой код позволяет обнаруживать ошибки в одном бите блока, но не в двух битах (строго говоря — позволяет обнаруживать нечетное количество ошибочных битов). Если вероятность ошибки в двух битах достаточно велика, нам следует либо разбить блок на два блока меньшего размера, каждый со своим битом четности, либо использовать более сложные схемы кодирования.

Самая распространенная из таких более сложных схем — это *CRC* (Cyclic Redundancy Code, циклический избыточный код). При вычислении CRC разрядности N выбирают число R требуемой разрядности и вычисляют остаток от деления на R блока данных (рассматриваемого как единое двоичное чис-

ло), сдвинутого влево на N битов. Двоичное число, образованное блоком данных и остатком, делится на R , и этот факт можно использовать для проверки целостности блока (но не для восстановления данных при ошибке!).

Способность контрольной суммы обнаруживать ошибки логичнее измерять не в количестве ошибочных битов, а в вероятности необнаружения ошибки. При использовании CRC будут проходить незамеченными лишь сочетания ошибок, удовлетворяющие весьма специальному условию, а именно такие, вектор ошибок (двоичное число, единичные биты которого соответствуют ошибочным битам принятого блока, а нулевые — правильно принятым) которых делится на R . При случайном распределении ошибок вероятность этого может быть грубо оценена как $1/R$, поэтому увеличение разрядности контрольной суммы в сочетании с выбором простых R обеспечивает достаточно быстрый и дешевый способ проверки целостности даже довольно длинных блоков. 32-разрядный CRC обеспечивает практически полную гарантию того, что данные не были повреждены, а 8-разрядный — уверенность, достаточную для многих целей. Однако ни четность, ни CRC не могут нам ничем помочь при восстановлении поврежденных данных.

Простой метод кодирования, позволяющий не только обнаруживать, но и исправлять ошибки, называется блочной или параллельной четностью и состоит в том, что мы записываем блок данных в виде двухмерной матрицы и подсчитываем бит четности для каждой строки и каждого столбца. При одиночной ошибке, таким образом, мы легко можем найти бит, который портит нам жизнь. Двойные ошибки такая схема кодирования может обнаруживать, но не способна восстанавливать (рис. 1.9).

Данные																Биты четности	
0	0	0	0	1	0	0	0	0	0	0	1	1	1	1	0	1	0
1	0	0	0	0	1	0	0	1	0	1	0	0	0	0	1	1	0
1	0	0	0	0	1	0	0	1	0	1	0	0	0	0	1	0	0
1	0	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1
1	0	0	0	0	1	0	0	1	0	1	0	0	0	0	1	0	1
1	0	0	0	1	1	0	0	1	1	1	1	0	1	1	1	1	0
<hr/>															<hr/>		
Биты четности																	

Рис. 1.9. Параллельная четность

Широко известный и применяемый код Хэмминга (Hamming code) находится в близком родстве с параллельной четностью. Его теоретическое обоснование несколько менее очевидно, чем у предыдущего алгоритма, но в реализации он, пожалуй, даже проще [Берлекэмп 1971]. Идея алгоритма состоит в том,

чтобы снабдить блок данных несколькими битами четности, подсчитанными по различным совокупностям битов данных. При выполнении неравенства Хэмминга (1.1) сформированный таким образом код обеспечивает обнаружение и исправление одиночных ошибок либо гарантированное обнаружение (но не исправление!) двойных ошибок. Важно подчеркнуть гарантию обнаружения, в отличие от всего лишь высокой вероятности обнаружения при использовании CRC.

$$d + p + 1 \leq 2^p, \quad (1.1)$$

где d — количество битов данных, p — разрядность контрольного кода.

Код, использующий d и p , при которых выражение (1.1) превращается в равенство, называют *оптимальным кодом Хэмминга*.

В некоторых приложениях для подсчета контрольных сумм используются более сложные коды, известные как коды Рида-Соломона. Алгоритмы вычисления этих кодов позволяют генерировать коды с произвольным заданным уровнем избыточности, т. е. обеспечивать восстановление любого наперед заданного количества бит. Разумеется, увеличение надежности всегда приводит к увеличению объема кода.

Часто оказывается целесообразно сочетать упаковку данных с их избыточным кодированием. Наиболее удачным примером такой целесообразности опять-таки являются тексты на естественных языках: статистический анализ такого текста показывает очень высокий, более чем двукратный, уровень избыточности. С одной стороны, это слишком много для большинства практически применяемых способов цифровой передачи и кодирования данных, а с другой — правила формирования этой избыточности слишком сложны и плохо formalизованы для использования в современных программно-аппаратных комплексах. Поэтому для длительного хранения или передачи по низкоскоростным линиям целесообразно упаковывать текстовые данные и снабжать их простыми средствами избыточности, например CRC.

1.8. Введение в криптографию

Список замеченных опечаток:
В шифровке на стр. 325 напечатано: 212850А
Следует читать: 212850Б

При хранении и передаче данных нередко возникает требование защитить их от нежелательного прочтения и/или модификации. Если задача защиты от нежелательной модификации решается обсуждавшимися в предыдущем разделе избыточными кодами, то с прочтением все существенно сложнее.

Проще всего обеспечить защиту данных, лишив потенциальных злоумышленников доступа к физическому носителю данных или физическому каналу, по которому происходит их передача. К сожалению, иногда это невыполнимо (например, если обмен данными происходит по радиоканалу), а зачастую просто слишком дорого. В этом случае на помощь приходят методики, собирательное название которых — *криптография* (круптоурафюс — тайнопись). В отличие от большинства терминов компьютерной лексики это слово не английского, а греческого происхождения. Сам процесс криптографического кодирования называют *шифрованием* (encryption).

История криптографии насчитывает тысячи лет, и многие основополагающие принципы современной криптографии известны, возможно, с доисторических времен, однако существенный прогресс в теории шифрования был достигнут лишь относительно недавно, в связи с разработкой современной теории информации.

Практически все методы криптографии сводятся к преобразованию данных в набор из конечного количества символов и осуществлению над этими символами двух основных операций: подстановки и перестановки. Подстановка состоит в замене одних символов на другие. Перестановка состоит в изменении порядка символов. В качестве символов при этом могут выступать различные элементы сообщения — так, при шифровании сообщений на естественных языках подстановке и перестановке могут подвергаться как отдельные буквы, так и слова или даже целые предложения (как, например, в аллегорических изложениях магических и священных текстов). В современных алгоритмах этим операциям чаще всего подвергаются блоки последовательных битов. Некоторые методики можно описать как осуществление операции подстановки над полным сообщением.

Подстановки и перестановки производятся по определенным правилам. При этом надежда возлагается на то, что эти правила и/или используемые в них параметры известны только автору и получателю шифрованного сообщения и неизвестны посторонним лицам. В докомпьютерную эру старались засекретить обе составляющие процесса шифрования. Сейчас для шифрования, как правило, используют стандартные алгоритмы, секретность же сообщения достигается путем засекречивания используемого алгоритмом параметра, ключа (key).

Прочтение секретного сообщения посторонним лицом, теоретически, может быть осуществлено двумя способами: похищением ключевого значения либо его угадыванием путем анализа перехваченной шифровки. Если первое мероприятие может быть предотвращено только физической и организационной защитой, то возможность второго определяется используемым алгоритмом. Далее мы будем называть процесс анализа шифровки взломом шифра, а че-

ловека, осуществляющего этот процесс, — взломщиком. По-научному эта деятельность называется более нейтрально — *криптоанализ*.

К примеру, сообщение на естественном языке, зашифрованное подстановкой отдельных букв, уязвимо для частотного анализа: основываясь на том факте, что различные буквы встречаются в текстах с разной частотой, взломщик легко — и с весьма высокой достоверностью — может восстановить таблицу подстановки. Существуют и другие примеры неудачных алгоритмов, которые сохраняют в неприкосновенности те или иные присутствовавшие в сообщении автокорреляции — каждый такой параметр можно использовать как основу для восстановления текста сообщения или обнаружения ключа.

Устойчивость шифра к поиску автокорреляций в сообщении называется *криптостойкостью алгоритма*. Даже при использовании удачных в этом смысле алгоритмов, если взломщик знает, что исходные (нешифрованные) данные удовлетворяют тому или иному требованию, например, содержат определенное слово или снабжены избыточным кодом, он может произвести полный перебор пространства ключей: перебирать все значения ключа, допускаемые алгоритмом, пока не будет получено удовлетворяющее требованию сообщение. При использовании ключей достаточно большой разрядности такая атака оказывается чрезмерно дорогой, однако прогресс вычислительной техники постоянно сдвигает границу "достаточности" все дальше и дальше. Так, сеть распределенных вычислений Bovine в 1998 году взломала сообщение, зашифрованное алгоритмом DES с 56-разрядным ключом за 56 часов работы [www.distributed.net]!

Простым и эффективным способом борьбы с такой атакой является расширение пространства ключей. Увеличение ключа на один бит приводит к увеличению пространства вдвое — таким образом, линейный рост размера ключа обеспечивает экспоненциальный рост стоимости перебора. Некоторые алгоритмы шифрования не зависят от разрядности используемого ключа — в этом случае расширение достигается очевидным способом. Если же в алгоритме присутствует зависимость от разрядности, расширить пространство можно, всего лишь применив к сообщению несколько разных преобразований, в том числе и одним алгоритмом, но с разными ключами. Еще один способ существенно усложнить работу взломщику — это упаковка сообщения перед шифрованием и/или дополнение его случайными битами.

Важно подчеркнуть, впрочем, что количество двоичных разрядов ключа является лишь оценкой объема пространства ключей сверху, и во многих ситуациях эта оценка завышена. Некоторые алгоритмы в силу своей природы могут использовать только ключи, удовлетворяющие определенному условию — например, RSA использует простые числа. Это резко сужает объем работы по перебору, поэтому для обеспечения сопоставимой криптостойко-

сти разрядность ключа RSA должна быть намного больше, чем у алгоритмов, допускающих произвольные ключи.

Низкая криптостойкость может быть обусловлена не только алгоритмом шифрования, но и процедурой выбора ключа: если ключ может принимать любые двоичные значения заданной разрядности, но реально для его выбора используется страдающий неоднородностью генератор псевдослучайных чисел, мы можем значительно сократить объем пространства, которое реально должен будет перебрать взломщик наших сообщений (вопросы генерации равномерно распределенных псевдослучайных чисел обсуждаются во втором томе классической книги [Кнут 2000]). Еще хуже ситуация, когда в качестве ключа используются "легко запоминаемые" слова естественного языка: в этом случае реальный объем пространства ключей даже довольно большой разрядности может измеряться всего лишь несколькими тысячами различных значений.

Современные алгоритмы шифрования делятся на два основных класса: с секретным и с публичным ключом (кажущееся противоречие между термином "публичный ключ" и приведенными ранее рассуждениями будет разъяснено далее).

Алгоритмы с секретным ключом, в свою очередь, делятся на потоковые (stream) и блочные (block). Потоковые алгоритмы обычно используют подстановку символов без их перестановки. Повышение криптостойкости при этом достигается за счет того, что правила подстановки зависят не только от самого заменяемого символа, но и от его позиции в потоке.

Примером простейшего — и в то же время абсолютно не поддающегося взлому — потокового алгоритма является *система Вернама* или *одноразовый блокнот* (рис. 1.10). Система Вернама основана на ключе, размер которого равен размеру сообщения или превосходит его. При передаче двоичных данных подстановка осуществляется сложением по модулю 2 (операцией ИСКЛЮЧАЮЩЕГО ИЛИ) соответствующих битов ключа и сообщения.

Если ключ порожден надежным генератором случайных чисел (например, правильно настроенным оцифровщиком теплового шума), никакая информация об автокорреляциях в исходном тексте сообщения взломщику не поможет: перебирая полное пространство ключей, взломщик вынужден будет проверить все сообщения, совпадающие по количеству символов с исходным, в том числе и все сообщения, удовлетворяющие предполагаемому автокорреляционному соотношению.

Это преимущество теряется, если один и тот же ключ будет использован для кодирования нескольких сообщений: взломщик, перехвативший их все, сможет использовать эти сообщения и предположения об их содержимом при попытках отфильтровать ключ от полезной информации — отсюда и второе

название алгоритма. Применение системы Вернама, таким образом, сопряжено с дорогостоящей генерацией и, главное, транспортировкой ключей огромной длины, и поэтому она используется лишь в системах экстренной правительственной и военной связи.

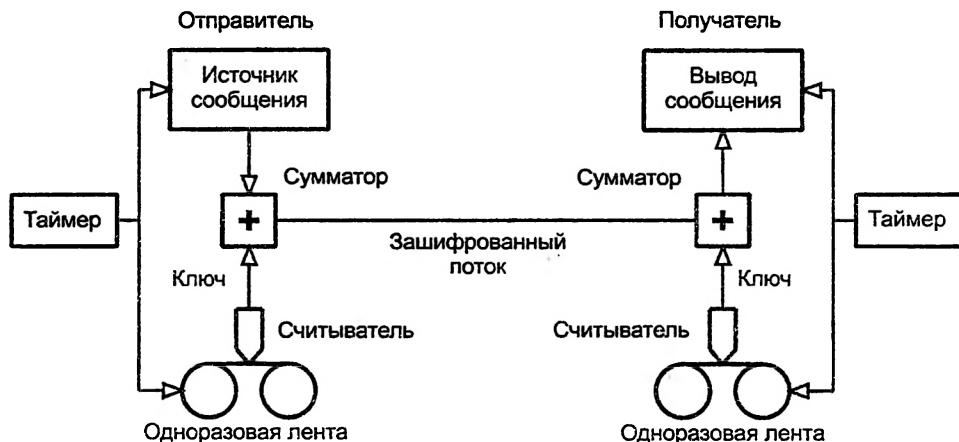


Рис. 1.10. Система Вернама

Более практичным оказалось применение в качестве ключа псевдослучайных последовательностей, порождаемых детерминированными алгоритмами. В промежутке между первой и второй мировыми войнами широкое распространение получили шифровальные машины, основанные на механических генераторах таких последовательностей. Чаще всего использовались сочетания, получаемые при вращении колес с взаимно простыми количествами зубцов.

Основной опасностью при использовании таких методов шифрования является возможность определить текущую точку последовательности — узнав ее (например, по косвенным признакам догадавшись, что в данной точке сообщения должно быть такое-то слово, и восстановив использовавшийся при ее шифровании элемент ключа), взломщик может продолжить генерацию с той же точки и расшифровать весь дальнейший поток.

В системах цифровой связи широкое применение получили блочные алгоритмы, выполняющие над блоками данных фиксированной длины последовательности — иногда довольно сложные — перестановок, подстановок и других операций, чаще всего двоичных сложений данных с ключом по какому-либо модулю. В операциях используются компоненты ключевого слова относительно небольшой разрядности.

Например, широко применяемый блочный алгоритм DES шифрует 64-битные блоки данных 56-битным ключом. Полное описание алгоритма приводится в документе [NBS FIPS PUB 46, 1977]. Русский перевод этого документа может

быть найден в приложениях к книге [Дейтел 1987]. Для современной вычислительной техники полный перебор 56-битного пространства — "семечки", поэтому сейчас все большее распространение получают алгоритмы с большей разрядностью ключа — Blowfish, IDEAL и др. [Анин 2000].

Шифры с открытым ключом называются также двухключевыми. Если в алгоритмах со скрытым ключом для кодирования и декодирования сообщений используется один и тот же ключ, то здесь используются два ключа: публичный и приватный. Для прочтения сообщения, закодированного публичным ключом, необходим приватный, и наоборот.

Помимо обычных соображений криптостойкости, к таким алгоритмам предъявляется дополнительное требование: невозможность восстановления приватного ключа по публичному иначе как полным перебором пространства ключей.

Двухключевые схемы шифрования намного сложнее в разработке, чем схемы с секретным ключом: требуется найти преобразование, не поддающееся обращению с помощью применявшегося в нем публичного ключа, но такое, чтобы с применением приватного ключа его все-таки можно было обратить. Известные криптоустойчивые схемы основаны на произведениях простых чисел большой разрядности, дискретных логарифмах и эллиптических криевых. Наиболее широкое применение получил разработанный в 1977 году алгоритм RSA [www.rsa.com FAQ].

Известные двухключевые алгоритмы требуют соответствия ключей весьма специфическим требованиям, поэтому для достижения криптостойкости, со-поставимой с блочными алгоритмами, необходимо использовать ключи намного большей разрядности. Так, снятые в 2000 году ограничения Министерства торговли США устанавливали ограничения разрядности ключа, который мог использоваться в экспортном за пределы США программном обеспечении: для схем с секретным ключом устанавливался предел, равный 48 битам, а для схем с открытым — 480.

Использование ключей большой разрядности требует значительных вычислительных затрат, поэтому двухключевые схемы чаще всего применяются в сочетании с обычными: обладатель публичного ключа генерирует случайную последовательность битов, кодирует ее и отправляет обладателю приватного ключа. Затем эта последовательность используется в качестве секретного ключа для шифрования данных. При установлении двустороннего соединения стороны могут сначала обменяться своими публичными ключами, а затем использовать их для установления двух разных секретных ключей, используемых для шифрования данных, передаваемых в разных направлениях.

Такая схема делает практической частую смену секретных ключей: так, в протоколе SSH ключ генерируется на каждую сессию [www.cs.hut.fi SSH];

в протоколе виртуальных приватных сетей IPSEC время жизни ключа ограничено восемью часами [[redbooks.ibm.com sg245234.pdf](http://redbooks.ibm.com/sg245234.pdf)].

Еще более широкое применение двухключевые схемы нашли в области аутентификации и электронной подписи. Электронная подпись представляет собой контрольную сумму сообщения, зашифрованную приватным ключом отправителя. Каждый обладатель соответствующего публичного ключа может проверить аутентичность подписи и целостность сообщения. Это может использоваться для проверки аутентичности как сообщения, так и самого отправителя. Использование в качестве контрольной суммы обычного CRC (см. разд. 1.7) невозможно, потому что генерация сообщения с заданным CRC не представляет вычислительной сложности. Для применения в электронной подписи были разработаны специальные алгоритмы вычисления контрольных сумм, затрудняющие подбор сообщения с требуемой суммой [RFC 1320, RFC 1321].



ГЛАВА 2

Машинные языки

А главное, боги совсем не умели разговаривать.
Они просто верещали, как обезьяны.

Дж. Юис, к.ф. "Наверное, боги сошли с ума"

Часто приходится сталкиваться с утверждением, что современные технологии программирования — такие как языки высокого уровня, интерпретаторы, серверы приложений, библиотеки классов, визуальные среды программирования и т. д. — избавляют разработчика от необходимости знать машинный язык и язык ассемблера.

Действительно, стоимость разработки на языке ассемблера значительно выше, чем на языках высокого уровня (**ЯВУ**). Эмпирически известно, что производительность программиста в строках исходного текста (или, точнее, в операторах) приблизительно одинакова для различных языков. Этот факт имеет определенное психологическое объяснение, состоящее в том, что реальным ограничителем при разработке программного обеспечения является ограниченная способность человека одновременно оперировать большим числом объектов, атрибутов и отношений; большинство исследований на эту тему утверждают, что человек не способен одновременно работать более чем с семью объектами.

Один оператор типичного ЯВУ соответствует нескольким командам (а часто нескольким десяткам, а то и сотням команд) машинного языка или приблизительно тому же количеству директив ассемблера, поэтому программист, пишущий на ЯВУ, будет в несколько раз, а то и на два порядка продуктивнее, чем программист сопоставимой квалификации, работающий на ассемблере.

Преимущества, которые дает язык ассемблера, также невелики. В 70-е и 80-е годы программирование на ассемблере обеспечивало выигрыш по скорости исполнения кода в несколько раз по сравнению с программами, написанными на ЯВУ, но даже в этих условиях часто оказывалось экономически нецелесообразно писать программы на ассемблере полностью. Действитель-

но, в соответствии с широко известным эмпирическим правилом, 90% времени исполняется 10% кода. Переписав на ассемблере только эти 10%, можно было получить почти такой же выигрыш по производительности, как и при переписывании всей программы.

Однако современные оптимизирующие компиляторы свели даже это преимущество на нет. В большинстве случаев код, порожденный современным компилятором, не уступает написанному человеком; в ряде случаев оказывается, что многие оптимизации, применимые к современным суперскалярным процессорам, практически невозможно выполнить вручную — так что на многих тестах компилированный код превосходит, порой весьма значительно, результаты ручного творчества.

Языки высокого уровня не позволяют выполнять некоторые операции над служебными регистрами процессоров, поэтому ряд модулей ОС и низкоуровневых библиотек все-таки приходится писать на ассемблере. Однако все эти операции на практике оказываются сосредоточены в очень небольшом количестве реально используемых программ. Ядро типичной современной ОС, такой как Linux, содержит буквально несколько ассемблерных модулей по несколько сотен строк каждый. Все остальные исходные тексты ядра написаны на С или С++. Впрочем, если учесть ассемблерные вставки в код на языке С и макроопределения, в конечном итоге раскрывающиеся в ассемблерную вставку, можно сказать, что объем ассемблерного кода в ядре Linux составляет несколько процентов.

Практически единственная ниша, в которой ассемблер еще прочно удерживает позиции, — это разработка приложений для восьмиразрядных микроконтроллеров с программной памятью, измеряемой несколькими килословами. Впрочем, по мере распространения более мощных 32-разрядных микроконтроллеров, ассемблер теряет позиции и здесь.

В главе 13 мы познакомимся с некоторыми дополнительными соображениями, которые могут сделать разработку на ассемблере и даже непосредственно в машинных кодах все-таки целесообразной, но рыночные ниши, в которых эти соображения действительно актуальны, также очень узки.

Поэтому ассемблер на практике используется все меньше и меньше, и объявления о найме на работу, в которых требовалось бы знание ассемблера, действительно встречаются все реже и реже. Однако само по себе утверждение, что современному программисту ассемблер знать вообще не нужно, следует признать ошибочным. Оно достаточно похоже на правду, чтобы быть убедительным, но и достаточно отличается от правды, чтобы быть опасным заблуждением.

Правильная формулировка этого утверждения должна звучать приблизительно так: современные технологии программирования позволяют разработчику

не думать все время в терминах ассемблера и машинного языка. Благодаря этому оказываются возможны программные проекты, которые при разработке на ассемблере были бы экономически нецелесообразны или даже практически нереализуемы — однако в ряде случаев способность переформулировать задачу или решение в терминах машинного языка все-таки оказывается крайне полезна.

Так, только человек, знающий, что на самом деле массив в языке С — это всего лишь логическое обозначение для определеной области памяти, — может по-настоящему понять, что такое срыв стека. Действительно, если рассматривать массив как абстрактный объект, к которому применима операция индексирования, то самым умным ответом на вопрос, что случится, если индекс выйдет за границы, определяемые размерами массива, — будет "определяется реализацией". Если же знать, что массив — это набор ячеек памяти с последовательными адресами, то легко понять, что при выходе индекса за границу мы попросту обратимся к соседней переменной — либо, в случае срыва стека, к системной структуре данных (а конкретно, к записи активации процедуры), которой не посчастливилось оказаться рядом с массивом. А для того чтобы понять, что может произойти при неконтролируемой модификации записи активации процедуры, крайне желательно знать, что собой представляет эта самая запись активации.

При разработке многопоточной программы жизненно важно понимать, что большинство конструкций ЯВУ, таких как условный оператор или даже оператор увеличения счетчика на единицу, "на самом деле" — т. е. с точки зрения процессора и программы, переключающей потоки (так называемого *планировщика*), — вовсе не являются элементарными неделимыми операциями, и потому во время проверки условия значения переменных, входящих в это условие, могут измениться. Без специальных дополнительных мер простая проверка условия в многопоточной программе может порождать совершенно неожиданные результаты.

Приведем еще один пример из области, которая слабо связана с темой нашей книги. Если человек знает, как реализованы виртуальные методы в C++ (до уровня понимания того, что такое VMT), то ему можно буквально в двух, в крайнем случае, трех предложениях объяснить, почему возникают трудности при множественном наследовании и что такое виртуальное наследование. В то же время для человека, незнакомого с понятием VMT, вся эта проблематика выглядит глубокой и часто совершенно непонятной абстракцией.

Строго говоря, полезно не только (и даже не столько) знание машинного языка как таковое, сколько способность рассматривать разные описания одной и той же проблемы (или ее решения) и быстро переключаться между этими описаниями, например, между описанием в терминах ЯВУ — т. е. в терминах процедур, переменных, структур и, в крайнем случае, указателей, а то и объ-

ектов — и описанием в терминах машинного языка — т. е. в терминах ячеек памяти и адресов. Крайне полезно также понимать, что большинство этих описаний в конечном итоге эквивалентны, просто в определенной ситуации одно описание удобнее другого, а в другой — наоборот.

Далее в этой книге мы столкнемся со многими понятиями и концепциями, которые практически невозможно понять, не будучи знакомым с основными понятиями машинного языка. К сожалению, в данной главе я не смогу полноценно обучить вас ни машинному языку, ни его мнемонической форме, языку ассемблера. Главным препятствием на пути к этому является просто объем материала — действительно, приличный учебник по ассемблеру (например, [Пирогов 2005]) сопоставим по объему со всей этой книгой, поэтому желающим полноценно изучить программирование на ассемблере я рекомендую воспользоваться специально посвященным данной теме учебным пособием. Вообще, я настоятельно рекомендую всем, кто всерьез интересуется программированием, изучить язык ассемблера для какого-либо доступного процессора (подойдет и x86) и хотя бы немного попрактиковаться. Даже если вы в дальнейшем не сможете заработать ни одного рубля написанием программ на этом языке или не захотите этим заниматься, время не будет потрачено зря, а полученные знания помогут вам углубить понимание многих вопросов в самых разных, порой неожиданных, областях.

В приложении 2 приводится описание системы команд процессора x86 и ассемблера с синтаксисом AT&T. Этот синтаксис используется рядом ассемблеров для систем семейства Unix, в том числе таким популярным продуктом, как GNU assembler, который поставляется с BSD Unix и Linux и доступен для большинства других ОС, в том числе DOS, Win32 и OS/2. Большинство примеров ассемблерного кода для x86 и SPARC, приводимых в этой книге, написаны с использованием синтаксиса AT&T.

Впрочем, я постараюсь построить изложение таким образом, чтобы даже читатель, хорошо знакомый с языками ассемблера, мог найти в настоящей главе для себя что-то интересное.

2.1. Системы команд

Центральный процессор современного компьютера — это устройство, исполняющее команды. Полный набор команд конкретного процессора называют *машинным языком* или *системой команд* (иногда систему команд называют также архитектурой, но это слово слишком перегружено различными значениями).

Разные процессоры часто имеют одну и ту же (или слабо варьирующую) систему команд — например, процессоры Intel 80386, 486, Pentium, Pentium II,

AMD K6, Athlon и т. д. — далее в тексте книги мы будем называть все эти процессоры x86.

Процессоры, которые могут исполнять программы на одном и том же машинном языке, называются *бинарно совместимыми*. Отношение бинарной совместимости не всегда симметрично: например, более новый процессор может иметь дополнительные команды — тогда он будет бинарно совместим с более старым процессором того же семейства, но не наоборот. Нередко бывает и так, что более новый процессор имеет совсем другую систему команд, но умеет исполнять программы на машинном языке старого процессора в так называемом режиме совместимости — например, все процессоры семейства x86 могут исполнять программы для Intel 8086 и 80286. Некоторые ОС для x86 даже предоставляют возможность собрать единую программу из модулей, использующих разные системы команд.

Еще более обширны семейства процессоров, совместимые между собой по языку ассемблера. Такая совместимость означает всего лишь, что каждая команда одного процессора имеет полный функциональный аналог в системе команд другого, это дает возможность автоматизировать преобразование программ из одного машинного языка в другой. Так, Intel 8086 совместим по языку ассемблера с более ранними процессорами той же фирмы, 8080 и 8085.

Как уже говорилось, асимметрия отношений совместимости обычно направлена от предыдущего поколения процессоров к следующему: более новое (как правило, более мощное) устройство совместимо со своим более старым аналогом, но не наоборот, поэтому часто говорят о *совместимости снизу вверх (backward compatibility)*. Это отношение позволяет нам не только классифицировать вычислительные системы по поколениям, но и выделять в разных поколениях предка и совместимых с ним потомков, а в пределах одного поколения находить "братьев" и более дальних родственников.

Прослеживание генеалогий систем команд современных процессоров — увлекательное занятие, которому посвящено немало публикаций, например, [jbayko v12.1.2]. Иногда, кроме бинарной и ассемблерной совместимостей, при построении таких генеалогий учитывают и столь размытое понятие, как концептуальное родство — и тогда, например, процессоры Motorola 680x0 оказываются родней DEC PDP-11.

Наборы команд различных процессоров отличаются большим разнообразием, однако есть операции, которые в той или иной форме умеют выполнять все современные процессоры.

Во-первых, это арифметические операции над целыми числами в двоичном представлении. Даже в наше время многие микроконтроллеры предоставляют только операции сложения и вычитания, но процессоры современных компьютеров общего назначения все без исключения умеют также умножать и

делить, причем не только целые, но и вещественные числа (в разд. 1.2 мы видели, чем такие числа отличаются от того, что называется вещественным числом в математическом анализе и производных от него дисциплинах). Некоторые специализированные процессоры предоставляют и более экзотические математические функции, например, отдельные шаги алгоритма дискретного преобразования Фурье.

Но самая главная команда (или, точнее, самое главное семейство команд), которая и делает процессор полностью программируемым, — это команда перехода, точнее, как минимум две команды: безусловного и условного перехода. На практике большинство современных процессоров имеют по несколько команд безусловного перехода с разными механизмами вычисления адреса точки перехода, а также до десятка, а иногда и более, команд условного перехода по различным условиям.

Микропрограммные автоматы

У фон-неймановских процессоров команды исполняются последовательно, в соответствии с порядком размещения в памяти, и только команды условных и безусловных переходов нарушают этот порядок. "Младшие родственники" фон-неймановских процессоров, микропрограммные автоматы, часто имеют более мощное средство управления последовательностью исполнения: каждая команда автомата имеет битовое поле, содержащее номер следующей по порядку исполнения команды и, таким образом, одновременно является и функциональной командой, и командой безусловного перехода. Методы реализации условных переходов в устройствах такого типа отличаются большим разнообразием.

Такая структура команды облегчает размещение программы в памяти (логически последовательные команды могут быть размещены в любых свободных участках), но приводит к значительному увеличению длины команды — и потому применима лишь в устройствах с очень небольшой длиной адреса команды, т. е. с маленькой программной памятью. Микропрограммные автоматы обычно компенсируют это ограничение сложной структурой каждой отдельной команды — длина таких команд достигает нескольких сотен битов и, в действительности, они содержат по отдельной команде для каждой из функциональных подсистем автомата. Но все равно, сложность программ для таких устройств невелика по сравнению с программами для фон-неймановских процессоров общего назначения, и их часто называют не программами, а микрокодом.

Программируемые логические матрицы (ПЛМ), микропрограммные автоматы и фон-неймановские процессоры представляют собой непрерывный спектр устройств возрастающей сложности — причем далеко не всегда можно с уверенностью отнести конкретное устройство к одной из перечисленных категорий. Простая ПЛМ не может сохранять предыдущее состояние и способна только преобразовывать текущие состояния своих входов в состояния своих выходов. Замкнув некоторые из выходов ПЛМ на некоторые из ее входов через простое запоминающее устройство (регистр), мы получаем более сложное устройство, обладающее памятью. Простые микропрограммные автоматы реализуются на основе ПЛМ и нескольких регистров. Более сложные автоматы содержат много

регистров и специализированные функциональные устройства, такие как счетчики и сумматоры.

Типичный современный фон-неймановский процессор общего назначения, такой как Pentium VI, представляет собой сложный микропрограммный автомат, микрокод которого интерпретирует коды команд x86.

С помощью процессора, не имеющего команд перехода, нельзя реализовать алгоритм, который имел бы циклы или условные операторы с неопределенным количеством итераций. Следовательно, такой "процессор" пригоден лишь для осуществления вычислений по фиксированной формуле или генерации фиксированных последовательностей сигналов. В наше время такие функции обычно реализуются с помощью программируемых логических матриц.

История компьютеров с хранимой программой

Первые программируемые компьютеры, такие как IBM ASCC (более известный как Harvard Mark I) и Z3 Конрада Цузе, использовали линейную программу, записанную на перфоленте, и не имели команд перехода. Цикл с известным количеством повторений на таком компьютере реализовали с помощью многократного повторения тела цикла. Циклы с неизвестным количеством повторений соответствуют операторам `while` современных ЯВУ

Более быстродействующий электронный компьютер ENIAC, разработанный в 1942 году Эккертом и Мочли, программировался с помощью коммутационных панелей (plug-board). Коммутационная панель представляет собой плату с разъемами, которые можно попарно соединять проводами. Эта плата вставляется в управляющее устройство компьютера и, фактически, изменяет способ соединения узлов компьютера между собой. Первоначально такие устройства использовались для программирования табуляторов (устройств обработки перфокарт) компании IBM. Коммутационные панели легко заменять (рис. 2.1), но разработка таких "программ" представляла очень сложную и трудоемкую задачу, главным образом потому, что сама по себе панель не имеет средств для определения порядка операций.

ENIAC поставлялся заказчику и вводился в эксплуатацию по частям: в 1944 году на полигон в Эбердине были поставлены аккумуляторы и счетчик циклов (часть того, что позднее получило название *master programmer*), а полностью машина была собрана только осенью 1945 года. Позднее к ENIAC добавлялись другие устройства, в том числе сдвиговый регистр, а в 1953 году — ОЗУ на ферритовых сердечниках объемом 100 слов [Goldstine 1972].

ENIAC в конфигурации 1945 года имел двадцать аккумуляторов, умножитель, делитель, устройство для извлечения квадратного корня и три так называемых "функциональных таблицы" (function table).

Аккумуляторы использовались для хранения данных. В то же время, любые два аккумулятора совместно могли осуществить сложение или вычитание хранящихся в них чисел.

"Функциональная таблица" представляла собой, говоря современным языком, банк программируемого ПЗУ, который мог рассматриваться либо как 200 шестизначных десятичных чисел, либо как 100 двенадцатизначных чисел. Это ПЗУ

предназначалось для хранения значений таблично заданных функций, например, логарифмов. Физически этот банк представлял собой матрицу рубильников, каждый из которых соответствовал одной десятичной цифре.



Рис 2.1. Загрузка "программы" (коммутационной панели) в табулятор UNIVAC 120 (Remington Rand 409), [Weik 1961] Справа виден стеллаж с набором "программ"

Все устройства ENIAC имели собственные управляющие устройства с собственными коммутационными панелями и, теоретически, могли работать параллельно. Таким образом, ENIAC в определенном смысле был похож на современные суперскалярные процессоры. Координация работы машины в целом осуществлялась с помощью главного управляющего устройства (*master programmer*), который имел десять шестиразрядных счетчиков и собственную коммутационную панель.

"Перепрограммирование" всех этих панелей могло занимать несколько дней, при том, что исполнявшиеся на ENIAC программы были, по современным меркам, довольно простыми. Машина первоначально использовалась для баллистических расчетов (численного решения дифференциальных уравнений движения тела в поле тяжести с учетом сопротивления воздуха), позднее на ней делались некоторые расчеты для атомного проекта.

В 1947 году Дж. фон Нейман предложил радикально изменить технологию программирования машины. А именно, он предложил разработать схему коммутации, которая могла бы обеспечить интерпретацию последовательностей команд, коды которых должны были храниться в одной из "функциональных таблиц". Один из аккумуляторов при этом использовался для хранения номера (или, говоря современным языком, адреса) ячейки, из которой следовало извлечь следующую команду. Запись значений в этот аккумулятор приводила к передаче управления в, вообще говоря, произвольную точку программной памяти.

Переход к новой технологии лишил программиста возможности использовать заложенный в архитектуре ENIAC параллелизм — команды исполнялись строго последовательно, хотя некоторые из них и действовали несколько исполните-

тельных устройств одновременно. Поэтому производительность компьютера (измеряемая в количествах сложений в секунду) упала приблизительно в шесть раз, но время перепрограммирования уменьшилось с одного дня до нескольких часов. Главным достоинством новой техники оказался тот факт, что радикально упростилась проверка и отладка программ, так что это позволило автору отчета [Clippinger 1948] впервые написать слова, которые затем лишь с небольшими вариациями произносились при каждом значительном прорыве в технологиях программирования: "Автор надеется, что этот доклад сделает работу по кодированию задач настолько ясной и прямой, что физики, аэродинамики, прикладные математики и другие, не имеющие опыта работы с вычислительными машинами, смогут кодировать собственные задачи и готовить тесты".

В действительности, спор о том, кто именно изобрел компьютеры с программной памятью произвольного доступа, не прекращался до 60-х годов прошлого столетия и так и не был удовлетворительно разрешен. Приоритет в деле публикации этой идеи, безусловно, принадлежит фон Нейману [Goldstine/Neumann 1947], однако Эккерт и Мочли утверждали, что идея хранимой программы пришла им в голову во время работ над ENIAC. Это подтверждается тем фактом, что следующий их компьютер (EDVAC), введенный в эксплуатацию в 1949 году, сразу разрабатывался как последовательное устройство с хранимой программой.

2.2. Форматы команд машинного языка

Каждая отдельная команда машинного языка представляет собой последовательность бит. У процессоров, минимальной единицей адресации ОЗУ которых является слово, длина команды обычно равна размеру слова или целому числу слов. У процессоров с байтовой адресацией длина команды равна целому числу байт. У некоторых процессоров — например, у PDP-11 или SPARC — все команды имеют одинаковую длину, конкретно у PDP-11 — два байта, а у SPARC — четыре. У других процессоров, например, у DEC VAX и x86, различные команды имеют различную длину, от одного до десятка (а у VAX и до нескольких десятков) байт. Вершиной экстравагантности среди когда-либо реализованных "в железе" архитектур следует признать процессор Intel 432, который так и не пошел в серийное производство [Органик 1987], — у этого процессора длина команд была не только переменной, но и не кратной байту, так что в командах переходов приходилось указывать адреса с точностью до бита.

Команда центрального процессора состоит из *кода операции* (operation code, opcode) — кода, т. е., попросту говоря, номера команды, и одного или нескольких *операндов* (описателей объектов, над которыми совершается операция). То и другое кодируется битовыми полями команды. Операнд обычно содержит идентификатор режима адресации (это понятие разъясняется далее

в этой главе) и дополнительные поля, смысл которых зависит от режима адресации. Как правило, среди этих полей есть номера одного или нескольких регистров. Во многих режимах адресации код операнда может также содержать адресные смещения или непосредственные значения (в зависимости от команды, непосредственное значение может представлять собой целое число или число с плавающей точкой).

В некоторых системах команд эти битовые поля выровнены на границу байта, в большинстве других это битовые поля произвольных размеров. Во многих архитектурах с переменной длиной команды поле кода операции само имеет переменную длину. Обычно часто используемые команды стараются кодировать короткими кодами, а редко используемые — длинными. На практике, однако, длина кода команды часто свидетельствует не о том, как часто команда употребляется, а о том, когда эта команда была добавлена к архитектуре. Так, у x86 базовый набор команд, соответствующий командам 8086, имеет однобайтовые коды, а расширенные команды, появившиеся в 80486 (некоторые управляющие команды), Pentium MMX ("мультимедийные") и Pentium III (SIMD и SSE), имеют коды операции, занимающие два и даже три байта.

В PDP-11 и x86 размер большинства битовых полей кратен трем, так что при восьмеричной записи кода команды отдельные восьмеричные цифры или их группы соответствуют определенным полям. К сожалению, у x86 восьмеричная запись кодов команд не прижилась — в документации Intel используется шестнадцатиричная запись, и практически все ассемблеры и деассемблеры для x86 следуют этой традиции (примеры 2.1, 2.2). Мало кто может переводить в уме многобайтовые коды из шестнадцатиричной записи в восьмеричную, поэтому схему кодирования команд x86 ошибочно считают трудной для запоминания, а схему кодирования команд PDP-11 — напротив, легкой.

Пример 2.1. Машинный код для процессора x86 (двоичное представление)

.00010070:	55 89 E5
.00010080: 81 EC 28 01-00 00 8D 85-E8 FE FF FF-50 8D 45 E8	
.00010090: 50 68 38 00-01 00 FF 75-08 E8 8A 05-00 00 83 C4	
.000100A0: 10 83 EC 08-68 3F 00 01-00 8D 45 E8-50 E8 6E 04	
.000100B0: 00 00 83 C4-10 85 C0 75-26 83 EC 04-8D 85 E8 FE	
.000100C0: FF FF 50 68-44 00 01 00-FF 75 08 E8-A8 04 00 00	
.000100D0: 83 C4 10 C7-85 E4 FE FF-FF C8 00 00-00 EB 21 83	
.000100E0: EC 04 8D 45-E8 50 68 65-00 01 00 FF-75 08 E8 85	
.000100F0: 04 00 00 83-C4 10 C7 85-E4 FE FF FF-F4 01 00 00	
.00010100: 8B 85 E4 FE-FF FF C9 C3	

**Пример 2.2. Машинный код для процессора x86
(декодированное представление)**

```

.0001007D: 55          push    ebp
.0001007E: 89E5        mov     ebp,esp
.00010080: 81EC28010000 sub     esp,000000128
.00010086: 8D85E8FFFFF lea     eax,[ebp][0FFFFFFEE8]
.0001008C: 50          push    eax
.0001008D: 8D45E8      lea     eax,[ebp][-0018]
.00010090: 50          push    eax
.00010091: 6838000100  push    000010038
.00010096: FF7508      push    d,[ebp][00008]
.00010099: E88A050000  call    .000010628 ----- (1)
.0001009E: 83C410      add    esp,010
.000100A1: 83EC08      sub    esp,008
.000100A4: 683F000100  push    00001003F
.000100A9: 8D45E8      lea     eax,[ebp][-0018]
.000100AC: 50          push    eax
.000100AD: E86E040000  call    .000010520 ----- (2)
.000100B2: 83C410      add    esp,010
.000100B5: 85C0          test   eax,eax
.000100B7: 7526          jne    .0000100DF ----- (3)
.000100B9: 83EC04      sub    esp,004
.000100BC: 8D85E8FFFFF lea     eax,[ebp][0FFFFFFEE8]
.000100C2: 50          push    eax
.000100C3: 6844000100  push    000010044
.000100C8: FF7508      push    d,[ebp][00008]
.000100CB: E8A8040000  call    .000010578 ----- (4)
.000100D0: 83C410      add    esp,010
.000100D3: C785E4FEFFFFC8000000 mov    d,[ebp][0FFFFFFEE4],0000000C8
.000100DD: EB21          jmps   .000010100 ----- (5)
.000100DF: 83EC04      sub    esp,004
.000100E2: 8D45E8      lea     eax,[ebp][-0018]
.000100E5: 50          push    eax
.000100E6: 6865000100  push    000010065
.000100EB: FF7508      push    d,[ebp][00008]
.000100EE: E885040000  call    .000010578 ----- (6)
.000100F3: 83C410      add    esp,010
.000100F6: C785E4FEFFFFF4010000 mov    d,[ebp][0FFFFFFEE4],0000001F4
.00010100: 8B85E4FEFFF lea     eax,[ebp][0FFFFFFEE4]
.00010106: C9          leave
.00010107: C3          retn

```

Примечание

В примере 2.1 приведен код в том виде, в каком он хранится в памяти, в шестнадцатиричной записи.

В примере 2.2 код разбит на команды и указаны их мнемонические обозначения; разбиение осуществлялось деассемблером *hiew*, который использует синтаксис Intel).

У процессоров VAX битовые поля команд выровнены на четыре бита (шестнадцатиричную цифру), поэтому систему команд VAX иногда называют шестнадцатиричной, а PDP-11 и x86 — восьмеричными. RISC-процессоры обычно имеют 32 регистра, так что и восьмеричная, и шестнадцатиричная схемы кодирования неудобны для представления их команд.

Команда центрального процессора состоит из кода операции и одного или нескольких *операндов* (объектов, над которыми совершается операция). В зависимости от числа operandов, команды делятся на *безадресные* (не имеющие operandов или имеющие неявно указанные), *одноадресные* (производящие операцию над одним объектом или одним явно и одним или несколькими неявно указанными), *двух- и трехадресные*. Встречаются архитектуры, в которых есть команды и с большим числом operandов, но это экзотика.

Примеры безадресных команд без операнда.

NOP

No OPeration, отсутствие операции.

HALT

Остановка процессора.

Примеры безадресных команд с неявно указанными operandами.

RETURN

Возврат из подпрограммы. Выталкивает из стека адрес возврата и помещает его в счетчик команд.

WDR

WatchDog Reset, сброс сторожевого таймера микроконтроллера.

ADD

Вытолкнуть из стека два значения, сложить их и протолкнуть результат в стек.

SCS

Skip if Carry Set, пропустить следующую команду, если бит переноса в слове состояния установлен.

Примеры одноадресных команд с одним операндом.

INC x

INCrement, добавить к операнду 1 и сохранить результат по тому же адресу.

TST x

TeST, установить в слове состояния флаги знака и равенства нулю в соответствии со значением операнда.

Примеры одноадресных команд с неявным операндом.

ADD x [, Acc]

Сложить operand с аккумулятором и сохранить результат в аккумуляторе.

PUSH x

Протолкнуть значение operand'a в стек.

CALL x

Вызов подпрограммы, сохраняет адрес следующей команды в стеке и передает управление по указанному адресу.

BNEQ x

Передает управление по указанному адресу, если в слове состояния не установлен флаг равенства нулю.

Примеры двух- и трехадресных команд.

MOVE x, y

Присвоить значение объекта x объекту y.

ADD x, y

Сложить x и y, поместить результат в y.

ADD x, y, z

Сложить x и y и поместить результат в z.

Четырехадресная команда:

DIV x, y, z, w

выполняет деление x на y, помещает частное в z, а остаток — в w.

Шестиадресная команда:

INDEX b, l, h, s, i, a

вычисляет адрес элемента массива, расположенного по адресу b, с нижней и верхней границами индекса l и h соответственно и размером элемента s. Операнд i — индекс элемента, a — место, куда следует поместить вычисленный адрес.

Количество адресов иногда используют и для общей характеристики системы команд. Двухадресной называют систему команд, в которой команды имеют максимум два операнда, трехадресной — максимум три. Нередко, впрочем, вместо максимального количества операндов адресность системы команд определяют по количеству операндов у наиболее "ходовых" команд — сложения и вычитания. Таким образом, VAX, из системы команд которого взяты примеры четырех- и шестиадресных команд, часто относят к трехадресным архитектурам.

Одноадресную аккумуляторную архитектуру имеют микроконтроллеры семейства RISC фирмы Microchip. Такая архитектура была типична для компьютеров первых поколений, разрабатывавшихся в 50-е и начале 60-х годов. Большинство современных процессоров имеют двух- и трехадресные системы команд.

Легко догадаться, что архитектуры с большим числом операндов у команд чаще используют команды переменной длины. Впрочем, есть много исключений из этого правила. Так, IBM System 360, Motorola 680x0, Intel 8086, x86 — двухадресные архитектуры с переменной длиной команд, а RISC-процессоры — трехадресные архитектуры с постоянной длиной команд.

Одноадресные системы команд обычно используют в качестве неявно заданного операнда выделенный регистр, так называемый *аккумулятор*, или *стек*. Такие архитектуры называют, соответственно, *аккумуляторными* и *стековыми*.

2.2.1. Стековые системы команд

Йоды магистра речи раскрыта тайна. На Форте программист старый он есть просто.

На примере стековой команды ADD мы видели, что многие из команд стековой архитектуры могут обойтись вообще без явно указанных операндов, однако команды проталкивания значений переменных в стек и выталкивания их оттуда все-таки необходимы, поэтому все стековые архитектуры одно-, а не безадресные.

Стеки привлекательны, во-первых, тем, что не нуждающиеся в операндах команды могут иметь очень короткий код операции (как правило, достаточно одного байта) и, во-вторых, тем, что работающая с ними программа представляет собой арифметическое выражение, записанное в *обратной польской нотации* — когда мы сначала пишем операнды, а потом знак операции. Например, операция $a+b$ в этой записи выглядит как $ab+$ (в программе — push a; push b; add;).

Задача преобразования привычных нам арифметических выражений в обратную польскую запись легко формализуется, поэтому стековые процессоры долгое время позиционировались как "ориентированные на языки высокого уровня". Позже, впрочем, выяснилось, что более сложная логика разбора арифметических выражений позволяет проводить разного рода оптимизации (сокращать введенные лишь для удобства записи переменные, заменять выражения, которые всегда дают одно и то же значение, на константы, выносить повторяющиеся вычисления из тела цикла и т. д.).

Аппаратно реализованные стековые архитектуры — в наше время редкость. Из относительно современных процессоров, имевших коммерческий успех, можно назвать Transputer фирмы Inmos (в настоящее время эти микропроцессоры выпускаются фирмой SGC-Thomson).

Шире всего стековая архитектура распространена в *байт-кодах* или, как это еще называют, системах команд виртуальных машин. Байт-код — это промежуточное представление программы, используемое интерпретатором, чтобы избежать лексического и синтаксического анализа программы на этапе исполнения. Исполнение байт-кода осуществляется не процессором, а программой-интерпретатором. Таким образом, реализуются многие современные языки программирования — многочисленные диалекты языка BASIC, Lisp, SmallTalk, Fort (этот язык любопытен тем, что сам имеет стековый синтаксис), наконец, Java. Некоторые реализации интерпретаторов этих языков используют так называемую *JIT-компиляцию* (Just In Time, точно в момент [исполнения]), когда перед исполнением байт-код компилируется в систему команд физического процессора. Такая технология позволяет достичь для "интерпретируемых" программ производительности, не уступающей производительности компилированного кода.

Первым промышленным применением JIT-компиляции была система AS/400 фирмы IBM. В настоящее время JIT широко используется в реализациях Java. JIT-компиляция привлекательна тем, что позволяет выполнять один и тот же код на разнообразных процессорах без потерь (или почти без потерь) скорости.

2.3. Команды перехода

Как говорилось в разд. 2.1, команды условного перехода — это то, что отличает фон-неймановский процессор от непроцессора или, в крайнем случае, от не фон-неймановского процессора. Большинство современных процессоров имеет обширный набор команд условного перехода по различным арифметическим условиям и их комбинациям.

Арифметические флаги выставляются в соответствии с результатами последней арифметической и логической операции. Типичный набор арифметиче-

ских флагов — это бит переноса, бит нуля (выставляется, если все биты результата равны нулю), знаковый бит (если равен нулю старший бит результата) и бит переполнения. В процессорах первых поколений нередко использовался обратный подход: процессор имел всего один флаг условия перехода (так называемый *ω-признак*) и одну команду условного перехода, зато несколько команд сравнения, придававших этому флагу различную семантику. Набор команд перехода, приведенный в табл. 2.1, несколько шире обычного — команды SBRC/SBRS для процессоров общего назначения нетипичны.

Микроконтроллеры PIC (по-видимому, самая экстравагантная система команд среди современных промышленно выпускаемых процессоров) имеют всего две команды, выполняющие функции команд условного перехода: BTFCs (Bit Test, Skip if Set, проверить бит и, если он установлен, пропустить следующую команду) и BTFFC (Bit Test, Skip if Clear, пропустить следующую команду, если бит сброшен). Объектом проверки может служить произвольный бит любого регистра процессора, в том числе и биты арифметических условий статусного слова. Для реализации условного перехода следом за такой командой нужно разместить команду безусловного перехода. Именно таким образом ассемблеры для этого микроконтроллера реализуют псевдокоманды условных переходов.

Таблица 2.1. Команды условного перехода микроконтроллеров семейства AVR, цит. по [www.atmel.com]

Команда	Описание	Условие перехода
SBRC Rr, b	Пропустить, если бит в регистре сброшен	if (Rr(b)=0) PC = PC + 2 or 3
SBRS Rr, b	Пропустить, если бит в регистре установлен	if (Rr(b)=1) PC = PC + 2 or 3
SBIC P, b	Пропустить, если бит в регистре B/B сброшен	if(I/O(P,b)=0) PC = PC + 2 or 3
SBIS P, b	Пропустить, если бит в регистре B/B установлен	if(I/O(P,b)=1) PC = PC + 2 or 3
BRBS s, k	Перейти, если статусный флаг установлен	if (SREG(s) = 1) PC = PC+k + 1
BRBC s, k	Перейти, если статусный флаг сброшен	if (SREG(s) = 0) PC = PC+k + 1
BREQ k	Перейти, если равно	if (Z = 1) PC = PC + k + 1
BRNE k	Перейти, если не равно	if (Z = 0) PC = PC + k + 1
BRCS k	Перейти, если перенос установлен	if (C = 1) PC = PC + k + 1

Таблица 2.1 (окончание)

Команда	Описание	Условие перехода
BRCC k	Перейти, если перенос сброшен	if (C = 0) PC = PC + k + 1
BRSH k	Перейти, если равно или выше	if (C = 0) PC = PC + k + 1
BRLO k	Перейти, если ниже	if (C = 1) PC = PC + k + 1
BRMI k	Перейти, если минус	if (N = 1) PC = PC + k + 1
BRPL k	Перейти, если плюс	if (N = 0) PC = PC + k + 1
BRGE k	Перейти, если больше или равно, знаковое	if (N XOR V= 0) PC = PC+ k + 1
BRLT k	Перейти, если меньше, знаковое	if (N XOR V= 1) PC = PC + k + 1
BRHS k	Перейти, если полуbyte-вый перенос установлен	if (H = 1) PC = PC + k + 1
BRHC k	Перейти, если полуbyte-вый перенос сброшен	if (H = 0) PC = PC + k + 1
BRTS k	Перейти, если T-флаг установлен	if (T = 1) PC = PC + k + 1
BRTC k	Перейти, если T-флаг сброшен	if (T = 0) PC = PC + k + 1
BRVS k	Перейти, если флаг переполнения установлен	if (V = 1) PC = PC + k + 1
BRVC k	Перейти, если флаг переполнения сброшен	if (V = 0) PC = PC + k + 1
BRIE k	Перейти, если прерывания разрешены	if (I = 1) PC = PC + k + 1
BRID k	Перейти, если прерывания запрещены	if (I = 0) PC = PC + k + 1

2.4. Регистры

Процессор соединен с банками памяти шиной, по которой за один раз передается только одно целое число. Иногда разрядность этой шины тоже называют разрядностью процессора. Тогда 16-разрядный i8088 оказывается 8-разрядным, 32-разрядные MC68000 и i80386SX — 16-разрядными, Pentium — 64-разрядным, Pentium IV — 128-разрядным, а младшие модели современных

64-разрядных RISC-процессоров — 32-разрядными. Для разработчиков аппаратуры такая классификация имеет смысл, а разработчиков программного обеспечения она может ввести в заблуждение.

Арифметико-логическое устройство процессора обычно не может оперировать данными, непосредственно размещенными в оперативной памяти: для выполнения арифметической операции нужен доступ одновременно к трем ячейкам памяти, хранящим операнды и результат.

Для решения этой проблемы любой процессор имеет один или несколько *регистров* — специализированных запоминающих устройств, обычно вмещающих целое число или адрес. Все процессоры имеют как минимум шесть регистров — регистр для адреса текущей команды (счетчик команд, program counter или instruction pointer), регистр флагов, где хранятся коды арифметических условий и, кроме того, много другой служебной информации (часто этот регистр называют словом состояния процессора, PSW — Processor Status Word), три буферных регистра АЛУ и буферный регистр, в котором хранится текущая команда (рис. 2.2).

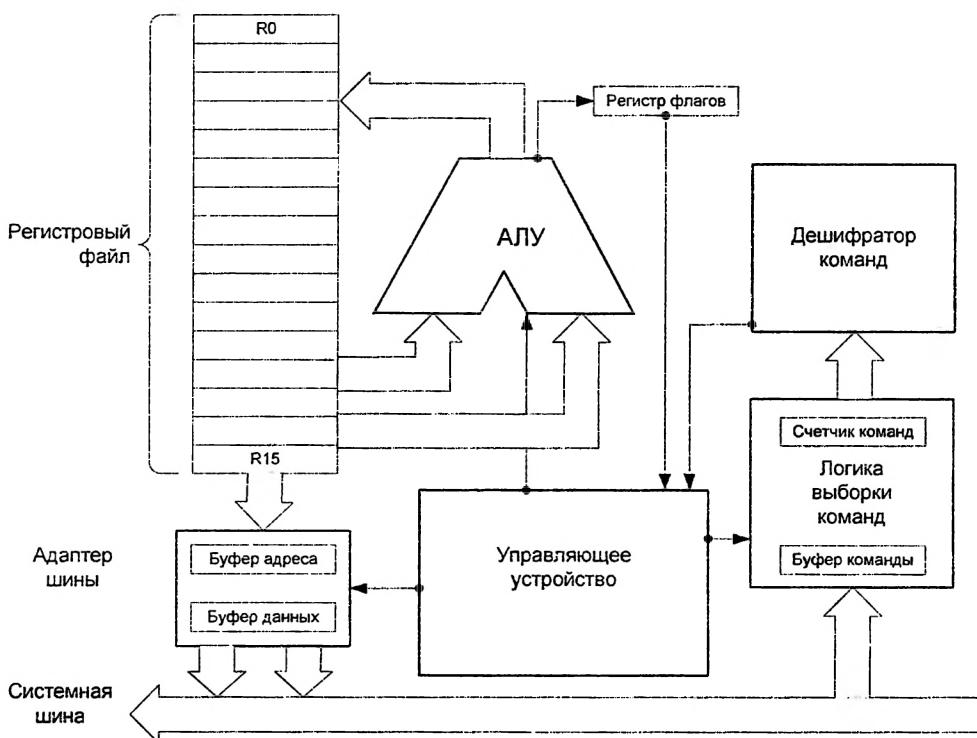


Рис. 2.2. Типичная структура микропроцессора

Из этих регистров программисту доступны только счетчик команд и слово состояния процессора, да и то не всегда. (Под доступностью программисту в данном случае мы подразумеваем возможность указывать регистры в качестве явных и неявных операндов команд.)

В старых процессорах регистры делились на два класса: *арифметические регистры* (arithmetic register), используемые для хранения данных, и индексные (index register), используемые для хранения указателей.

Регистры БЭСМ-6

Компьютер БЭСМ-6, разработанный в середине 60-х годов в ИТМИВТ им. Лебедева, имел архитектуру, довольно типичную для процессоров второго поколения. БЭСМ-6 имела одноадресную систему команд регистр-память. В качестве неявного операнда одноадресных команд использовался 48-разрядный регистр-аккумулятор, хранивший числа с плавающей либо фиксированной точкой. Формат числа определялся флагом в слове состояния процессора: при выключенной нормализации число считалось числом с фиксированной точкой, при включенной — соответственно, с плавающей. Операции над целыми числами реализовались как операции над числами с фиксированной точкой.

Некарактерная для компьютеров второго поколения особенность БЭСМ-6 состояла в том, что аккумулятор мог использоваться не только в качестве собственно аккумулятора, но и в качестве вершины стека. Кроме простых арифметических команд над ним были определены также команды со стековой семантикой, проталкивание, выталкивание и арифметические операции с выталкиванием operandов из стека. Стек размещался в памяти, но верхние шесть слов стека кэшировались в скрытых регистрах процессора.

Кроме арифметического аккумулятора, процессор имел 15 индексных регистров. Регистр с номером 16 использовался в качестве указателя стека. Регистра с номером 0 как такового не существовало, но этот номер можно было использовать в командах — в результате получалась адресация относительно адреса 0. Индексные регистры имели 15 разрядов. Таким образом, при том, что основное АЛУ и слово данных имели 48 разрядов, процессор БЭСМ-6 мог адресовать 32к слов. В действительности, ограничение в 32к касалось только одной задачи, система имела страничный диспетчер памяти и общий объем ОЗУ мог достигать 512к слов.

В современных процессорах разрядности адреса и основного АЛУ обычно совпадают, и также совпадают представления целых чисел и адресов, поэтому данные (во всяком случае, целочисленные) и адреса можно хранить в одних и тех же регистрах. Такие регистры совмещают в себе функции арифметических и индексных и называются *registramи общего назначения* (general purpose register).

Регистры процессора x86

Процессоры x86 имеют восемь 32-разрядных регистров общего назначения. Процессор имеет неортогональную систему команд (понятие ортогональности определяется далее в этом разделе), так что не все регистры доступны во всех командах, поэтому регистры обозначаются не номерами, а буквенными кодами

(рис. 2.3). При этом младшие разряды регистров доступны под отдельными именами: так, младшие 16 бит регистра `eah` доступны в качестве 16-разрядного регистра `ah` либо двух 8-разрядных регистров `al` и `ah`. 16- и 8-разрядные регистры соответствуют одноименным регистрам 8086/80286 и могут использоваться для обеспечения совместимости с этими процессорами по языку ассемблера. Однобайтовые регистры `ah`, `al`, `bh`, `bl` и т. д. соответствуют регистрам 8-разрядных процессоров 8008 и 8080, так что совместимость по языку ассемблера простирается на два поколения назад.

Расшифровки обозначений регистров также восходят к началу 70-х годов, к процессору 8008, в котором `A` действительно был аккумулятором, регистровая пара `BH` — 16-разрядным базовым (индексным) регистром, регистры `bh` и `bl` использовались для записи значений в этот регистр и т. д.

По мере перехода к процессорам следующих поколений — 8080 и 8086 — различия между регистрами постепенно стирались. Например, 8008 и 8080 имели одноадресную архитектуру с аккумулятором, а 8086 — двухадресную регистр-память, так что практически любой из регистров общего назначения мог использоваться в арифметических операциях. Однако основные арифметические команды, использующие два произвольных регистра или режим регистр-память в 8086 кодировались двумя байтами, а операции между `ah` и другим регистром — одним байтом.

Процессор x86 уже довольно близок к полной ортогональности: в частности, у 80286 еще нельзя было использовать `ah` в качестве адресного регистра, а `eah` у процессора x86 — можно, так что только у x86 эти регистры вполне заслуживают названия "регистры общего назначения" — однако такое обозначение использовалось уже в документации по 8086.

Кроме регистров общего назначения, x86 имеет восемь 80-разрядных арифметических регистров, используемых при операциях с плавающей точкой одинарной (32 бита), двойной (64 бита) и расширенной (80 бит) точности. В старых процессорах, от 8086 до 80386 включительно, эти регистры располагались на отдельной микросхеме, так называемом арифметическом сопроцессоре. Эта микросхема еще и реализовала команды, работающие с плавающей точкой, и команды передачи данных в свои регистры. Маркировка данной микросхемы соответствовала маркировке основного процессора — к процессору 8086 полагался сопроцессор 8087, к 80286 — 80287 и т. д. При отсутствии такой микросхемы процессор мог выполнять операции с плавающей точкой лишь с помощью программной эмуляции, т. е. от нескольких раз до нескольких сот раз медленнее.

При переходе к семейству Pentium арифметический сопроцессор стал неотъемлемой частью процессора. Позднее, в Pentium III, к процессору было добавлено еще восемь 128-разрядных регистров и команды, позволявшие совершать над этими регистрами векторные операции, интерпретирующие каждый регистр как четыре числа с плавающей точкой одинарной точности или два числа двойной точности, так называемые команды SSE (Streaming SIMD Extension, потоковое расширение SIMD). Аббревиатура SIMD, в свою очередь, расшифровывается как Single Instruction Many Data — одна команда, много данных, т. е., говоря попросту, это векторная операция. Такие операции могут применяться при реализации алгоритмов упаковки с потерями на основе Фурье-преобразования, при трехмерных расчетах и в ряде других приложений.

Кроме арифметических и индексных регистров или регистров общего назначения, процессор обычно имеет множество других регистров. Некоторые из них интересны только проектировщикам аппаратуры, с другими — например, регистрами диспетчера памяти, — мы еще встретимся в следующих главах. Некоторые из этих регистров не доступны программисту, некоторые другие — доступны лишь при использовании специальных команд.

Так, у x86 кроме регистров общего назначения и арифметических регистров сопроцессора есть еще слово состояния процессора (PSW), счетчик команд, шесть так называемых "сегментных" регистров и некоторое (различное у разных процессоров) количество управляемых регистров, доступ к которым осуществляется специальными командами. Подробнее система команд и номенклатура регистров x86 описывается в *приложении 2*.

Для доступа к регистрам процессору не нужно занимать внешнюю шину данных, да и цикл доступа к регистру обычно очень короток и совпадает с циклом работы АЛУ. Следовательно, чем больше у процессора регистров, тем быстрее он может работать с оперативными данными. Кроме того, если команда использует регистры, то ей не требуются дополнительные поля (адресные смещения и т. д.), поэтому команды, работающие с регистрами, оказываются короче команд, работающих с оперативной памятью. Но здесь выигрыш уже оказывается не столь однозначным: если у процессора много регистров, то соответствующие битовые поля команд тоже необходимо увеличить, т. е. команда удлиняется. Тем не менее в целом все равно получается, что у процессоров с большим количеством регистров код не только быстрее, но и несколько компактнее, чем у процессоров с малым их количеством. Особенно это заметно на 64-разрядных процессорах, у которых длина адресных смещений в командах может достигать восьми байт.

В те времена, когда компьютеры были большими, стремление к увеличению количества регистров упиралось в стоимостные и электротехнические ограничения. У компьютеров первых поколений для реализации регистров использовались отдельные транзисторы или микросхемы малой степени интеграции, поэтому они стоили гораздо дороже (в расчете на один бит памяти), потребляли гораздо больше энергии и рассеивали гораздо больше тепла, чем ферритовая память. Переход на современные кристаллы высокой интеграции изменил положение, но не принципиально: триггеры, которые используются для реализации регистров, по перечисленным параметрам всегда хуже, чем исполненное по той же технологии динамическое ОЗУ. Поэтому компьютеры первых поколений обычно имели лишь несколько регистров, а мини- и микроКомпьютеры 70-х и начала 80-х годов прошлого века — не более нескольких десятков.

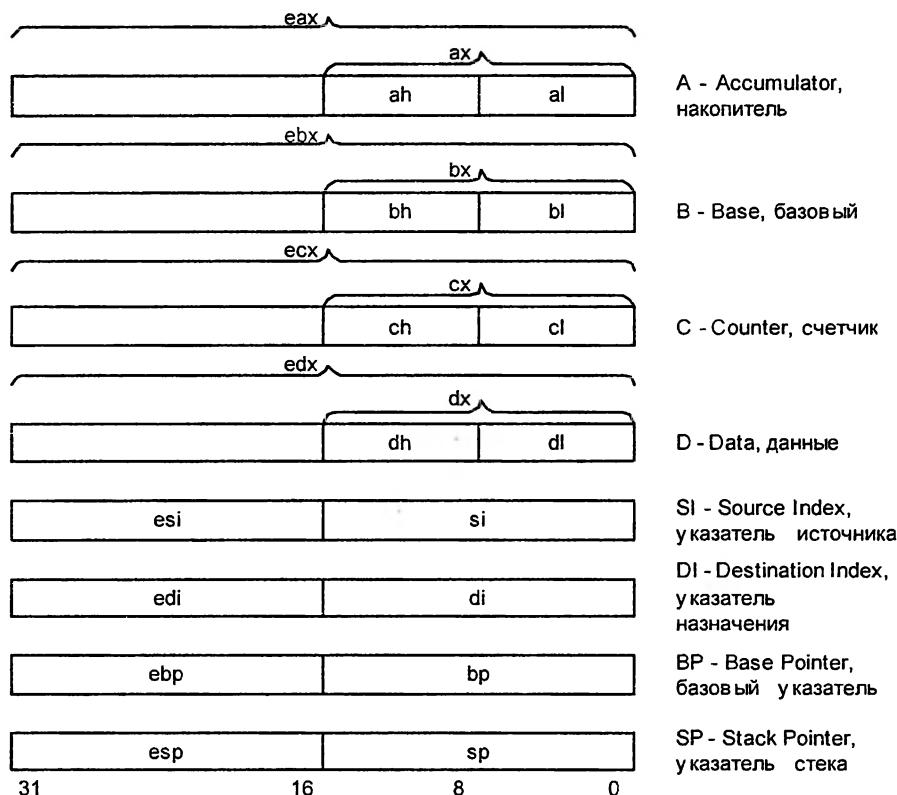


Рис. 2.3. Регистры общего назначения в системе команд x86

У современных процессоров количество регистров измеряется сотнями, а иногда и тысячами. Например, вместо буфера на одну только текущую команду, все без исключения современные процессоры имеют так называемую очередь предварительной выборки команд. У микроконтроллеров PIC и Atmel то, что в спецификациях указано как ОЗУ (у младших моделей 128 или 256 байт, а у старших — много килобайт), фактически представляет собой регистры.

Регистровое окно SPARC

RISC-процессор SPARC [[www.sparc.com v9](http://www.sparc.com/v9)] имеет регистровый файл, объем которого у старых версий процессора составлял 136 32-разрядных слов, а у современных 520 и более. Этот файл состоит из групп по 8 регистров.

Программе одновременно доступно 32 регистра, нумеруемые от 0 до 31, называемые регистровым окном. Из них регистр 0 (обозначаемый как %g0) — выделенный, чтение из него всегда возвращает 0. Следующие 7 регистров (обозначаемые как %g1—%g7) используются для глобальных переменных. Эти регистры

не входят в регистровое окно. Регистры с $r18$ по $r15$ ($%o0$ — $%o7$) используются для передачи параметров вызываемой процедуре ($r15$ применяется для хранения адреса возврата подпрограммы), регистры с $r24$ по $r31$ ($%i0$ — $%i7$) — для доступа к параметрам, и, наконец, регистры с $r16$ по $r23$ ($%l0$ — $%l7$) — для хранения локальных переменных. При вызове подпрограммы команда `Save` сдвигает регистровое окно, так что регистры $%o0$ — $%o7$ вызывающей процедуры становятся регистрами $%i0$ — $%i7$ процедуры вызванной. Регистры $r0$ — $r7$ сдвигом регистрового окна не затрагиваются.

Таким образом, регистровый файл выполняет функции стека вызовов — в нем сохраняются адреса возврата, передаются параметры и хранятся локальные переменные. Впрочем, такой подход при всей его привлекательности имеет большой недостаток — глубина стека вызовов оказывается ограничена глубиной регистрового файла. Способ устранения этого ограничения описан в разд. 2.6.4.

Важно еще раз отметить, что обычно далеко не все имеющиеся регистры непосредственно доступны программе. Так, в системе команд `x86` предусмотрено всего восемь регистров общего назначения, но современные суперскалярные реализации этой архитектуры (`Pentium II`, `Athlon` и т. д.) имеют более сотни физических регистров и несколько арифметико-логических устройств. Логика предварительной выборки дешифрует несколько команд и назначает исполнение этих команд на свободные АЛУ, по мере их поступления, с использованием свободных копий регистров, обеспечивая только сохранение связей по данным между последовательными командами, но не соответствие порядка исполнения команд порядку, в котором они встречаются в коде.

При наличии большого количества регистров существует два подхода к их обозначению в командах. Первый состоит в том, что каждый код операции работает только со своим регистром. Команда `move %eax, %ebx` — это одна команда, а `move %eax, %esp` — совсем другая. При этом бывает и так, что некоторые команды для отдельных регистров не определены.

Второй подход, более мне симпатичный, называется *ортогональной системой команд*, и состоит в том, что все команды (или хотя бы большинство) могут работать с любыми регистрами. Номер регистра при этом кодируется специальным битовым полем в коде команды (или несколькими битовыми полями, если команда оперирует несколькими регистрами) (рис. 2.4). При всех недостатках первого подхода, ортогональная система команд создает специфическую проблему: увеличение количества регистров оказывается невозможным. Ведь это требует расширения битового поля номера регистра, а изменение размера битовых полей команд — это нарушение бинарной совместимости. Впрочем, в разд. 8.2 мы поймем, почему даже в неортогональных системах команд расширение номенклатуры регистров практически невозможно.

CALL

OP	30-битное смещение		
31 29			0

SETHI и команды условного перехода

OP	rd или условие	OP1	22-разрядная константа или смещение		
31 29		24 21			0

Арифметико-логические команды и LD/ST

OP	rd	OP2	rs1	0	XXXXXXXXXX		rs2
OP	rd	OP2	rs1	1	XXXXXXXXXX		4 0

OP[12] - код операции

rs1 - исходный регистр 1

rs2 - исходный регистр 2

rd - регистр назначения

Рис. 2.4. Формат команды микропроцессора SPARC (цит. по [www.sparc.com v9])

2.5. Адресация оперативной памяти

С точки зрения процессора, оперативная память представляет собой массив пронумерованных ячеек. Номер каждой ячейки памяти называется ее *адресом*.

Очень важно понимать, что многие понятия ЯВУ и ассемблера, такие как процедуры, метки, переменные, структуры и их поля и т. д. — с точки зрения процессора не существуют в природе. Все эти объекты существуют только в "воображении" компилятора и, иногда, некоторых вспомогательных инструментов (например, компоновщика) — ну и, разумеется, в воображении (уже без кавычек) программиста, пишущего или читающего программу. При компиляции все эти объекты отображаются в участки кода или ячейки памяти, выделенные для хранения данных, а ссылки на них — в адреса соответствующих ячеек памяти. Информация о том, какой переменной соответствует та или иная ячейка, при компиляции обычно отбрасывается, а если и сохраняется, то только в отладочных целях (подробнее данный вопрос рассматривается в разд. 2.10). Одной из важнейших характеристик процессора и реализуемой им системы команд является разрядность адреса. Разрядность важна не как самоцель, а потому, что ею обусловлен объем пространства допустимых значений адреса.

мых адресов или, что то же самое, *адресного пространства*. Системы с 16-разрядным адресом способны адресовать 64 Кбайт (65 536) ячеек памяти, а с 32-разрядным — 4 Гбайт (4 294 967 296) ячеек. В наше время адресуемая память в 4 Гбайт для многих приложений считается неприемлемо маленькой и требуется 64-разрядная адресация.

Процессору обычно приходится совершать арифметические операции над адресами, поэтому разрядность адреса у современных процессоров обычно совпадает с разрядностью основного АЛУ. Как мы видели в предыдущем разделе, у компьютеров первых поколений это не всегда было так.

У некоторых компьютеров адресация (нумерация) ячеек памяти фиксированная: одна и та же ячейка памяти всегда имеет один и тот же номер. Такая адресация называется физической. Адрес при этом разбит на битовые поля, которые непосредственно используются в качестве номера физической микросхемы памяти, а также номеров строки и столбца в данной микросхеме. Напротив, большинство современных процессоров общего назначения используют *виртуальную адресацию*, когда номер конкретной ячейки памяти определяется не физическим размещением этой ячейки, а контекстом, в котором происходит адресация. Способы реализации виртуальной памяти и необходимость ее применения обсуждаются в главе 5.

В старых компьютерах размер адресуемой ячейки памяти данных совпадал с разрядностью АЛУ центрального процессора и разрядностью шины данных. Адресуемая ячейка называлась *словом*. В процессорах манчестерской архитектуры, которые могут использовать одну и ту же память как для команд, так и для данных, оба размера определялись длиной команды. Из-за этого многие процессоры такого типа имели странные по современным представлениям разрядности — 48, 36, иногда даже 25 бит.

БЭСМ-6

Так, БЭСМ-6 имела слово разрядностью 48 бит и команды длиной 24 бита, состоявшие из 15-разрядного адресного поля и 9-разрядного кода операции. Адресное поле позволяло адресовать 32к слов. В одном слове размещалось две команды, при этом команды перехода могли указывать только на первую из упакованных в одно слово команд.

Использование 36-разрядных слов связано также с тем, что 36 двоичных разрядов довольно точно соответствуют 10 десятичным. Такая точность в середине XX столетия считалась достаточной для инженерных и научных вычислений. Многие компьютеры первых поколений, начиная с ENIAC, были 36-разрядными.

У процессоров гарвардской архитектуры (имеющих раздельные памяти для команд и данных) разрядность АЛУ и размер команды не связаны.

Microchip PIC

Микроконтроллеры семейства PIC фирмы Microchip имеют 8-разрядное АЛУ и накристалльное ОЗУ той же разрядности [www.microchip.com/PICMicro]. Команды этих микроконтроллеров размещаются в ПЗУ (также накристалльном), в котором каждое слово имеет 12 бит и содержит одну команду. Аналогично БЭСМ-6, команда микроконтроллера состоит из адресного поля (которое может содержать как адрес, так и константное значение длиной 1 байт) и кода операции. Под код операции остается всего 4 бита, поэтому команд, имеющих полное адресное поле, может быть не более 16. Адресное пространство микроконтроллера составляет 8 бит, т. е. всего 256 слов кода и 256 байт данных. Однако с помощью ухищрения, называемого банковской адресацией (подробнее об этом см. в разд. 2.6), старшие модели семейства PIC адресуют и по несколько килобайтов как кода, так и данных.

Впрочем, для многих целей и 256 команд вполне достаточно. Самый маленький в мире Web-сервер [www-ccs.cs.umass.edu] реализован именно на основе PIC. В 512 слов кода удалось упаковать реализацию полноценных подмножеств протоколов RS232 (использованная модель микроконтроллера не имеет аппаратно реализованного последовательного порта), PPP, TCP/IP и HTTP. Web-сервер состоит из двух микросхем — собственно сервера и кристалла флэш-памяти, в котором хранятся экспортируемые сервером HTML-документы и изображения. Сервер включается в последовательный порт компьютера и получает питание от этого порта.

Разработчик сервера рекламирует его как основу (или, во всяком случае, как демонстрацию технической возможности) создания Web-интерфейсов для разнообразного бытового оборудования.

В современных компьютерах единицей адресации является *байт* — ячейка памяти размером 8 бит. При этом и АЛУ, и шина данных могут иметь большую разрядность (хотя, конечно, эта разрядность всегда кратна байту — 8, 16, 32 или 64 бита). Разрядность АЛУ и шины данных может не совпадать: так, i8088 имел 16-разрядное АЛУ и 8-разрядную шину, MC68000 и i80386SX — 32-разрядное АЛУ и 16-разрядную шину. Что при этом понимается под "словом" — зависит от фантазии изготовителей процессора. Например, у вполне 32-разрядного VAX документация называет 16-битовую структуру словом, а 32-битовую — двойным или длинным словом, хотя обычно длиной слова считается все-таки разрядность АЛУ.

При этом различные процессоры по-разному подходят к адресации слов.

Во-первых, некоторые процессоры (PDP-11, SPARC) запрещают обращения к словам, адрес которых не кратен размеру слова, и генерируют при попытках такого обращения исключительную ситуацию ошибки шины. Другие процессоры, например, VAX и x86 такие обращения разрешают, но в документации есть честное предупреждение, что обращения к невыровненным словам будут минимум в два раза медленнее, чем к выровненным (рис. 2.5).

Во-вторых, есть разногласия в том, как следует адресовать (или, как следует интерпретировать при арифметических операциях) байты одного слова — по

какому адресу находятся старшие 8 битов, а по какому — младшие? У одних процессоров (IBM/390, MC68000, SPARC) старший байт располагается по меньшему адресу (*big endian*), у других (VAX, x86) — по большему (*little endian*). Англоязычные термины происходят от названий партий в книге "Путешествия Гулливера" — в русских переводах их обычно называют остроконечниками и тупоконечниками. Применение этих названий для описания представления данных в вычислительных системах обычно приписывают Дэнни Коэну [IEN 137].

4-байтовая структура данных, выровненная на границу 32-разрядного слова.
Все 4 байта могут быть считаны за одно обращение к памяти

0xDE	0xAD	0xBE	0xEF
0x02	0x12	0x85	0x06

Та же структура, но невыровненная.
Для считывания требуется два обращения к памяти

00	00	DE	AD
BE	EF	02	12

Рис. 2.5. Выровненные и невыровненные обращения к словам

Некоторые современные процессоры, например, PowerPC, MIPS, SPARC V9 могут работать и с тем, и с другим порядком байтов, причем иногда даже в пределах одной программы. Встречаются и процессоры со смешанным порядком байтов, когда, например, из двух байтов полуслова по меньшему адресу расположен младший, а из двух полуслов по меньшему адресу расположено старшее (рис. 2.6). Таким образом хранились 32-разрядные целые числа и числа с плавающей точкой в 16-разрядных компьютерах PDP-11, поэтому такой порядок байт даже называют PDP-endian.

Способы хранения числа 0xDEADBEEF

Адрес	Big endian	Little endian	Mixed endian
0	0xDE	0xEF	0xAD
1	0xAD	0xBE	0xDE
2	0xBE	0xAD	0xEF
3	0xEF	0xDE	0xBE

Рис. 2.6. Порядок байтов в слове

Возможные различия в порядке байтов следует принимать во внимание при разработке форматов для передачи данных по сети или сохранения в файле.

Кроме того, некоторые ошибки программирования (например, неявное преобразование указателя на байт в указатель на 16- или 32-битовое целое число) относительно безобидны на машинах, у которых младший байт первый, и приводят к немедленным катастрофическим результатам на компьютерах, использующих меньший адрес для старшего байта.

2.6. Режимы адресации

Once upon the time, when men were men, women were women, and pointers were int...

Однажды в далёкие времена, когда мужчины были мужчинами, женщины были женщинами, а указатели были целыми [числами]...

Операнды команд могут быть как регистрами, так и ячейками памяти. Некоторые архитектуры, например, PDP-11 и VAX, допускают произвольное сочетание регистров и ячеек памяти в одной команде. В частности, допустимы команды пересылки из памяти в память и арифметические операции, оба (у двухадресной PDP-11) или все три (у трехадресного VAX) операнда которых расположены в памяти. В других архитектурах, например, в x86 и MC680x0, только один operand команды может размещаться в памяти, а второй всегда обязан быть регистром (впрочем, оба эти процессора имеют и отдельные команды память-память, например, инструкции групповой пересылки данных). У RISC-процессоров арифметические операции разрешены только над регистрами, а для обращений к памяти выделены специальные команды LD (Load, загрузить) и ST (Store, сохранить).

В зависимости от подхода, применяемого в конкретной системе команд, архитектуры подразделяются на *память-память*, *регистр-память* и *регистр-регистр*. Архитектура регистр-регистр привлекательна тем, что позволяет сделать длину команды фиксированной (адресное поле могут иметь только команды LD/ST, а также команды вызова и перехода) и за счет этого упростить работу дешифратора и логики предварительной выборки команд. При небольшой длине адреса (как у старых компьютеров и современных микроконтроллеров) этим преимуществом обладают и архитектуры регистр-память.

Напротив, процессоры с большим адресным пространством и архитектурами память-память и регистр-память вынуждены иметь команды переменной длины. У процессоров VAX длина команды меняется от 1 (безадресная команда) до 61 байта (шестиадресная команда, все операнды которой используют самый сложный из допустимых режимов адресации).

По мере роста адресного пространства адресные поля команд, обращающихся к памяти, занимают все большую и большую долю кода, что является до-

полнительным стимулом к замене, где это возможно, обращений к памяти на обращения к регистрам. Благодаря этому же, код активно использующих регистры RISC-процессоров, несмотря на гораздо большую длину кодов команд (если у x86 наиболее широко используемые операции кодируются двумя байтами, то у типичного RISC все команды имеют длину 4 байта), ненамного превосходит по объему эквивалентный код для CISC-процессоров. По мере перехода к 64-разрядным адресам, выигрыш в объеме кода может стать преимуществом RISC-архитектур.

На основе сказанного ранее, у читателя могло сложиться впечатление, что единственным способом указания адреса операнда в памяти является помещение этого адреса в адресное поле команды. В действительности это не так, или, точнее, не всегда так — в зависимости от режима адресации, адрес операнда может вычисляться различными, иногда довольно сложными способами, с учетом значений одного или нескольких регистров, и как с использованием адресного поля, так и без него.

Большинство современных процессоров поддерживает многочисленные режимы адресации. Как и при работе с регистрами, это может реализоваться двумя путями: ортогональным, когда режим адресации кодируется битовым полем в коде команды, и неортогональным, когда различные режимы адресации соответствуют разным командам.

Поскольку различные режимы адресации могут как использовать адресное поле, так и не использовать его, чтобы реализовать ортогональную систему с командами фиксированной длины, нужно проявить незаурядную фантазию.

Режимы адресации VAX

У процессоров VAX операнды команд кодируются одним байтом. Старшие 4 бита операнда указывают режим адресации, младшие — номер регистра. Если режим предполагает использование адресных полей, эти поля следуют за операндом. При некоторых режимах возможно использование нескольких адресных полей, и длина кода одного операнда, таким образом, может доходить до 10 байт (рис. 2.7).

0	7	15	
Код команды	режим	регистр	

0	7	15	23	
Код команды	режим	регистр	режим	регистр

0	7	15	23	32	
Код команды	режим	регистр	режим	регистр	регистр

Рис. 2.7. Форматы одно-, двух- и трехадресной команд процессора VAX

Режимы адресации SPARC

У процессоров SPARC адресацию осуществляют лишь четыре группы команд: LD (загрузка слова из памяти в регистр), ST (сохранение значения регистра в памяти), JMPL (переход по указанному адресу и сохранение текущего адреса в регистре) и команды условного перехода. Все остальные команды манипулируют регистрами и константами. Команда длиной 32 бита имеет три битовых поля: два задают 5-разрядные номера регистров, третье — либо регистр, либо 13-разрядное целое число (см. рис. 2.4). Команды LD, SP и JMPL имеют такой же формат и позволяют использовать в качестве адреса либо сумму двух регистров, либо регистра и 13-разрядного значения, интерпретируемого как знаковое число в двоично-дополнительной кодировке. Это перекрывает далеко не все перечисленные далее режимы адресации. Многие распространенные режимы адресации на SPARC приходится реализовать с помощью нескольких команд.

Даже классические полностью ортогональные архитектуры — PDP-11, VAX, MC680x0 — имеют, по крайней мере, одно отклонение от полной ортогональности: режим адресации коротким смещением относительно счетчика команд (см. разд. 2.5.7), используемый в этих архитектурах в командах условного перехода и недоступный в других командах.

2.6.1. Вырожденные режимы адресации

К этой группе относятся режимы, в которых доступ к операнду не содержит адресации как таковой.

Первым из таких режимов является операнд-регистр. Режим этот концептуально крайне прост и в дополнительных комментариях не нуждается.

Второй режим — операнд-константа. В документациях по многим процессорам этот режим называют *литеральной* (literal) и *немедленной* (immediate) адресацией.

Казалось бы, трудно придумать более простой и жизненно необходимый режим. Однако полноценно реализовать такие операнды можно только двумя способами:

- используя команды переменной длины;
- используя команды, которые длиннее слова.

Так, в уже упоминавшемся микроконтроллере РСС разрядность слова данных составляет 8 бит, а длина слова программной памяти — и, соответственно, длина команды, — 12 бит. Благодаря этому в команде можно разместить полноценный 8-битный литерал.

Литеральная адресация в системе команд SPARC

Разработчики процессоров, которых не устраивает ни одно из названных условий, вынуждены проявлять фантазию. Так, у RISC-процессоров SPARC и команда, и слово имеют одинаковую длину — 32 бита. Адресное поле такой дли-

ны в команде невозможно — не остается места для кода операции. Выход, предложенный разработчиками архитектуры SPARC, при первом знакомстве производит странное впечатление, но, как говорят в таких случаях, "не критикуйте то, что работает".

Трехадресные команды SPARC могут использовать в качестве operandов три регистра или два регистра и беззнаковую константу длиной 13 бит. Если константа, которую мы хотим использовать в операции, умещается в 13 бит, мы можем просто использовать эту возможность. На случай, если значение туда не помещается, предоставляется команда `sethi const22, reg`, которая имеет 22-разрядное поле и устанавливает старшие биты указанного регистра равными этому полю, а младшие биты — равными нулю.

Таким образом, если мы хотим поместить в регистр 32-разрядную константу `value`, мы должны делать это с помощью двух команд: `sethi %hi(value), reg; or %g0, %lo(value), reg;` (в соответствии с [docs.sun.com 806-3774-10], именно так реализована ассемблерная псевдокоманда `set value, reg`).

С точки зрения занимаемой памяти, это ничуть не хуже, чем команда `set value, reg`, которая тоже должна была бы занимать 64 бита. Зато такое решение позволяет соблюсти принцип: одна команда — одно слово, который облегчает работу логике опережающей выборки команд.

Впрочем, для 64-разрядного SPARC v9 столь элегантного решения найдено не было. Способ формирования произвольного 64-битового значения требует дополнительного регистра и целой программы (пример 2.3). В зависимости от значения константы этот код может подвергаться оптимизации. Легче всего, конечно, дело обстоит, если требуемое значение помещается в 13 бит.

Пример 2.3. Формирование 64-разрядного значения на SPARC v9,
цит. по [docs.sun.com 806-3774-10]

```
! reg — промежуточный регистр, rd — целевой.  
sethi %hi(value), reg  
or reg, %ulo(value), reg  
sllx reg,32,reg ! сдвиг на 32 бита  
sethi %hi(value), rd  
or rd, reg, rd  
or rd, %lo(value), rd
```

"Короткие литералы" разного рода нередко используются и в других процессорах, особенно имеющих большую разрядность. Действительно, большая часть реально используемых констант имеет небольшие значения, и выделение под каждую такую константу 32- или, тем более, 64-разрядного значения привело бы к ненужному увеличению кода.

Короткие литералы VAX

У процессоров семейства VAX есть режим адресации, позволяющий использовать битовое поле, которое в других режимах интерпретируется как номер ре-

гистра, в качестве 4-битового литерала. Вместе с двумя битами режима адресации этим способом можно задать 6-разрядный литерал, знаковый или беззнаковый в зависимости от контекста [Прохоров 1990].

Короткие литералы MC680x0

У процессоров семейства MC680x0 литерал может иметь длину 1 или 2 байта. Кроме того, предоставляются команды ADDQ и SUBQ, которые позволяют добавить к указанному операнду или вычесть из него целое число в диапазоне от 1 до 8.

2.6.2. Абсолютная адресация

При *абсолютной адресации* адресное поле команды непосредственно содержит номер целевой ячейки памяти. Таким способом производятся обращения к объектам с постоянными адресами: статическим и внешним переменным, точкам входа подпрограмм.

Как и в случае с обсуждавшейся в предыдущем подразделе литеральной адресацией, процессор, который использует такую адресацию, либо должен иметь команды переменной длины, либо его команда должна быть длиннее адреса. Второе условие выполняется не только у процессоров гарвардской архитектуры, но и у многих старых компьютеров с небольшим адресным пространством.

Абсолютная адресация в системе команд SPARC

У процессоров SPARC реализация абсолютной адресации похожа на реализацию адресации литеральной: под адресацией занимается регистр, командой `sethi %hi(addr)`, `reg` в нем загружается старшая часть адреса, а затем происходит собственно адресация. Для формирования 64-разрядного адреса необходимо занимать два регистра и выполнять ту же программу, что и в примере 2.3.

Обращение к переменной в памяти происходит так, как показано в примере 2.4. В модулях, содержащих много обращений к переменным, рекомендуется выделить для этой цели регистр и использовать смещения относительно него — как, кстати, и сделано в приведенном примере. Но это уже совсем не абсолютная адресация.

Пример 2.4. Обращение к переменной на процессоре SPARC

```
sethi %hi(var), %g1      ! помещаем старшие биты адреса в %g1
ld [%g1+%lo(var)], %l1  ! загружаем значение в %l1
inc %l1                  ! производим операцию
st %l1, [%g1+%lo(var)] ! сохраняем результат.
```

2.6.3. Косвенно-регистровый режим

В этом режиме, как и в регистровом, адресное поле не используется. Значение регистра интерпретируется как адрес операнда. Данный режим применяется для разыменования указателей или для обращения к памяти по предварительно вычисленному адресу.

Некоторые процессоры, такие как PDP-11, VAX, MC680x0, имеют любопытные варианты данного режима — адресацию с постинкрементом и предекрементом. Постинкремент означает, что после собственно адресации значение регистра увеличивается на величину адресуемого объекта. Предекремент, соответственно, означает, что регистр уменьшается на ту же величину перед адресацией.

Эти режимы могут использоваться для разнообразных целей, например, для реализации операций над текстовыми строками или поэлементного сканирования массивов. Но одно из основных назначений — реализация стека.

Стек

Стек, или *магазин*, — это структура данных, над которой мы можем осуществлять две операции: проталкивание (*push*) значения и выталкивание (*pop*). Значения выталкиваются из стека в порядке, обратном тому, в котором проталкивались: *LIFO* (Last In, First Out, первый вошел, последний вышел). Стековые структуры находят широкое применение при синтаксическом разборе арифметических выражений и алголоподобных языков программирования [Кормен/Лейзерсон/Ривест 2000].

Самая простая реализация стека — это массив и индекс последнего находящегося в стеке элемента (рис. 2.8). Этот индекс называется *указателем стека* (Stack Pointer, SP). Стек может расти как вверх, так и вниз (рис. 2.9). Важным недостатком такого стека является его ограниченный размер; при этом либо необходимо проверять указатель стека при каждом проталкивании или выталкивании значения, либо приходится мириться с опасностью выхода за границы массива.

Широко применяются также реализации стеков в виде односвязных списков. Такие реализации позволяют создавать стеки практически неограниченного размера, но сами операции проталкивания и выталкивания оказываются намного дороже, чем в стеках, реализованных на массиве.

При реализации стека скалярных значений удобно использовать непрерывную область памяти в качестве массива, регистр SP в качестве указателя и режимы адресации с постинкрементом и предекрементом при реализации команд проталкивания и выталкивания.

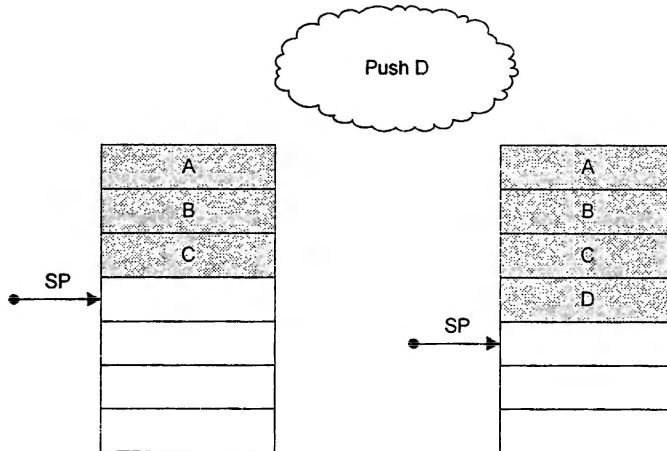


Рис. 2.8. Стек на основе массива

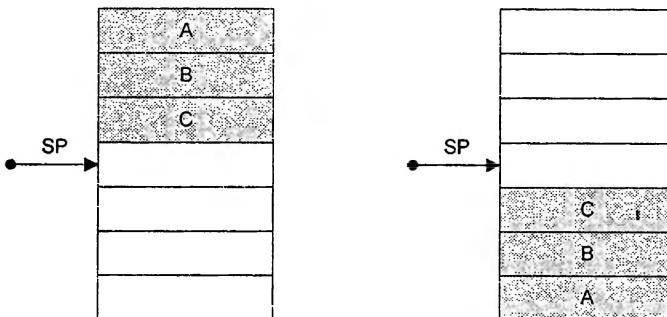


Рис. 2.9. Стеки, растущие вверх и вниз

Действительно, команда

`MOVE x, -(SP)`

приведет к тому, что указатель стека уменьшится на размер x , и мы положим x в освободившуюся память. Напротив, команда

`MOVE (SP)+, y`

приведет к получению значения и продвижению указателя стека в обратном направлении. Поэтому первая команда имеет также мнемоническое обозначение

`PUSH x`

а вторая

`POP y`

Если мы поместим несколько значений в стек командой `PUSH`, команда `POP` вытолкнет их из стека в обратном порядке. Стек можно использовать для хранения промежуточных данных (пример 2.5) и при реализации арифметических выражений — например, команда

```
ADD (SP)+, (SP)
```

в точности воспроизводит описанную ранее семантику безадресной команды `ADD` стековой архитектуры. Впрочем, безадресной команде `ADD` операнды не нужны, а в данном случае они просто не используются, но никуда не исчезают. Команда получается длиннее: у типичной стековой архитектуры команда сложения занимает 1 байт, у PDP-11 ее имитация занимает 2 байта, а у VAX — целых три. Поэтому, если мы хотим использовать стековую технику генерации кода, лучше применять предназначенный для этого процессор.

Пример 2.5. Использование стека для хранения промежуточных значений

```
void swap(int &a, int &b) {  
    int t;  
    t=a;  
    a=b;  
    b=t;  
}
```

; Для простоты мы не рассматриваем механизм передачи параметров

; и считаем, что они передаются в регистрах A и B

.GLOBL swap

swap:

```
    PUSH (A)  
    MOVE (B), (A)  
    POP (B)  
    RET.
```

Одно из основных назначений стека в регистровых архитектурах — это сохранение адреса возврата подпрограмм. Кроме того, если принятое в системе соглашение о вызовах подпрограмм предполагает, что вызываемая процедура должна сохранить все или некоторые регистры, которые использует сама, стек обычно применяют и для этого.

Неортогональные процессоры, такие как x86, часто предоставляют специальные команды `PUSH` и `POP`, работающие с выделенным регистром `SP` (Stack Pointer), который не может быть использован для других целей.

2.6.4. Косвенно-регистровый режим со смещением

Адрес операнда образуется путем сложения регистра и адресного поля команды. Этот режим наиболее богат возможностями и, в зависимости от стиля использования, имеет много других названий, например, *базовая адресация* или *индексная адресация*. Адресное поле необязательно содержит полноценный адрес и может быть укороченным.

Команды `ld/st` процессора SPARC, используемые в примере 2.4, реализуют косвенно-регистровую адресацию с 13-разрядным смещением.

Возможные варианты использования этого режима адресации многочисленны. Например, если смещение представляет собой абсолютный адрес начала массива, а в регистре хранится индекс, данный режим может использоваться для индексации массива. В этом случае смещение должно представлять собой полноценный адрес.

В другом случае, в регистре может храниться указатель на структуру данных, а смещение может означать смещение конкретного поля относительно начала структуры. Еще один вариант — регистр хранит указатель на стековый кадр или блок параметров процедуры, а смещение — адрес локальной переменной в этом кадре или определенного параметра. В подобных случаях можно использовать (и обычно используется) укороченное смещение.

Стековый кадр

Стековый кадр (stack frame) или, что то же самое, *запись активации*, является стандартным способом выделения памяти под локальные переменные в алголоподобных процедурных языках (C, C++, Pascal) и других языках, допускающих рекурсивные вызовы.

Семантика рекурсивного вызова в алголоподобных языках требует, чтобы каждая из рекурсивно вызванных процедур имела собственную копию локальных переменных. В SPARC это достигается сдвигом регистрового окна по регистровому файлу (рис. 2.10), но большинство других процессоров такой роскоши лишены и вынуждены выделять память под локальные переменные в стеке, размещаемом в ОЗУ.

Для этого вызванная процедура уменьшает (если стек растет вниз) указатель стека на количество байтов, достаточное, чтобы разместить переменные. Адресация этих переменных у некоторых процессоров (например, у PDP-11) происходит относительно указателя стека, а у большинства — например, у MC680x0 и VAX, с большим количеством регистров или у x86, указатель

стека которого нельзя использовать для адресации со смещением — для этой цели выделяется отдельный регистр (рис. 2.11, примеры 2.6 и 2.7).

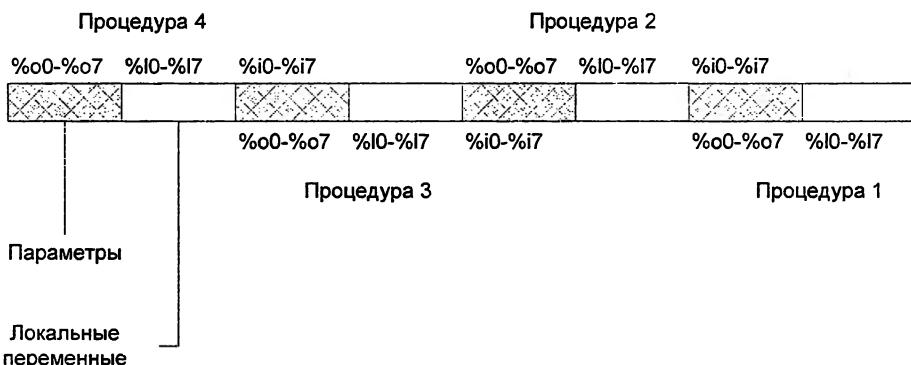


Рис. 2.10. Регистровый стек процессора SPARC

Пример 2.6. Формирование, использование и уничтожение стекового кадра.
Код на языке С и результат его обработки GNU C 2.7.2.1
(комментарии автора)

```
#include <stdio.h>
#include <strings.h>

/* Фрагмент примитивной реализации сервера SMTP (RFC0822) */
int parse_line(FILE * socket)
{
/* Согласно RFC0822, команда имеет длину не более 4 байт,
   а вся строка — не более 255 байт */
char cmd[5], args[255];
fscanf(socket, "%s %s\n", cmd, args);

if (strcmp(cmd, "HELO") == 0) {
fprintf(socket, "200 Hello, %s, glad to meet you\n", args);
return 200;
}
/* etc */
fprintf(socket, "500 Unknown command %s\n", cmd);
return 500;
}
```

```

Bar()
{
    char Str1[25], Str2[25];
    Foo(Str1, Str2);
    return 0;
}

Foo(char *Src, char *Dst)
{
    char Tmp[25];
    int I, J;

    // ...
    Return 0;
}

```

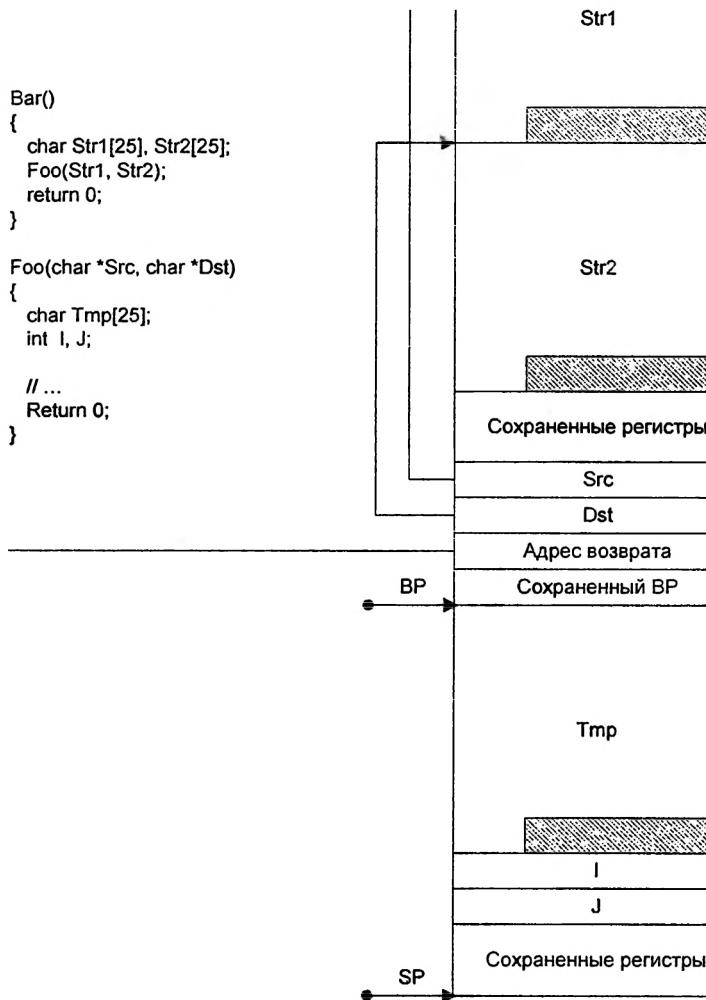


Рис. 2.11. Стековый кадр

Пример 2.7. Ассемблерный код, порождаемый GNU C при компиляции программы 2.6

```

.file "sample.c"
.text
LC0:
.ascii "%s %s\12\0"
LC1:
.ascii "HELO\0"
.balign 4
LC2:
.ascii "200 Hello, %s, glad to meet you\12\0"
LC3:
.ascii "500 Unknown command %s\12\0"
.globl _parse_line
_parse_line:
    pushl %ebp          ; Сохраняем указатель калра
                           ; вызвавшей нас подпрограммы.
    movl %esp, %ebp ; Формируем указатель нашего кадра
    subl $296, %esp ; И сам кадр.
; x86 имеет для этой цели специальную команду enter, но она может
; формировать кадры размером не более 255 байт. В данном случае кадр
; имеет больший размер, и его необходимо формировать вручную.
; Конец пролога функции
    leal -280(%ebp), %eax ; Помещаем в стек указатель на args
    pushl %eax
    leal -24(%ebp), %eax ; ... на cmd
    pushl %eax
    pushl $LC0           ; на строковую константу
    pushl 8(%ebp)        ; на строковую константу
; Наши собственные параметры тоже адресуются относительно кадра.
    call _fscanf          ; Вызов (параметры в стеке)
    addl $16, %esp        ; очищаем стек
; В языке С допустимо переменное количество параметров, поэтому
; вычищать их из стека должна вызывающая процедура.
; Вызываемая процедура просто не знает, сколько параметров было
; на самом деле.
; В Pascal и C++ стек очищает вызываемая процедура.
    subl $8, %esp
    pushl $LC1            ; Передаем параметры strcmp

```

```

leal    -24(%ebp), %eax
pushl  %eax
call   _strcmp
addl   $16, %esp
testl  %eax, %eax          ; Проверка условия оператора if
jne    L14
subl   $4, %esp
leal    -280(%ebp), %eax
pushl  %eax
pushl  $LC2
pushl  8(%ebp)
call   _fprintf
addl   $16, %esp
movl   $200, -284(%ebp)
jmp    L13

L14:
subl   $4, %esp
leal    -24(%ebp), %eax
pushl  %eax
pushl  $LC3
pushl  8(%ebp)
call   _fprintf
addl   $16, %esp
movl   $500, -284(%ebp)

L13:
movl   -284(%ebp), %eax
; Команда leave совершает действия, обратные прологу функции:
; Она эквивалентна командам: move %ebp, %esp; pop %ebp.
; Размер кадра явным образом не указывается, поэтому ограничений
; на этот размер в данном случае нет.
leave
ret

```

Примечание

Обратите внимание, что программа из примера 2.6 содержит серьезнейшую ошибку. В комментариях сказано, что команда обязана иметь длину не более 4 байтов, а вся строка вместе с аргументами не более 255. Если программа-клиент на другом конце сокета (сетевого соединения) соответствует RFC 0822 [RFC 0822], так оно и будет. Но если программа требованиям этого документа не соответствует, нас ждет беда: нам могут предложить более длинную команду и/или более длинную строку. Последствия, к которым это может привести, будут более подробно разбираться в главе 12.

Стековые кадры в системе команд SPARC

Микропроцессоры SPARC также не могут обойтись без стекового кадра. Во-первых, не всегда локальные переменные процедуры помещаются в восьми 32-разрядных локальных регистрах. Именно такая процедура приведена в примере 2.6. Во-вторых, нередки ситуации, когда в качестве параметров надо передать по значению структуры, для которых 6 регистров-параметров тоже не хватит. В-третьих, глубина регистрового файла ограничена и при работе рекурсивных или просто глубоко вложенных процедур может исчерпаться. В-четвертых, в многозадачной системе регистровый файл может одновременно использоваться несколькими задачами. Все эти проблемы решаются с помощью создания стекового кадра [[www.sparc.com v9](http://www.sparc.com)].

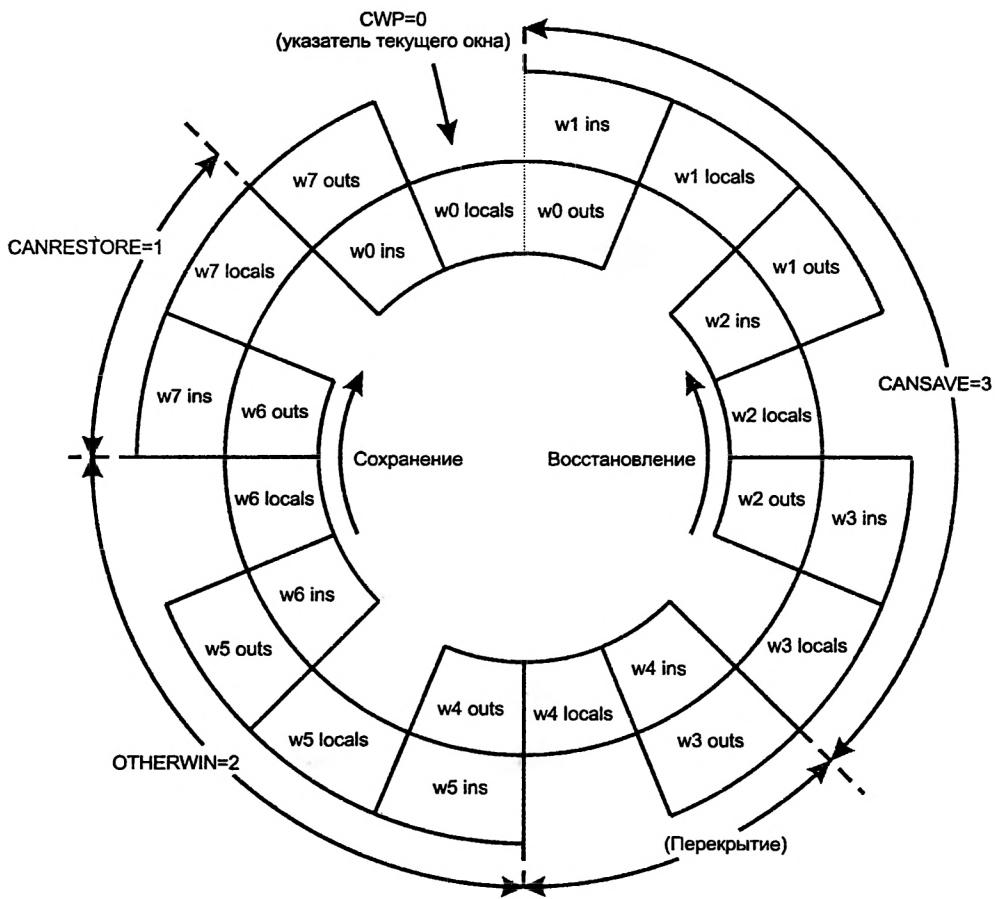
Для этой цели используются регистры %sp (об) и %fp (i6). Команда save %sp, -96 %sp делает следующее: она складывает первые два операнда, сдвигает стековый кадр и помещает результат сложения в третий operand. Благодаря такому порядку исполнения отдельных операций, старый %sp становится %fp, а результат сложения помещается уже в новый %sp.

Самую важную роль стековые кадры играют при обработке переполнений регистрационного файла. Регистровый файл SPARC представляет собой кольцевой буфер, доступность отдельных участков которого описывается привилегированными регистрами CANSAVE и CANRESTORE. Окна, находящиеся между значениями этих двух регистров, доступны текущей программе (рис. 2.12). На рисунке показано состояние регистрационного файла, в котором текущий процесс может восстановить один стековый кадр (CANRESTORE=1) и сохранить три (CANSAVE=3). Регистр OTHERWIN указывает количество регистрационных окон, занятых другим процессом. Регистровое окно w4 на рисунке (обозначенное как *перекрытие*) занято лишь частично. Текущее окно, частично занятое окно и участки регистрационного файла, описанные перечисленными регистрами, в сумме должны составлять весь регистрационный файл, так чтобы соблюдалось отношение CANSAVE + CANRESTORE + OTHERWIN = NWINDOWS – 2, где NWINDOWS — количество окон (на рисунке регистрационный файл имеет 8 окон, т. е. 128 регистров).

Когда же программа пытается сдвинуть свое окно за описанные границы (в ситуации, изображенной на рис. 2.11, это может произойти после вызовов четырех вложенных процедур или после возврата из двух процедур — текущей и соответствующей окну w7), генерируются исключительные состояния заполнения окна (window fill) и сброса окна (window spill). При этом вызывается системная процедура, которая освобождает окна из интервала OTHERWIN, сбрасывая их содержимое в стековые кадры соответствующих процедур и при заполнении восстанавливает содержимое принадлежащего нам окна из соответствующего кадра.

В многозадачной системе заполнение и сброс окна могут произойти в любой момент, поэтому пользовательская программа всегда должна иметь по стековому кадру на каждое из используемых ею регистрационных окон, а указатель на соответствующий кадр должен всегда лежать в %sp каждого окна. При этом очень важно, чтобы создание стекового кадра и сдвиг регистрационного окна производились одной командой. Если делать это двумя командами, то у нас может возникнуть регистрационное окно без соответствующего стекового кадра в памяти. В этот момент может произойти прерывание, и исполнение будет передано другому процессу, который может вызвать переполнение окна. Обработчик ис-

ключения по переполнению окна не будет знать, что наше окно еще ни разу не использовалось, и сохранит регистры в области памяти, в которую указывает %sp, т. е., скорее всего, не там, где мы хотели бы.



$$\text{CANSAVE} + \text{CANRESTORE} + \text{OTHERWIN} = \text{NWINDOWS} - 2$$

Рис. 2.12. Регистровый файл SPARC в виде кольцевого буфера.
Регистры CANSAVE и CANRESTORE (цит. по [www.sparc.com v9])

2.6.5. Базово-индексный режим

В базово-индексном режиме адрес операнда образуется сложением двух или, реже, большего количества регистров и, возможно, еще и адресного смещения. Такой режим может использоваться для адресации массивов: один ре-

гистр при этом содержит базовый адрес массива, второй — индекс, откуда и название. Иногда значение индексного регистра умножается на размер операнда, иногда — нет.

На первый взгляд, ортогональные архитектуры должны испытывать определенные сложности с кодированием такой адресации: для этого нужно два регистровых поля, а большинство остальных режимов довольствуются одним регистром. Однако многие ортогональные архитектуры, например, VAX, MC680x0, SPARC, реализуют этот режим, пусть иногда и с ограничениями.

Индексный режим адресации VAX

У VAX за операндом, указывающим индексный режим адресации и индексный регистр, следует еще один байт, кодирующий режим адресации и регистр, используемые для вычисления базового адреса (рис. 2.13). Идея разрешить многократное указание индексного регистра в одном операнде, к сожалению, не реализована.

Индексный режим адресации в системе команд SPARC

SPARC позволяет использовать для вычисления адреса в командах LD, ST и JMPL как сумму двух регистров, так и сумму регистра и 13-разрядного смещения. Таким образом, эти команды реализуют либо косвенно-регистровый режим (если используется смещение и оно равно 0), либо косвенно-регистровый режим со смещением, либо базово-индексный режим без смещения. Это, конечно, беднее, чем у CISC-процессоров, но жить с таким набором вполне можно.

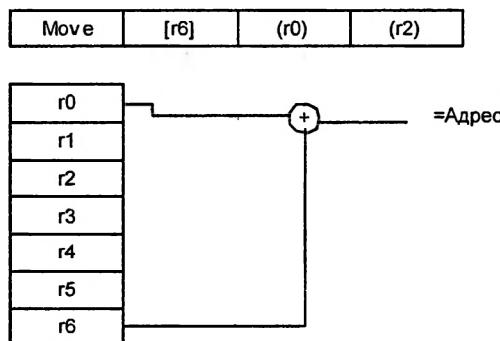


Рис. 2.13. Индексный режим адресации VAX

Немало современных процессоров, впрочем, предлагают программисту реализовать такой режим с помощью нескольких команд и с использованием промежуточного регистра, в который следует поместить сумму базового и индексного регистров.

2.6.6. Сложные режимы адресации

Среди промышленно выпускавшихся процессоров самым богатым набором экзотических режимов адресации обладает VAX. Кроме всех перечисленных ранее, предлагаются следующие:

- **косвенный с постинкрементом:** регистр содержит адрес слова, которое является адресом операнда. После адресации регистр увеличивается на 4;
- **косвенный со смещением** (не путать с косвенно-регистровым со смещением!): регистр со смещением адресует слово памяти, которое содержит адрес операнда, — удобен для разыменования указателя без его загрузки в регистр.

2.6.7. Адресация с использованием счетчика команд

Любой процессор предоставляет как минимум один способ такой адресации: адресация самих команд при их последовательной выборке осуществляется с помощью счетчика команд с постинкрементом. У процессоров с командами переменной длины величина постинкремента зависит от кода команды.

Некоторые процессоры позволяют использовать счетчик команд наравне со всеми остальными регистрами общего назначения. Запись в этот регистр приводит к передаче управления по адресу, который соответствует записанному значению. Чтение из этого регистра позволяет узнать адрес текущей команды, что само по себе не очень полезно и часто может быть сделано другими способами. Однако использование других режимов адресации со счетчиком команд порой позволяет делать неожиданные, но весьма полезные трюки.

Литеральная и абсолютная адресация в PDP-11 и VAX

VAX и PDP-11 не реализуют в чистом виде ни литерального, ни абсолютного режимов адресации. Вместо этого литерал или адрес помещается в программную память непосредственно за операндом и используются, соответственно, косвенно-регистровый с постинкрементом и косвенный с постинкрементом режимы со счетчиком команд в качестве регистра. При исполнении команды счетчик команд указывает на слово, следующее за текущим отрабатываемым операндом (рис. 2.14). Использование постинкремента приводит к тому, что счетчик увеличивается на размер, соответственно, литерала или адреса, и таким образом процессор находит следующий операнд. Этот остроумный прием можно рассматривать как своеобразный способ реализовать команды переменной длины.

Использование счетчика команд в косвенно-регистровом режиме со смещением позволяет адресовать код и данные относительно адреса текущей команды. Такой режим адресации называется *относительным (relative)*. Про-

граммный модуль, в котором используется только такая адресация, позиционно независим: его можно перемещать по памяти, и он даже не заметит факта перемещения, если только не получит управление в процессе самого перемещения или не будет специально проверять адреса на совпадение. Впрочем, почти того же эффекта можно достичь базовой адресацией.

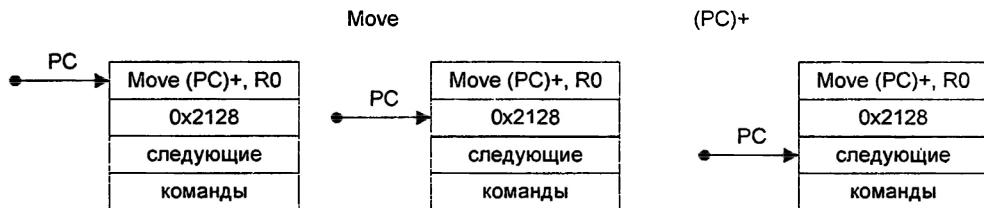


Рис. 2.14. Реализация литеральной адресации через постинкрементную адресацию счетчиком команд

Многие современные процессоры такого режима адресации для данных не предоставляют, зато почти все делают нечто подобное для адресации кода. А именно, во всех современных процессорах команды условного перехода используют именно такую адресацию: эти команды имеют короткое адресное поле, которое интерпретируется как знаковое смещение относительно текущей команды.

Дело в том, что основное применение условного перехода — это реализация условных операторов и циклов, в которых переход осуществляется в пределах одной процедуры, а зачастую всего на несколько команд вперед или назад. Снабжать такие команды длинным адресным полем было бы расточительно и привело бы к ненужному раздуванию кода.

Условные переходы на большие расстояния в коде встречаются относительно редко, и чаще всего их предлагают реализовать двумя командами (пример 2.8).

Пример 2.8. Реализация условного перехода с длинным смещением

```

        Beq    distant_label ; Перейти, если равно
; реализуется как
        Bneq $1                ; Перейти, если не равно
        Jmp    distant_label

; У команд безусловного перехода обычно используется длинное смещение
; или абсолютный адрес
$1:

```

Относительные переходы в системе команд SPARC

У большинства CISC-процессоров адресное смещение в командах условного перехода ограничено одним байтом. У SPARC такие команды используют адресное поле длиной 22 бита. С учетом того факта, что команды у SPARC всегда выровнены на границу слова (4 байта), такая адресация позволяет непосредственно указать до 4 Мбайт ($4 \times 2^{20} = 4\ 194\ 304$) команд или 16 Мбайт, т. е. целиком адресовать сегмент кода большинства реально используемых программ (рис. 2.15).

Команда вызова подпрограммы у SPARC также использует адресацию относительно счетчика команд, но адресное поле у нее 30-разрядное и интерпретируется как адрес слова, а не байта. При сложении смещения и счетчика команд возможные переполнения игнорируются, поэтому такой командой можно адресовать любое слово (т. е. любую команду) в 32-разрядном адресном пространстве. На первый взгляд, неясно даже, какая польза от того, что адресация производится относительно счетчика команд, а не абсолютно. Но в 64-разрядных процессорах SPARC v9 польза от этого большая — абсолютный 30-разрядный адрес позволял бы адресовать только первое гигаслово памяти, а относительное смещение адресует именно сегмент кода, в какой бы части 64-разрядного адресного пространства он ни находился. Программ, имеющих объем более одной гигакоманды или даже половины гигакоманды, пока что не написано, поэтому 30-разрядного смещения практически достаточно для адресации в пределах любой современной программы.

CALL

OP	30-битное смещение		
31 29			0

SETHI и команды условного перехода

OP	rd или условие	OP2	22-разрядная константа или смещение
31 29		24 21	0

Рис. 2.15. Формат команд условного перехода и вызова процессора SPARC

Процессоры, не предоставляющие программисту прямого доступа к счетчику команд, зачастую все-таки дают возможность записывать в него произвольные значения с помощью специальных команд вычислимого перехода и вычислимого вызова. Команды вычислимого вызова широко используются для реализации указателей на функции из таблиц виртуальных методов в объектно-ориентированных языках. Главное применение команд вычислимого перехода — реализация операторов `switch` языка C/C++ или `case` языка Pascal.

2.7. Банки памяти

Банки памяти используются, когда адресное пространство процессора мало, а приложение требует много памяти. Например, у многих микроконтролле-

ров адрес имеет длину всего 8 бит, однако 256 байт данных, и тем более 256 команд кода для большинства приложений недостаточно. Многие из ранних персональных компьютеров, основанных на 8-разрядных микропроцессорах i8085 и Z80 с 16-разрядным адресом, имели гораздо больше 64 Кбайт памяти. Например, популярные в годы моего детства компьютеры Yamaha MSX на основе процессора Z80 могли иметь до 2 Мбайт оперативной памяти.

Адресация дополнительной памяти в этой ситуации обеспечивается дополнительным адресным регистром, который может быть как конструктивным элементом процессора, так и внешним устройством. Данный регистр дает нам дополнительные биты адреса, которые и обеспечивают адресацию дополнительной памяти. Регистр этот называется *расширителем адреса* (address extension) или *селектором банка* (bank selector), а область памяти, которую можно адресовать, не изменяя селектор банка, — *банком памяти*. Значение регистра-селектора называют *номером банка*.

Банковая адресация в 16-разрядных микропроцессорах

Внимательный читатель, знакомый с системой команд Intel 8086, не может не отметить, что "сегментные" регистры этого процессора имеют мало общего с собственно сегментацией, описываемой в главе 5. Эти регистры более похожи на причудливый гибрид селектора банков и базового регистра. Как и описываемый далее PIC, i8086 имеет команды "ближних" (внутрибанковых) и " дальних" (межбанковых) переходов, вызовов и возвратов.

Относящийся к тому же поколению процессоров Zilog 800 имеет полноценные селекторы банков. Из всех изготовителей 16-разрядных микропроцессоров только инженеры фирмы Motorola осмелились расширить адрес до 24 бит (это потребовало увеличения разрядности регистров и предоставления команд 32-разрядного сложения), все остальные так или иначе экспериментировали с селекторами банков и вариациями на эту тему.

Работа с банками памяти данных обычно не представляет больших проблем, за исключением ситуаций, когда нам нужно скопировать из одного банка в другой структуру данных, которую невозможно разместить в регистрах процессора. Существенно более сложную задачу представляет собой передача управления между банками программной памяти.

В том случае, когда селектор банка программной памяти интегрирован в процессор, предоставляются специальные команды, позволяющие перезагрузить одновременно "младшую" (собственно регистр PC) и "старшую" (селектор банка) части счетчика команд.

Банки команд в PIC

У микроконтроллеров PIC арифметические операции производятся только над младшими 8 битами счетчика команд, поэтому относительные и вычислимые переходы допустимы только в пределах 256-командного банка. Однако полное — с учетом селектора банка — адресное пространство для команд дости-

гает 64 Кбайт, а у старших моделей и 16 Мбайт за счет использования двух регистров-расширителей. Переключение банка осуществляется специальными командами "длинного" — межбанкового — перехода.

Если банковая адресация реализована как внешнее устройство, проблема межбанковой передачи управления встает перед нами в полный рост. Поскольку мы не имеем команд межбанкового перехода, любой такой переход состоит минимум из двух команд: переключения банка и собственно перехода. Каждая из них нарушает порядок исполнения команд.

Рассмотрим ситуацию детальнее (рис. 2.16): из кода, находящегося в банке 1 по адресу 0x10af, мы хотим вызвать процедуру, находящуюся в банке 2 по адресу 0x2000. Если мы сначала выполним переключение банка, мы окажемся в банке 2 по адресу 0x10b0, не имея никакого представления о том, какой же код или данные размещены по этому адресу. С той же проблемой мы столкнемся, если сначала попытаемся сделать переход по адресу 0x1fff.

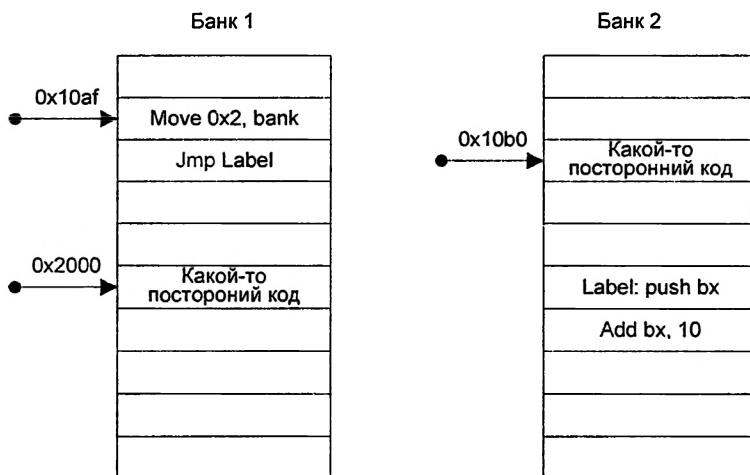


Рис. 2.16. Межбанковый переход

В качестве решения можно предложить размещение по адресу 0x1fff в банке 1 команды переключения на банк 2. Возможно, для этого придется переместить какой-то код или данные, но мы попадем по желаемому адресу. Впрочем, если мы постоянно осуществляем межбанковые переходы, этот подход потребует вставки команд переключения банка для каждой возможной точки входа во всех остальных банках. Ручное (да и автоматизированное) размещение этих команд — операция чрезвычайно трудоемкая, и возникает естественная идея: сконцентрировать все вставленные команды и соответствующие им точки входа в каком-то одном месте. Впрочем, даже эта идея не

дает нам ответа на вопрос, как же при такой архитектуре возвращать управление из процедур? Вставлять команду переключения еще и для каждой команды вызова?

Развитие этой идеи приводит нас к чему-то, похожему на менеджер оверлеев (см. разд. 3.6): программный модуль, который присутствует во всех банках по одному и тому же адресу (рис. 2.17). Если нам нужно вызвать известную процедуру в определенном банке, мы передаем ее адрес и номер банка этому модулю, и он осуществляет сохранение текущего банка, переключение и переход. Если процедура делает возврат, она возвращает управление тому же модулю, который, в свою очередь, восстанавливает исходный банк и возвращает управление в точку вызова.

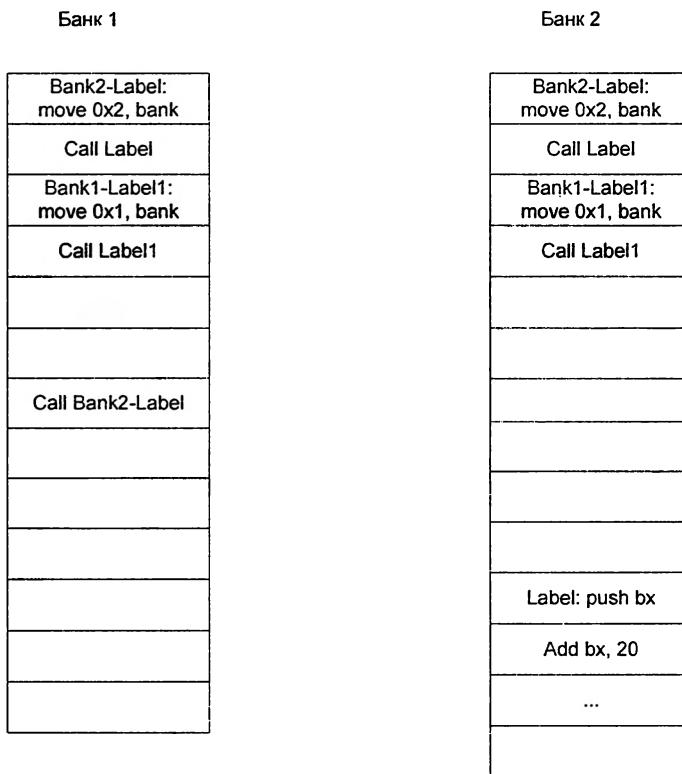


Рис. 2.17. Переключатель банков

Дальнейшее развитие этой идеи приводит к мысли, что самый простой способ разместить этот код во всех банках — усложнить схему работы селектора банков, например, всегда отображать первый килобайт адресного пространства на одни и те же физические адреса. Аппаратно это несложно. Селектор

банков мог бы анализировать старшие шесть битов адресной шины процессора. Если они не равны нулю, на старшие биты адресной шины памяти по-давалось бы содержимое регистра селектора банка, если же равны — нулевые биты. Примерно этим способом и расширяют память большинство микроКомпьютеров на основе 8-разрядных процессоров.

Поскольку мы вступили на путь анализа логического адреса, можно пойти и дальше: разбить адресное пространство процессора на несколько банков, каждый со своим селектором.

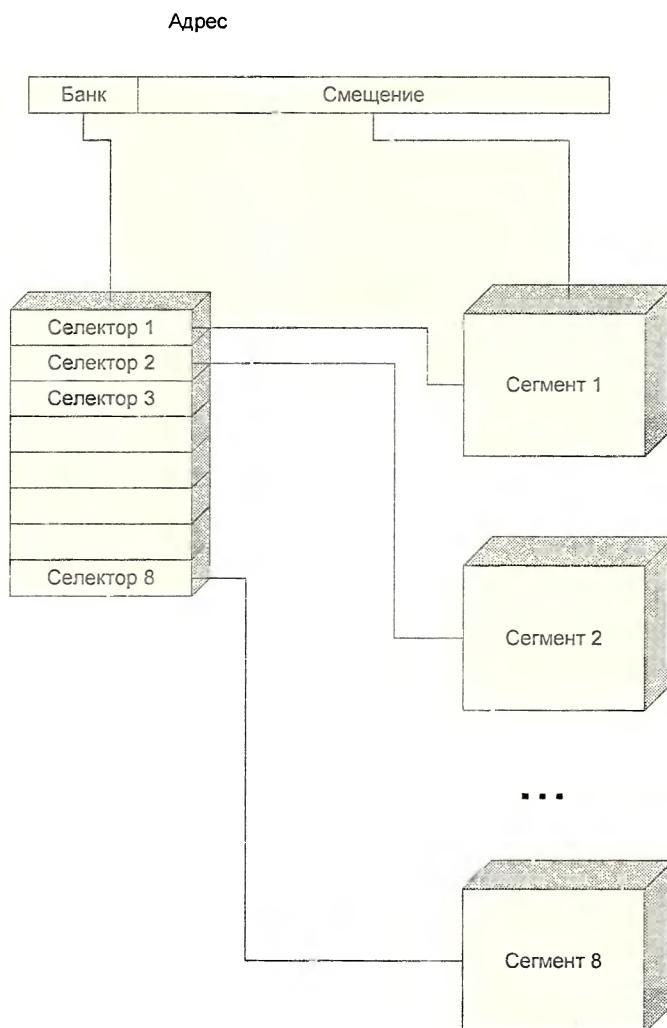


Рис. 2.18. Виртуальная память PDP-11/20

Адресное пространство PDP-11

Машины серии PDP-11 имеют 16-разрядный адрес, который позволяет адресовать 64 Кбайт. У старших моделей серии это пространство разбито на 8 сегментов по 8 Кбайт. Каждому из этих сегментов соответствует свой селектор банка (в данном случае следует уже говорить о *дескрипторе сегмента*) (рис. 2.18). Физическое адресное пространство, которое может быть охвачено дескрипторами сегментов, составляет 2 Мбайт, что намного больше адресов, доступныхциальному процессору. На первый взгляд, эта конструкция представляет собой усложненную реализацию банковой адресации, цель которой — только расширить физическое адресное пространство за пределы логического, но тот факт, что, кроме физического адреса, каждый сегмент имеет и другие атрибуты, в том числе права доступа, заставляет нас признать, что это уже совсем другая история, заслуживающая отдельной главы (см. главу 5).

2.8. CISC- и RISC-процессоры

Часто приходится сталкиваться с непониманием термина RISC, общепринятая расшифровка которого — Reduced Instruction Set Computer (компьютер с уменьшенной системой команд). Какой же, говорят, SPARC или PowerPC — RISC, если у него количество кодов команд не уступает или почти не уступает количеству команд в x86? Почему тогда x86 не RISC, ведь у него команд гораздо меньше и они гораздо проще, чем у VAX 11/780, считающегося классическим примером CISC-архитектуры. Да и технологии повышения производительности у современных x86- и RISC-процессоров используются примерно те же, что и у больших компьютеров 70-х годов: множественные АЛУ, виртуальные регистры, динамическая перепланировка команд.

В действительности, исходно аббревиатура RISC расшифровывалась несколько иначе, а именно как Rational Instruction Set Computer (RISC, компьютер с рациональной системой команд). RISC-процессоры, таким образом, противопоставлялись процессорам с необязательно сложной (CISC — Complex Instruction Set Computer, компьютер со сложной системой команд), но "иррациональной", исторически сложившейся архитектурой, в которой, в силу требований бинарной и ассемблерной совместимости с предыдущими поколениями, накоплено множество команд, специализированных регистров и концепций, в общем-то и не нужных, но... вдруг отменишь команду двоично-десятичной коррекции, а какое-то распространенное приложение "сломается"? А мы заявляли бинарную совместимость. Скандал!

Понятно, что быть "рациональными" в таком понимании могут лишь осваивающие новый рынок разработчики, которых не заботит та самая бинарная совместимость. Уже сейчас, например, фирма Sun одним из главных достоинств своих предложений на основе процессоров UltraSPARC числит полную бинарную совместимость с более ранними машинами семейства SPARC. Новые процессоры вынуждены поддерживать бинарную совместимость с

64-разрядными младшими родственниками и режим совместимости с 32-разрядными. Где уж тут заботиться о рациональности.

С другой стороны, рациональность тоже можно понимать по-разному. В конце 70-х и первой половине 80-х годов общепринятым пониманием "рациональности" считалась своеобразная (с высоты сегодняшнего дня) ориентация на языки высокого уровня. Относительно примитивные трансляторы тех времен кодировали многие операции, например, прологи и эпилоги процедур, доступ к элементу массива по индексу, вычислимые переходы (*switch* в C, *case* в Pascal) с помощью стандартных последовательностей команд. Поскольку все меньше и меньше кода писалось вручную и все больше и больше — генерировалось трансляторами, разработчики процессоров решили пойти создателям компиляторов навстречу.

Этот шаг навстречу выразился в стремлении заменить там, где это возможно, последовательности операций, часто встречающиеся в откомпилированном коде, одной командой. Среди коммерчески успешных архитектур апофеозом этого подхода следует считать семейство миникомпьютеров VAX фирмы DEC, в котором одной командой реализованы не только пролог функции и копирование строки символов, но и, скажем, операции удаления и вставки элемента в односвязный список. Приведенная в *разд. 2.2* в качестве примера шестиадресной команды операция *INDEX* — реальная команда этого процессора. Отдельные проявления данной тенденции без труда прослеживаются и в системах команд MC68000 и 8086. Аббревиатура CISC позднее использовалась именно для характеристики процессоров этого поколения.

С другой стороны, неумение трансляторов обсуждаемого поколения эффективно размещать переменные и промежуточные значения по регистрам считалось доводом в пользу того, что от регистров следует отказываться и заменять их регистровыми стеками или кэш-памятью. (Впрочем, и у VAX, и у MC68000 с регистрами общего назначения было все в порядке, по 16 штук.)

Ко второй половине 80-х годов развитие технологий трансляции позволило заменить генерацию стандартных последовательностей команд более интеллектуальным и подходящим к конкретному случаю кодом. Разработанные технологии оптимизации выражений и поиска инвариантов цикла позволяли, в частности, избавляться от лишних проверок. Например, если цикл исполняется фиксированное число раз, а счетчик цикла используется в качестве индекса массива, не надо вставлять проверку границ индекса в тело цикла — достаточно убедиться, что ограничитель счетчика не превосходит размера массива. Более того, если счетчик используется только в качестве индекса, можно вообще избавиться и от него, и от индексации, а вместо этого применять указательную арифметику. Наконец, если верхняя граница счетчика фиксирована и невелика, цикл можно развернуть (пример 2.9).

Пример 2.9. Эквивалентные преобразования программы.

```
/* Пример возможной стратегии оптимизации.
 * Код, вставляемый компилятором для проверки границ индекса,
 * выделен с помощью нестандартного выравнивания. */

int array[100];

int bubblesort(int size) {
    int count;
    do {
        count=0;
        for(i=1; i<100; i++) {
            if (i<0 || i>100) raise(IndexOverflow);
            if (i-1<0 || i-1>100) raise(IndexOverflow);
            if (array[i-1]<array[i])
            {
                if (i<0 || i>100) raise(IndexOverflow);
                int t=array[i];
                if (i<0 || i>100) raise(IndexOverflow);
                if (i-1<0 || i-1>100) raise(IndexOverflow);
                array[i]=array[i-1];
                if (i-1<0 || i-1>100) raise(IndexOverflow);
                array[i-1]=t;
                count++;
            }
        }
    } while (count != 0);
}

// Оптимизированный внутренний цикл может выглядеть так:
register int *ptr=array;
register int *limit=ptr;
register int t=*ptr++;

if (size<100) limit+=size; else limit+=100;

while (ptr<limit) {
    if (t<*ptr) {
        ptr[-1]=*ptr;
```

```
*ptr++=t;
count++;
} else t=*ptr++;
}
if (size>100) raise(IndexOverflow);
```

По мере распространения в мини- и микрокомпьютерах кэшей команд и данных, а также конвейерного исполнения команд объединение множества действий в один код операции стало менее выгодным с точки зрения производительности.

Это привело к радикальному изменению взглядов на то, каким должен быть идеальный процессор, ориентированный на исполнение откомпилированного кода. Во-первых, компилятору не нужна ни бинарная, ни даже ассемблерная совместимость с чем бы то ни было (отсюда "рациональность"). Во-вторых, ему требуется много взаимозаменяемых регистров — минимум тридцать два, а на самом деле чем больше, тем лучше. В-третьих, сложные комбинированные команды усложняют аппаратуру процессора, а толку от них все равно нет или мало.

Коммерческий успех процессоров, построенных в соответствии с этими взглядами (SPARC, MIPS, PA-RISC), привел к тому, что аббревиатура RISC стала употребляться к месту и не к месту — например, уже упоминавшийся Transputer (имевший регистровый стек и реализованный на уровне системы команд планировщик, т. е. являющийся живым воплощением описанного ранее CISC-подхода) в документации называли RISC-процессором, фирма Intel хвасталась, что ее новый процессор Pentium построен на RISC-ядре (что под этим только ни подразумевалось?) и т. д.

2.9. Языки ассемблера

Непосредственно на машинном языке в наше время не программирует практически никто. Первый уровень, позволяющий абстрагироваться от схемы кодирования команд, — это уже упоминавшийся язык *ассемблера*. В языке ассемблера каждой команде машинного языка соответствует мнемоническое обозначение. Все приведенные ранее примеры написаны именно на языке ассемблера, да и в тексте использовались не бинарные коды команд, а их мнемоники.

Встречаются ассемблеры, которые предоставляют мнемонические обозначения для часто используемых групп команд. Большинство таких языков позволяет пользователю вводить свои собственные мнемонические обозначения — так называемые *макроопределения* или *макросы* (macros), в том числе и параметризованные (пример 2.10).

Отличие макроопределений от процедур языков высокого уровня в том, что процедура компилируется один раз, и затем ссылки на нее реализуются в виде команд вызова. Макроопределение же реализуется путем подстановки тела макроопределения (с заменой параметров) на место ссылки на него и компиляцией полученного текста. Компиляция ассемблерного текста, таким образом, осуществляется в два или более проходов — на первом осуществляется раскрытие макроопределений, на втором — собственно компиляция, которая, в свою очередь, может состоять из многих проходов, их смысл мы поймем далее. Часть ассемблера, реализующая первый проход, называется макропроцессором.

Пример 2.10. Использование макроопределений

```
; Фрагмент драйвера LCD для микроконтроллера PIC
; (c) 1996, Дмитрий Иртегов.

; Таблица знакогенератора: 5 байт/символ.
; W содержит код символа. Пока символов может быть
; только 50, иначе возникнет переполнение.
; Scanline содержит номер байта (не строки!)

; Сначала определим макрос, а то устанем таблицу сочинять.
; Необходимо упаковать 7 скан-строк по 5 бит в 5 байт.

CharDef macro scan1, scan2, scan3, scan4, scan5, scan6, scan7
; Следующий символ
    RetLW (scan7 & 0x1c) >> 2
    RetLW ((scan5 & 0x10) >> 4) + ((scan6 & 0x1f) << 1) + ((scan7 & 0x3) << 6)
    RetLW ((scan4 & 0x1e) >> 1) + ((scan5 & 0xf) << 4)
    RetLW ((scan2 & 0x18) >> 3) + ((scan3 & 0x1f) << 2) + ((scan4 & 0x1) << 7)
    RetLW (scan1 & 0x1f) + ((scan2 & 0x7) << 5)
endm

FetchOneScanline
IFNDEF NoDisplay
    ClrF PCLATH
    AddWF PCL, 1
    NOP ;
else
    RetLW 0
endif
```

```
; А вот идет собственно таблица:  
Nolist  
; 0  
CharDef B'01110',B'10001',B'10001',B'10001',B'10001',B'10001',B'01110'  
; 1  
CharDef B'00100',B'01100',B'00100',B'00100',B'00100',B'00100',B'01110'  
; 2  
CharDef B'01110',B'10001',B'00001',B'00010',B'00100',B'01000',B'11111'  
; 3  
CharDef B'01110',B'10001',B'00001',B'00110',B'00001',B'10001',B'01110'  
; 4  
CharDef B'00010',B'00110',B'01010',B'10010',B'11111',B'00010',B'00010'  
; 5  
CharDef B'11111',B'10000',B'11110',B'00001',B'00001',B'10001',B'01110'  
; 6  
CharDef B'01110',B'10001',B'10000',B'11110',B'10001',B'10001',B'01110'  
; 7  
CharDef B'11111',B'00001',B'00010',B'00100',B'01000',B'01000',B'01000'  
; 8  
CharDef B'01110',B'10001',B'10001',B'01110',B'10001',B'10001',B'01110'  
; 9  
CharDef B'01110',B'10001',B'10001',B'01111',B'00001',B'10001',B'01110'  
; A  
Constant CharacterA = 0xa  
CharDef B'00100',B'01010',B'10001',B'10001',B'11111',B'10001',B'10001'  
Ifndef NO_ALPHABET  
; B  
Constant CharacterW = 0xb  
CharDef B'11110',B'10001',B'10001',B'11110',B'10001',B'10001',B'11110'  
else  
; Р — для асона  
CharDef B'11110',B'10001',B'10001',B'10001',B'11110',B'10000',B'10000'  
endif  
; C  
CharDef B'01110',B'10001',B'10000',B'10000',B'10000',B'10001',B'01110'  
; D  
CharDef B'11110',B'10001',B'10001',B'10001',B'10001',B'10001',B'11110'  
; E  
CharDef B'11111',B'10001',B'10000',B'11110',B'10000',B'10001',B'11111'  
; F
```

```
CharDef B'11111',B'10001',B'10000',B'11110',B'10000',B'10000',B'10000'  
; пробел  
Constant SPACE_CHARACTER = 0x10  
CharDef B'00000',B'00000',B'00000',B'00000',B'00000',B'00000',B'00000'  
;  
Constant DASH_CHARACTER = 0x11  
CharDef B'00000',B'00000',B'00000',B'11111',B'00000',B'00000',B'00000'  
List
```

Макропроцессор, кроме раскрытия макросов, обычно предоставляет также директивы условной компиляции — в зависимости от условий, те или иные участки кода могут передаваться компилятору или нет. Условия, конечно же, должны быть известны уже на этапе компиляции. Например, в зависимости от типа целевого процессора одна и та же конструкция может реализоваться как в одну команду, так и эмулирующей программой. В зависимости от используемой операционной системы могут применяться разные системные вызовы (это чаще случается при программировании на языках высокого уровня), или в зависимости от значений параметров макроопределения, макрос может порождать совсем разный код.

Макросредства есть не только в ассемблерах, но и во многих языках высокого уровня (ЯВУ). Наиболее известен препроцессор языка С. В действительности, многие средства, предоставляемые языками, претендующими на большую, чем у С, "высокоуровневость" (что бы под этим ни подразумевалось), также реализуются по принципу макрообработки, т. е. с помощью текстовых подстановок и компиляции результата: шаблоны (template) С++, параметризованные типы Ada и т. д.

Умелое использование макропроцессора облегчает чтение кода и увеличивает возможности его повторного использования в различных ситуациях. Злоупотребление же макросредствами (как, впрочем, и многими другими мощными и выразительными языковыми конструкциями) или просто бестолковое их применение может приводить к совершенно непонятному коду и трудно диагностируемым ошибкам, поэтому многие теоретики программирования выступали за полный отказ от использования макропроцессоров.

Современные методы оптимизации в языках высокого уровня — проверка константных условий, разворачивание циклов, inline-функции — часто стирают различия между макрообработкой и собственно компиляцией.

Кроме избавления программиста от необходимости запоминать коды команд, ассемблер выполняет еще одну, пожалуй, даже более важную функцию: он позволяет снабжать символическими именами (*метками*, *label* или символами, *symbol*) команды или ячейки памяти, предназначенные для данных. Зна-

чение этой возможности для практического программирования трудно переоценить.

Рассмотрим простой пример из жизни: мы написали программу, которая содержит команду перехода (бывают и программы, которые ни одной команды перехода не содержат, но это вырожденный случай). Затем в процессе тестирования этой программы или уточнения спецификаций мы поняли, что между командой перехода и точкой, в которую переход совершается, необходимо вставить еще два десятка команд. Для вставки необходимо пересчитать адрес перехода. На практике, вставка даже одной только инструкции часто затрагивает и приводит к необходимости пересчитывать адреса множества команд перехода, поэтому возможность автоматизировать этот процесс крайне важна.

Важное применение меток — организация ссылок между модулями в программах, собираемых из нескольких раздельно компилируемых файлов. Изменение объема кода или данных в любом из модулей приводит к необходимости пересчета адресов во всех остальных модулях. В современных программах, собираемых из сотен отдельных файлов и содержащих тысячи индивидуально адресуемых объектов, выполнять такой пересчет вручную невозможно. Способы автоматического решения этой задачи обсуждаются в разд. 3.7.

Первый ассемблер

Первый ассемблер был создан в 1949 году для компьютера EDSAC, разработанного Морисом Уилкисом из Кембриджского университета в Англии. Это был первый работающий компьютер, использовавший ОЗУ в качестве программной памяти (модифицированный фон Нейманом ENIAC и EDVAC использовали в качестве программной памяти ПЗУ, программируемое с помощью рубильников). Уилкис также обычно приписывают изобретение названия "ассемблер".

Программная память EDSAC была реализована на ртутных линиях задержки и состояла из 512 18-разрядных двоичных слов. Позднее память была расширена до 1024 слов. Команды использовали одиннадцатиразрядный адрес, так что теоретически память могла быть расширена до 2048 слов. В качестве кодов команд использовались буквы латинского алфавита, кодируемые нестандартной пятибитной кодировкой.

Кроме программной памяти, EDSAC имел ПЗУ на основе механических шаговых искателей, в котором хранились так называемые "начальные приказы" (initial orders). При включении машины содержимое ПЗУ копировалось в начало ОЗУ и запускалось как программа.

В окончательном виде "начальные приказы" представляли собой простой ассемблер, который находил свободную память, загружал с перфоленты исходный код с символьными метками, пересчитывал метки в адреса и запускал программу. Поскольку команды и так обозначались однобуквенными мнемоническими кодами, фазы трансляции мнемонических кодов в машинные не требовалось. Благодаря этому, ассемблер был очень маленьким — вместе с загрузчиком они умещались в 41 слове программного ОЗУ [Wilkes/Wheeler 1951].

Фаза сопоставления символов с реальными адресами присутствует и при компиляции языков высокого уровня — компилятор генерирует символы не только для переменных, процедур и меток, которые могут быть использованы в операторе `goto`, но и для реализации "структурных" условных операторов и циклов. Нередко в описании компилятора эту фазу так и называют — ассемблирование.

Многие компиляторы — как старые, так и современные, например, популярный компилятор GNU C, даже не выполняют фазу ассемблирования самостоятельно, а вместо этого генерируют текст на языке ассемблера и вызывают внешний ассемблер. Средства межпроцессного взаимодействия современных ОС позволяют передавать этот промежуточный текст, не создавая промежуточного файла, поэтому для конечного пользователя эта деталь реализации часто оказывается незаметной.

Компиляторы, имеющие встроенный ассемблер, такие как Microsoft Visual C/C++ или Watcom, часто могут генерировать ассемблерное представление порождаемого кода. Это бывает полезно при отладке или написании подпрограмм на ассемблере, которые должны взаимодействовать с откомпилированным кодом.

2.9.1. Многопроходное ассемблирование

При ассемблировании с использованием меток возникает специфическая проблема: команды могут ссылаться на метки, определенные как до, так и после них по тексту программы. Следовательно, операндом команды может оказаться метка, которая еще не определена. Адрес, соответствующий этой метке, еще неизвестен, поэтому мы должны будем, так или иначе, вернуться к ссылающейся на нее команде и записать адрес. Эта же проблема возникает и при компиляции ЯВУ: предварительное определение переменных и процедур указывает тип переменной, количество и типы параметров процедуры, но не их размещение в памяти, а именно оно нас и интересует при генерации кода.

Две техники решения этой проблемы называются *одно- и двухпроходным ассемблированием* [Баррон 1974].

При двухпроходном ассемблировании на первом проходе мы определяем адреса всех описанных в программе символов и сохраняем их в промежуточной таблице. На втором проходе мы осуществляем собственно ассемблирование — генерацию кода и расстановку адресов. Если адресное поле имеет переменную длину, определение адреса метки может привести к изменению длины ссылающегося на нее кода, поэтому на таких архитектурах оказывается целесообразным трех- и более проходное ассемблирование. При однопро-

ходном ассемблировании мы запоминаем все точки, из которых происходят ссылки вперед, и, определив адрес символа, возвращаемся к этим точкам и записываем в них адрес. При однопроходном ассемблировании целесообразно хранить код, в котором еще не все метки расставлены, в оперативной памяти, поэтому в старых компьютерах двухпроходные ассемблеры были широко распространены. Впрочем, современные многопроходные ассемблеры также хранят промежуточные представления программы в памяти, поэтому количество проходов в конкретной реализации ассемблера представляет разве что теоретический интерес.

2.10. Отладчики и деассемблеры

При отладке программы часто оказывается желательно посмотреть значения тех или иных переменных в различные периоды исполнения. Нередко эту задачу удается решить простой расстановкой диагностических операторов печати в соответствующих участках кода, но этот метод часто оказывается трудоемким и непродуктивным. К счастью, существуют программные инструменты, позволяющие существенно облегчить эту задачу, в том числе остановить исполняющуюся программу в определенной точке, посмотреть значения произвольных ячеек памяти, продолжить исполнение и т. д. Эти инструменты называются *отладчиками* (debugger).

Удобства и пользу отладчиков для процесса разработки программного обеспечения трудно переоценить. Особенно полезен оказывается просмотр значений переменных в момент аварийного завершения программы. Для этого многие ОС при перехвате фатальных ошибок специально создают образ памяти задачи или, что то же самое, *посмертный дамп* (postmortem dump — слово dump по-английски обозначает "сбрасывание", "выкидывание") или, как это называлось в старой русскоязычной литературе, посмертную выдачу программы. Современные отладчики обычно могут анализировать не только образ памяти активной задачи, но и посмертные дампы. Так, в системах семейства Unix при аварийном завершении процесса создается файл core (название файла происходит от слова core — сердечник). Это воспоминание о тех временах, когда компьютеры были большими и использовали ОЗУ на ферритовых сердечниках). Этот файл содержит не только образ памяти и регистров процесса на момент завершения, но и образ системных структур данных, связанных с этим процессом, так что, анализируя файл в отладчике, можно воссоздать всю среду, в которой программа исполнялась в момент аварии. Если устранить причину, по которой возникла ошибка, некоторые отладчики позволяют даже продолжить исполнение задачи с момента аварии.

Многие ОС могут создавать посмертный дамп не только для аварийно завершающихся пользовательских программ, но и для самих себя — обнаружив

фатальную ошибку в каком-то из модулей ядра, ОС сбрасывает содержимое ОЗУ в специальную область на диске либо просто в файл подкачки. Стандартные отладчики, предназначенные для пользовательских программ, обычно не могут работать с такими дампами, для этого нужны специальные средства, называемые *отладчиками ядра* (kernel debugger).

2.10.1. Листинги и отладочная информация

Понятно, что программиста обычно не очень-то интересует значение абстрактной ячейки памяти — его интересует все-таки значение переменной. Определение, в какой ячейке памяти компилятор разместил конкретную переменную, представляет собой нетривиальную задачу. К счастью, компилятор в момент компиляции владеет необходимой информацией; многие компиляторы и практически все ассемблеры могут генерировать специальные текстовые файлы, называемые *листингами* (listing), в которых приводится исходный код программы и порожденный на его основании машинный код. Для всех символьических объектов, в том числе и переменных, в листинге указываются их адреса (пример 2.11). Листинги занимают довольно много места, поэтому современные компиляторы по умолчанию не генерируют их, а делают это только по специальному запросу.

Пример 2.11. Листинг программы из примера 2.6

```
GAS LISTING sample.s
1                      .file  "sample.c"
2                      .text
3          LC0:         .ascii "%s %s\12\0"
4 0000 25732025      4          730A00
5          LC1:         .ascii "HELO\0"
6 0007 48454C4F      6          00
7                      .balign 4
8          LC2:         .ascii "200 Hello, %s, glad to meet
9 000c 32303020      you\12\0"
you\12\0"
9          48656C6C
9          6F2C2025
9          732C2067
9          6C616420
10         LC3:         .ascii "500 Unknown command %s\12\0"
```

```
11      556E6B6E
11      6F776E20
11      636F6D6D
11      616E6420
12          .globl _parse_line
13          _parse_line:
14 0045 55          pushl  %ebp
15 0046 89E5          movl   %esp, %ebp
16 0048 81EC2801          subl   $296, %esp
16      0000
17 004e 8D85E8FE          leal    -280(%ebp), %eax
17      FFFF
18 0054 50          pushl  %eax
19 0055 8D45E8          leal    -24(%ebp), %eax
20 0058 50          pushl  %eax
21 0059 68000000          pushl  $LC0
21      00
22 005e FF7508          pushl  8(%ebp)
23 0061 E89AFFFF          call   _fscanf
23      FF
24 0066 83C410          addl   $16, %esp
25 0069 83EC08          subl   $8, %esp
26 006c 68070000          pushl  $LC1
26      00
27 0071 8D45E8          leal    -24(%ebp), %eax
28 0074 50          pushl  %eax
29 0075 E886FFFF          call   _stricmp
29      FF
30 007a 83C410          addl   $16, %esp
31 007d 85C0          testl  %eax, %eax
32 007f 7526          jne    L14
33 0081 83EC04          subl   $4, %esp
34 0084 8D85E8FE          leal    -280(%ebp), %eax
34      FFFF
35 008a 50          pushl  %eax
36 008b 680C0000          pushl  $LC2
36      00
37 0090 FF7508          pushl  8(%ebp)
38 0093 E868FFFF          call   _fprintf
38      FF
```

```

39 0098 83C410          addl    $16, %esp
40 009b C785E4FE        movl    $200, -284(%ebp)
40      FFFFC800
40      0000
41 00a5 EB21           jmp     L13
42      L14:
43 00a7 83EC04          subl    $4, %esp
44 00aa 8D45E8          leal    -24(%ebp), %eax
45 00ad 50              pushl    %eax
46 00ae 682D0000        pushl    $LC3
46      00
47 00b3 FF7508          pushl    8(%ebp)
48 00b6 E845FFFF        call    _fprintf
48      FF
49 00bb 83C410          addl    $16, %esp
50 00be C785E4FE        movl    $500, -284(%ebp)
50      FFFFF401
50      0000
51      L13:
52 00c8 8B85E4FE        movl    -284(%ebp), %eax
52      FFFF
53 00ce C9              leave
54 00cf C3              ret
DEFINED SYMBOLS
sample.s:13      .text:00000045 _parse_line

```

UNDEFINED SYMBOLS

`_fscanf`
`_strcmp`
`_fprintf`

В те времена, когда отладчиков еще не было, листинги были одним из самых важных вспомогательных средств при отладке и анализе посмертных дампов. Современные отладчики используют более компактный, чем листинг, формат вспомогательной информации, которая так и называется — *отладочная информация* (*debug information*), хотя многие старые отладчики умеют использовать и файлы листинга.

Современная отладочная информация не содержит исходных текстов программы, а лишь ссылку на файл (или файлы), в которых можно найти эти тексты. Отладочная информация генерируется компилятором и присоединя-

ется к загрузочному модулю. Во время отладки или при анализе дампа памяти отладчик должен иметь доступ и к загрузочному модулю, и к исходным текстам программы. Многие отладчики также поддерживают хранение отладочной информации в отдельных файлах, обычно имеющих расширение .SYM.

Для каждой строки исходного кода, на основе которой был порожден код (очевидно, что пустые строки, комментарии, ветви условной компиляции, условие которой оказалось ложным, и некоторые конструкции, например, закрывающие скобки блоков кода в алголоподобных языках, обычно кода не порождают), указывается диапазон адресов порожденного кода.

При оптимизирующей компиляции между строками исходного и командами машинного кода часто оказывается невозможно установить соответствие. Потому некоторые компиляторы отказываются порождать отладочную информацию при включенной оптимизации, а те компиляторы, которые все-таки удается заставить делать это, нередко генерируют информацию, приводящую к весьма странному поведению отладчика — некоторые операторы исходного текста вовсе не имеют соответствующего кода, порядок их исполнения не соответствует порядку, в котором они должны были бы исполняться в соответствии с наивной интерпретацией кода и т. д.

Для каждой строки, на основе которой были порождены переменные или другие символические объекты (например, метки или процедуры), указываются имена этих объектов и их адреса.

Легко понять, что в отладочной информации нуждаются не только программы, написанные на ЯВУ, но и программы на ассемблере, ведь в ассемблерном исходнике, который содержит макроопределения и директивы условной компиляции, восстановить строку исходного текста на основе адреса команды не всегда бывает легко, а имена символьических объектов во время исполнения просто неизвестны.

Таким образом, отладочная информация представляет собой две ассоциативные таблицы. Ключом поиска в первой таблице является адрес в образе памяти, а значением — строка исходного текста, на основе которой была порождена соответствующая переменная или участок кода. Во второй таблице ключом также является адрес, а значением — символьическое значение (имя метки, переменной или точки входа процедуры), соответствующее этому адресу. Поиск по обеим таблицам может осуществляться в двух направлениях, как по адресу, так и по значению — отладчик может установить адрес первой команды, соответствующей указанной строке и, напротив, может быстро найти строку, в которой в данный момент находится управление. Точнее, отладочная информация в том виде, в каком она складывается в загрузочный модуль, обычно представляет собой просто набор записей указанных типов,

но отладчик при работе вынужден так или иначе преобразовать ее в разреженные ассоциативные массивы.

Современные компиляторы порождают также дополнительную информацию, сообщающую о типах переменных, именах и типах параметров функций, содержимом структур и иерархии наследования классов C++ и других объектно-ориентированных языков. В примере 2.12 легко найти все упомянутые типы информации (отладочные записи вводятся директивами .stabs, .stabn и .stabd); можно заметить, что информация о типах сохраняется в языково-независимом формате — так, для встроенных типов языка С указывается их "природа" (целое это число или с плавающей точкой) и диапазон допустимых значений. Можно также заметить, что в отладочную информацию попали и данные о структурах, определенных в файлах заголовков, включенных директивой #include., например, о структуре FILE, определенной в stdio.h.

Примечание

Ассемблерный код примера 2.12 несколько отличается от кода в примерах 2.6 и 2.11, потому что он был порожден другой версией компилятора.

Пример 2.12. Ассемблерный код программы примера 2.6 с порожденной отладочной информацией

```

.stabs "double:t13=r1;8;0;",128,0,0,0
.stabs "long double:t14=r1;12;0;",128,0,0,0
.stabs "complex int:t15=s8real:1,0,32;imag:1,32,32;:",128,0,0,0
.stabs "complex float:t16=r16;8;0;",128,0,0,0
.stabs "complex double:t17=r17;16;0;",128,0,0,0
.stabs "complex long double:t18=r18;24;0;",128,0,0,0
.stabs "__builtin_va_list:t19=*2",128,0,0,0
.stabs "_Bool:t20=@s8;-16;",128,0,0,0
.stabs "size_t:t21=5",128,0,12,0
.stabs
"_FILE:T22=s48_ptr:23=*2,0,32;_buffer:23,32,32;_rcount:1,64,32;_
wcount:1,96,32;_handle:1,128,32;_flags:1,160,32;_buf_size:1,192,32;
_tmpidx:1,224,32;_pid:1,256,32;_char_buf:2,288,8;_ungetc_count:11,296,8;
_mbstate:8,304,16;_flush:24=*25=f1,320,32;_more:26=*27=xs_file2:,352,32;_
",128,0,0,0
.stabs "FILE:t28=22",128,0,49,0
.stabs
"fpot_t:t29=30=s12_pos:3,0,32;_reserved1:3,32,32;_mbstate:8,64,16;
_reserved2:8,80,16;_",128,0,109,0
.stabs "va_list:t31=23",128,0,114,0

LC0:
.ascii "%s %s\12\0"
LC1:
.ascii "HELO\0"
.balign 4
LC2:
.ascii "200 Hello, %s, glad to meet you\12\0"
LC3:
.ascii "500 Unknown command %s\12\0"
.globl _parse_line
_parse_line:
.stabd 68,0,6
LBB2:
.pushl %ebp
.movl %esp, %ebp
.subl $296, %esp
.stabd 68,0,10
.leal -280(%ebp), %eax
.pushl %eax
.leal -24(%ebp), %eax
.pushl %eax
.pushl $LC0

```

```
pushl  8(%ebp)
call   _fscanf
addl   $16, %esp
.stabd 68,0,12
subl   $8, %esp
pushl  $LC1
leal   -24(%ebp), %eax
pushl  %eax
call   _stricmp
addl   $16, %esp
testl  %eax, %eax
jne    L14
.stabd 68,0,13
subl   $4, %esp
leal   -280(%ebp), %eax
pushl  %eax
pushl  $LC2
pushl  8(%ebp)
call   _fprintf
addl   $16, %esp
.stabd 68,0,14
movl   $200, -284(%ebp)
jmp    L13
```

L14:

```
.stabd 68,0,17
subl   $4, %esp
leal   -24(%ebp), %eax
pushl  %eax
pushl  $LC3
pushl  8(%ebp)
call   _fprintf
addl   $16, %esp
.stabd 68,0,18
movl   $500, -284(%ebp)
```

LBE2:

```
.stabd 68,0,19
```

L13:

```
movl   -284(%ebp), %eax
leave
ret
.stabs "parse_line:F1",36,0,6,_parse_line
```

```
.stabs "socket:p32=*28",160,0,5,8
.stabs
"cmd:33=ar34=r34;000000000000;00377777777777;0;4;2",128,0,9,-24
.stabs "args:35=ar34;0;254;2",128,0,9,-280
.stabn 192,0,0,LBB2
.stabn 224,0,0,LBE2
```

2.10.2. Отладка

Два основных понятия, связанных с отладкой, — это *пошаговое исполнение* и *точки останова* (breakpoint).

Режим пошагового исполнения поддерживается многими современными процессорами и, как правило, включается определенным битом в слове состояния процессора. Сам режим состоит в том, что после исполнения каждой команды процессор останавливает исполнение текущей программы и вызывает определенную процедуру, обработчик отладочного исключения. При вызове этой процедуры слово состояния процессора сохраняется в стеке, а бит пошагового исполнения сбрасывается. Разумеется, обработчик исключения является модулем отладчика и позволяет решить, не следует ли передать управление основной нити отладчика и вступить в диалог с пользователем, показать ему строку исходного кода, исполнение которой сейчас происходит, позволить посмотреть и изменить значения переменных, прекратить исполнение программы и т. д.

Пошаговое исполнение таким способом приводит к очень большим накладным расходам — исключения генерируются при исполнении каждой команды, в то время как пользователя обычно интересуют шаги, соответствующие операторам ЯВУ, на котором была написана программа, поэтому даже когда пользователь указал режим пошаговой отладки, обычно все-таки используются точки останова.

Точка останова также требует определенной поддержки со стороны процессора: процессор должен предоставлять специальную команду, которая прекращает исполнение программы и опять-таки передает управление обработчику отладочного исключения. Отладчик записывает эту команду по адресу, по которому необходимо поставить точку останова. Код, который замещается этой командой, разумеется, сохраняется. Когда программа приходит в точку останова, генерируется исключение и отладчик заменяет команду останова на ранее находившийся там код.

Современные отладчики позволяют ставить точки останова на точки входа функций, на произвольные строки исходного текста (разумеется, при условии, что эта строка соответствует хоть какому-то коду) и на произвольные адреса кода.

Большинство современных отладчиков позволяет также создавать *условные точки останова*, которые срабатывают лишь при соблюдении некоторого условия (например, при равенстве определенной переменной определенному значению). С точки зрения отладчика, такая точка останова все-таки срабатывает при каждой передаче управления на нее, но пользователь оповещается о сработке лишь при выполнении условия.

На процессорах с переменной длиной команды команда отладочного останова должна иметь длину, равную длине самой короткой команды. Так, у процессора x86 в качестве отладочного останова используется команда `int 03h;`. В отличие от остальных команд программной генерации прерывания, которые кодируются двумя байтами, эта команда кодируется одним байтом.

При пошаговом исполнении, если шаги соответствуют строкам ЯВУ, отладчик просто ставит точку останова по адресу, который соответствует первой команде кода, порожденного на основании строки, в которую должно прийти управление. Если текущая строка представляет собой условный оператор или точку входа или выхода цикла, следующих потенциальных строк может оказаться несколько. Чтобы разобраться в возможных при этом коллизиях, отладчик иногда все-таки вынужден прибегать к пошаговому исполнению. Когда управление доходит до этой строки, точка останова (или, точнее, все точки останова, соответствовавшие возможным ветвям исполнения) снимается и ставится новая точка останова в начале следующей строки. Таким образом, даже при пошаговом исполнении обычно все-таки используются точки останова, хотя и скрытым от пользователя образом.

При исполнении под управлением ДОС, отладчик может непосредственно просматривать и модифицировать код и данные отлаживаемой программы. Впрочем, при работе под ДОС некоторые крайне полезные отладочные функции, например, анализ посмертного дампа памяти, оказываются недоступны — потому что под ДОС невозможен сам этот посмертный дамп, ведь при большинстве ошибок ДОС гибнет (оказывается неработоспособна) вместе с совершившей ошибку программой. Напротив, при работе под управлением ОС отладка возможна лишь при активном сотрудничестве со стороны ОС. В чем именно должно состоять это сотрудничество, мы поймем, когда изучим, каким образом ОС изолирует задачи друг от друга, т. е. в главе 5.

2.10.3. Удаленная отладка

Потребность в удаленной отладке возникает в целом ряде ситуаций, назовем наиболее важные:

- целевая система, на которой должна работать отлаживаемая программа, не имеет устройств пользовательского интерфейса (экрана, клавиатуры, мы-

ши и т. д.), необходимых для работы отладчика. Чаще всего эта ситуация возникает при разработке программ в режиме кросс-компиляции;

- целевая система имеет недостаточно ресурсов (скорее всего, недостаточно ОЗУ), чтобы вместить и отлаживаемую программу, и полноценный отладчик;
- отлаживаемая программа слишком активно использует устройства и/или модули ОС, обеспечивающие пользовательский интерфейс, так что работа пользовательского интерфейса отладчика приводит к искажениям работы программы;
- отлаживаемая программа представляет собой модуль ядра ОС (в действительности, это частный случай предыдущей ситуации, но он достаточно специфичен, чтобы рассматривать его отдельно).

Принцип удаленной отладки довольно прост. Удаленный отладчик состоит из двух подсистем. Одна из них представляет "системную" часть отладчика, которая умеет ставить точки останова и просматривать и модифицировать адресное пространство отлаживаемой программы. Этот модуль исполняется на целевой системе вместе с программой. Обычно его называют агентом удаленной отладки.

Вторая часть представляет собой пользовательский интерфейс отладчика и исполняется на инструментальной системе. Системная часть оповещает пользовательскую о сработавших точках останова и других заслуживающих внимания событиях, например, о падении программы в результате ошибки, перехваченной ОС целевой системы. Напротив, пользовательская часть оповещает системную о командах пользователя: запросах на установку точек останова и т. д.

Главная тонкость здесь состоит в том, что доступ к исходным текстам и отладочной информации нужен только пользовательской, но не системной части отладчика, поэтому при удаленной отладке наиболее логично осуществлять компиляцию и хранение исходных текстов на инструментальной, а не на целевой системе.

Механизмы взаимодействия между системной и пользовательской частями отличаются довольно большим разнообразием.

Наиболее распространены отладка по последовательному порту (если отлаживаемая программа сама работает с последовательным портом, то, разумеется, целевая система должна иметь как минимум два порта) и отладка по сети, обычно с помощью транспортных протоколов с надежной доставкой, например, TCP или SPX. Протоколы такого взаимодействия не стандартизованы, потому отладчики разных поставщиков обычно не могут взаимодействовать друг с другом, каждый поставщик использует собственные механизмы и протоколы.

2.10.4. Деассемблирование

В разд. 2.10.1 говорилось, что для отладки необходим доступ к исходным текстам программы и отладочной информации. Я, в принципе, разделяю убеждение основателя FSF (Free Software Foundation, фонд свободного программного обеспечения) Ричарда Столлмэна, что распространение программ без исходных текстов — практика незэффективная, порождающая много неудобств и нежелательных явлений, и в определенном смысле даже аморальная. Однако необходимо признать, что это убеждение по разным причинам до сих пор не получило должного распространения и программы без исходных текстов встречаются сплошь и рядом. Это относится как к самодостаточным программам, так и к разделяемому коду, различным библиотекам и т. д.

Необходимо также принимать во внимание, что для старых программ, особенно для разрабатывавшихся *in-house*, для собственных нужд, исходные тексты иногда — и при этом чаще, чем хотелось бы — бывают утеряны без какого-либо злого умысла со стороны разработчиков.

Особую роль деассемблирование играет при анализе поведения различных вредоносных программ, таких как вирусы и сетевые черви, исходные тексты которых практически всегда недоступны.

Если исходные тексты утеряны или недоступны, но все-таки необходимо понять или откорректировать поведение программы, то можно воспользоваться специальными инструментами, объединяемыми под общим названием *деассемблеров*, или *дизассемблеров*.

Очевидно (и для многих ЯВУ формально доказано), что восстановление исходного текста на языке высокого уровня на основе машинного кода, в общем случае, представляет собой неразрешимую задачу. Известные программы "декомпиляции" обычно просто генерируют исходный текст на ЯВУ, функционально эквивалентный анализируемой программе; при компиляции такого текста получится совсем другая программа, как правило, гораздо более медленная и гораздо большая по объему, чем исходная, а часто все-таки и не эквивалентная исходной с точки зрения функциональности.

Полное восстановление исходного текста на языке ассемблера, вместе со всеми макроопределениями и именами символьических меток также невозможно, однако здесь можно сделать довольно много. Действительно, поскольку поток команд декодируется центральным процессором, который, в конечном итоге, представляет собой относительно простой микропрограммный автомат, то не так уж сложно написать программу, которая воспроизводит логику работы этого автомата — однако вместо того, чтобы отрабатывать требуемые расшифрованными командами действия, эта программа могла бы выводить мнемонические обозначения найденных команд. При реализации

подобной программы возникают некоторые сложности, но на практике они не очень велики.

Во-первых, на процессорах манчестерской архитектуры не всегда бывает легко отличить код от данных. Сложные деассемблеры, такие как IDA, применяют различные и порой довольно сложные эвристики; главным образом, используется информация о том, каким образом остальная программа обращается к данной области памяти — передает на нее управление или пытается прочитать или модифицировать. Простые деассемблеры, например, встроенные деассемблеры отладчиков или такие программы, как `biew` или `hiew`, просто пытаются декодировать все подряд, не смущаясь получаемым при декодировании данных бессмысленным "кодом"(примеры 2.13, 2.14).

Пример 2.13. Машинный код для процессора x86 (двоичное представление)

```
.00010070: 20 63 6F 6D-6D 61 6E 64-20 25 73 0A-00 55 89 E5
.00010080: 81 EC 28 01-00 00 8D 85-E8 FE FF FF-50 8D 45 E8
.00010090: 50 68 38 00-01 00 FF 75-08 E8 8A 05-00 00 83 C4
.000100A0: 10 83 EC 08-68 3F 00 01-00 8D 45 E8-50 E8 6E 04
.000100B0: 00 00 83 C4-10 85 C0 75-26 83 EC 04-8D 85 E8 FE
.000100C0: FF FF 50 68-44 00 01 00-FF 75 08 E8-A8 04 00 00
.000100D0: 83 C4 10 C7-85 E4 FE FF-FF C8 00 00-00 EB 21 83
.000100E0: EC 04 8D 45-E8 50 68 65-00 01 00 FF-75 08 E8 85
.000100F0: 04 00 00 83-C4 10 C7 85-E4 FE FF FF-F4 01 00 00
.00010100: 8B 85 E4 FE-FF FF C9 C3
```

Пример 2.14. Машинный код для процессора x86 (декодированное представление)

```
.00010070: 20636F           and      [ebx] [0006F], ah
.00010073: 6D              insd
.00010074: 6D              insd
.00010075: 61              popad
.00010076: 6E              outsb
.00010077: 642025730A0055 and      fs:[055000A73], ah
.0001007E: 89E5             mov      ebp, esp
.00010080: 81EC28010000    sub      esp, 000000128 ;" ;(" ;
.00010086: 8D85E8FEFFFF    lea      eax, [ebp] [0FFFFFFEE8]
.0001008C: 50              push     eax
.0001008D: 8D45E8           lea      eax, [ebp] [-0018]
```

```
.00010090: 50          push    eax
.00010091: 6838000100      push    000010038 ;" ! 8"
.00010096: FF7508      push    d, [ebp] [00008]
.00010099: E88A050000      call    .000010628 ----- (1)
```

Примечание

В примере 2.13 приведен тот же код, что и в примере 2.1.

В примере 2.14 дано декодированное представление, деассемблирование начато с байта 0x10070, по которому размещен строковый литерал, поэтому программа декодирована ошибочно.

Во-вторых, у процессоров с переменной длиной команд результат декодирования нередко зависит от точки, с которой было начато декодирование. Простые деассемблеры, не имеющие доступа к контексту исполнения программы, такие как `hiew`, опять-таки декодируют, как придется; при пролистывании программы нередко можно видеть, как результат декодирования вдруг резко меняется. Разумеется, только один из возможных результатов декодирования является "правильным"; впрочем, не надо быть таким уж большим гуру в ассемблере, чтобы отличить "правильный" вариант декодирования от неправильных. Проще всего это делать, находя в декодированном коде стандартные элементы кода, такие как прологи и эпилоги процедур, в которых создаются и уничтожаются записи активации. Встроенный деассемблер отладчика, разумеется, может использовать адрес текущей команды (который, при нормальном исполнении, практически всегда указывает на "правильное" начало команды) и отладочную информацию, когда она доступна.

Разумеется, обе эти проблемы легче всего могут быть решены при анализе полного загрузочного модуля, где указан стартовый адрес, на который необходимо передать управление после загрузки, и разделены сегменты кода и данных. Разумеется, для этого необходимо уметь анализировать заголовок и структуры данных загрузочного модуля, но практически все сколько-нибудь приличные деассемблеры это могут.

Так или иначе, даже с помощью крайне тупого деассемблера можно довольно легко увидеть последовательности команд исследуемой программы, а глядя на эти последовательности, часто удается довольно много понять в функциональности программы. Нужно признать, что анализ функциональности деассемблированных программ — деятельность скорее творческая, чем алгоритмизуемая, хотя в этой сфере есть немало заслуживающих изучения эмпирических приемов; литература, которая могла бы быть полезна при изучении данных приемов, не очень обширна, но доступен ряд хороших книг, например, [Касперски 2001, Пирогов 2006].

Видно, что в примере 2.1 восстановлены мнемонические обозначения команд, но вместо меток используются численные адреса. Сложные деассемб-

леры, такие как уже упоминавшийся IDA, могут генерировать символьные метки для всех адресов, на которые ссылаются какие-либо команды (будь то команды передачи управления или команды обращения к данным), и потом заменять адреса названиями меток в мнемониках всех этих команд. Такие метки (хотя они и символьные), разумеется, имеют бессмысленные имена — сами по себе они мало что добавляют к пониманию программы, но при генерации этих меток деассемблер вынужден породить так называемую таблицу перекрестных ссылок, которую по запросу он также может вывести. Эта таблица, в свою очередь, сама по себе может немало сообщить о структуре исследуемой программы и очень полезна в качестве вспомогательного инструмента при дальнейшем ее анализе.

Впрочем, расстановка символьических меток очень полезна с той точки зрения, что после нее деассемблированная программа превращается в полноценный текст на языке ассемблера. Такая программа, например, может быть скомпилирована ассемблером для другого процессора, который несовместим с исходным на бинарном уровне, но совместим на уровне языка ассемблера. Таким образом, даже не разбираясь в логике программы, можно перевести ее в другую систему команд.

Возможности, которые деассемблирование предоставляет для анализа поведения программ и их "обратного проектирования" вызывают беспокойство у групп, лоббирующих расширение и ужесточение законодательства об охране "интеллектуальной собственности" и издательских прав (слово *copyright* часто ошибочно переводят на русский язык словосочетанием "авторские права", но сама эта правовая концепция к авторским правам имеет, в лучшем случае, опосредованное отношение, да и буквальный перевод слова — "право на копирование" — однозначно указывает, что речь идет о правах издателя, а не автора). Разумеется, "пользовательские лицензионные соглашения" практически всех коммерческих и некоторых некоммерческих программ запрещают деассемблирование, но юридический статус самих этих соглашений до сих пор окончательно не ясен.

К счастью, законодательство, действующее в России на время издания этой книги, явно разрешает деассемблирование программ и их модификацию к нуждам пользователя. Впрочем, даже в тех странах, в которых лоббистам удалось продавить законодательный запрет на деассемблирование коммерческих программ и запрет на использование "обратного проектирования" для снятия средств защиты от нелицензионного использования (таких как DMCA в США) само по себе владение деассемблером не является преступлением. Действительно, этот инструмент необходим и для многих целей, не имеющих отношения к нарушению издательских прав.

Кроме анализа поведения программ, для которых недоступны исходники, деассемблеры также незаменимы при отладке компиляторов, ассемблеров и

других программ, работающих с машинным кодом: компоновщиков, загрузчиков и т. д.

Вопросы для самопроверки

1. Почему стековая архитектура считается одноадресной, а не безадресной?
2. Как в архитектуре регистр-регистр осуществляется доступ к памяти?
3. Какие регистры называются регистрами общего назначения?
4. Перечислите режимы адресации, поддерживаемые процессором SPARC.
5. Если вы знакомы с системой команд какого-либо процессора, перечислите режимы адресации, поддерживаемые этим процессором. Используйте названия режимов в соответствии с терминологией, используемой в этой главе, а не из документации к процессору. Сколько регистров имеет этот процессор?
6. Зачем двухпроходному ассемблеру требуется два прохода? Как эта проблема решается в однопроходном ассемблере?
7. Как отладчик определяет, какая строка исходного кода сейчас исполняется?

Вопросы, над которыми следует задуматься

1. Почему производители процессоров уделяют такое большое внимание обеспечению совместимости (как бинарной, так и по языку ассемблера) с предыдущими поколениями своих процессоров и/или популярными процессорами других производителей?
2. Если совместимость так важна, как новые архитектуры процессоров могут появляться на рынке?
3. В разд. 2.10.2 говорится, что команда точки останова должна быть равна по длине самой короткой из прочих команд процессора (в действительности, эта команда может быть короче всех остальных команд). Почему это требование так важно, ведь если точка останова занимает место нескольких команд, то при прохождении точки останова эти команды могли бы быть сохранены и потом с таким же успехом выполнены, как и единственная команда?
4. В том же разделе описывается процедура обработки точки останова: утверждается, что команда, перезаписанная точкой останова, сохраняется отладчиком и записывается на место при проходе этой точки. Но если все происходит именно так, то после первого прохождения точка останова снимется. Вопрос: как можно реализовать постоянную (срабатывающую много раз) точку останова? Можно ли это сделать без применения пошагового исполнения?



ГЛАВА 3

Загрузка программ

Процесс пошел.
М. С. Горбачев

Выяснив, что представляет собой программа, давайте рассмотрим процедуру ее загрузки в оперативную память компьютера (многие из обсуждаемых далее концепций в известной мере применимы и к прошивке программы в ПЗУ).

Способы загрузки программ существенно различаются в разных средах программирования. Важно понимать, что способ загрузки не полностью определяется операционной системой: программы, написанные для одной и той же ОС, но на разных языках, вполне могут загружаться по-разному.

В первом приближении все используемые системы программирования можно разделить на два больших класса: системы с ранним (или, что то же самое, статическим) связыванием (*early/static binding*) и системы с поздним (или, что то же самое, динамическим) связыванием (*late/dynamic binding*) [Себеста 2001]. Связывание или, как говорят в других русскоязычных книгах, привязка (*binding*) в данном случае означает попросту определение адреса именованного объекта (переменной, процедуры, функции и др.). Во многих случаях технология связывания явно или неявно определяется спецификациями языка программирования.

В системах с динамическим связыванием программа определяет адрес функции в момент вызова этой функции. При определенных обстоятельствах может даже оказаться, что в разные моменты по одному и тому же имени могут быть вызваны разные функции. Это может использоваться в различных целях, в том числе и во вполне мирных. Так, перегруженные методы в объектно-ориентированных языках программирования, несомненно, требуют динамического связывания.

Примеры языков с динамическим связыванием — большинство диалектов языка Basic, Java, C#, Perl. Многие из этих языков осуществляют во время

исполнения не только сборку, но и компиляцию программы в машинный код. Это называется JIT-компиляцией (*Just In Time* — непосредственно в момент исполнения). Некоторые реализации даже вообще не генерируют машинного кода, а просто интерпретируют исходный код. Такие среды исполнения называются интерпретаторами (*interpreter*).

Современные реализации виртуальных машин Java и C# могут использовать своеобразную гибридную стратегию: сразу после запуска они некоторое время интерпретируют программу и набирают статистику о том, как она исполняется, насколько часто вызываются те или иные функции и блоки кода и т. д. Если программа работает достаточно долго, то через некоторое время после запуска интерпретатор генерирует машинный код и передает управление ему. Теоретически, JIT-компиляция в таком режиме могла бы порождать более эффективный код, чем статические оптимизирующие компиляторы — ведь статический компилятор делает лишь предположения о том, как будет работать генерируемая им программа, а JIT-компилятор описываемого типа имеет доступ к реальной статистике исполнения.

В системах со статическим связыванием программа считает, что адреса всех используемых символьических объектов (переменных, функций и т. д.) определены заранее. Как мы увидим далее в этой главе, реальное распределение адресов объектов может происходить в разное время. В некоторых случаях оно происходит во время компиляции, в других — уже в момент загрузки, но все-таки всегда до запуска программы.

Примеры языков со статическим связыванием: ассемблеры, Fortran, Pascal, C, C++. Статическое связывание позволяет прописывать адреса объектов непосредственно в адресные поля команд, работающих с этими объектами, поэтому в большинстве случаев оно приводит к повышению скорости исполнения программы.

Как уже отмечалось, объектно-ориентированные языки в определенных ситуациях требуют динамического связывания. В C++ и объектно-ориентированных диалектах Pascal эта сложность была обойдена за счет усложнения языка: методы, для которых требуется динамическая привязка, описываются несколько иначе, чем методы, для которых достаточно привязки статической. В C++ для этого используется ключевое слово *virtual*. Даже в C есть понятие, которое могло бы быть использовано (и используется) для реализации динамического связывания — указатель на функцию.

Так или иначе, во всех системах со статическим связыванием явно выделяется фаза сборки или компоновки (*linking*), которая следует за компиляцией и выполняется отдельной программой — так называемым компоновщиком (*linker*) или редактором связей (*link editor*). В результате сборки получается специальный файл, называемый загрузочным или загружаемым модулем (*executable*). Этот файл содержит машинный код порожденной программы и,

возможно, некоторую дополнительную информацию. Дело в том, что сборка не всегда приводит к привязке всех объектов в программе, зачастую собственно привязка (окончательное определение адреса объекта и вычисление значений адресных ссылок в коде) происходит только в момент загрузки программы. Поэтому компоновщик вынужден сохранять в загрузочном модуле информацию, которой мог бы воспользоваться загрузчик. Далее в этой главе мы постараемся разобраться во всех этих тонкостях.

Приведенная классификация в определенном смысле условна. Так, есть примеры загрузчиков с отложенным или ленивым (*lazy/late binding*) связыванием — например, рассматриваемый в разд. 5.4 загрузчик для модулей формата ELF. Ленивый компоновщик определяет адрес функции во время исполнения при первом обращении к ней. Таким образом, функции, к которым не было обращений, так никогда и не привязываются, как при динамическом связывании. Но, в отличие от "честного" динамического компоновщика, ленивый запоминает результаты разрешения имен, и все последующие обращения к функции будут происходить по ее адресу. Ленивую сборку можно использовать как для программ на языках, рассчитанных на статическое связывание, так и — с некоторыми предосторожностями — для языков с динамическим связыванием. В действительности, большинство современных реализаций Basic, Java и C# в тех случаях, когда это допускается семантикой языка, используют ленивую, а не динамическую компоновку.

Далее в этой главе мы будем рассматривать преимущественно системы программирования с ранним связыванием.

Для того чтобы не путаться, давайте будем называть *программой* ту часть загрузочного модуля, которая содержит исполняемый код. Результат загрузки программы в память будем называть *процессом* или, если нам надо отличать загруженную программу от процесса ее исполнения, *образом процесса*. К образу процесса иногда причисляют не только код и данные процесса (подвергнутые преобразованию как в процессе загрузки, так и в процессе работы программы), но и системные структуры данных, связанные с этим процессом. В старой литературе процесс часто называют *задачей* (task).

В системах с виртуальной памятью каждому процессу обычно выделяется свое адресное пространство, поэтому мы иногда будем употреблять термин "процесс" и в этом смысле. Впрочем, во многих системах значительная часть адресных пространств разных процессов перекрывается — это используется для реализации разделяемого кода и данных.

В рамках одного процесса может исполняться один или несколько *потоков* или *нитей управления*. Это понятие будет подробнее разбираться в главе 8.

Некоторые системы предоставляют и более крупные структурные единицы, чем процесс. Например, в системах семейства Unix существуют группы про-

цессов, которые используются для реализации логического объединения процессов в *задания* (*job*). Ряд систем имеют также понятие *сессии* — совокупности всех заданий, которые пользователь запустил в рамках одного сеанса работы. Впрочем, соответствующие концепции часто плохо определены, а их смысл сильно меняется от одной ОС к другой, поэтому мы практически не будем обсуждать эти понятия.

В более старых системах и в старой литературе называют результат загрузки задачей, а процессами — отдельные нити управления. Однако в наиболее распространенных ныне ОС семейств Unix и Win32, принято задачу называть процессом, а процесс — нитью (*thread*). Этой терминологии мы и будем придерживаться, кроме тех случаев, когда будем обсуждать примеры из жизни ОС, в которой принятая иная терминология.

Создание процессов в Unix

В системах семейства Unix новые процессы создаются системным вызовом *fork*. Этот вызов создает два процесса, образы которых в первый момент полностью идентичны, у них различается только значение, возвращенное вызовом *fork*. Типичная программа, использующая этот вызов, выглядит так, как представлено в примере 3.1.

При этом каждый из процессов имеет свою копию всех локальных и статических переменных. На процессорах со страничным диспетчером памяти физического копирования не происходит. Изначально оба процесса используют одни и те же страницы памяти, а дублируются только те из них, которые были изменены. На системах, не имеющих страничного или сегментного диспетчера памяти, *fork* требует копирования адресных пространств, что приводит к большим накладным расходам, да и просто не всегда возможно.

Если мы хотим запустить другую программу, то мы должны исполнить системный вызов из семейства *exec*s. Вызовы этого семейства различаются только способом передачи параметров. Все они прекращают исполнение текущего образа процесса и создают новый процесс с новым виртуальным адресным пространством, но с тем же идентификатором процесса. При этом у нового процесса будет тот же приоритет, будут открыты те же файлы (это часто используется), и он унаследует ряд других важных характеристик.

Несколько неожиданное, но тем не менее верное описание действия *exec* — это замена образа процесса в рамках того же самого процесса [Хевиленд/Грей/Салама 2000].

Запуск другой программы в UNIX выглядит примерно так, как представлено в примере 3.2. Программа в примере 3.2 запускает командный интерпретатор */bin/sh*, приказывает ему выполнить команду *ls -l* и перенаправляет стандартный вывод этой команды в файл *ls.log*.

Техника программирования, основанная на *fork/exec*, несколько отличается от принятой во многих других современных системах, в том числе Win32. В этих системах при создании нового процесса мы сразу же указываем программу, которую он будет выполнять.

Пример 3.1. Создание процесса в системах семейства Unix

```
int pid; /* Идентификатор порожденного процесса */

switch(pid = fork())
{
    case 0: /* Порожденный процесс */
        .....
        break;
    case -1: /* Ошибка */
        perror("Cannot fork");
        exit(1);
    default: /* Родительский процесс */
        .....
    /* Здесь мы можем ссылаться на порожденный процесс,
     * используя значение pid */
}
```

Пример 3.2. Создание процесса и замена программы в системах семейства Unix

```
int pid; /* Идентификатор порожденного процесса */

switch(pid = fork())
{
    case 0: /* Порожденный процесс */
        dup2(1, open("ls.log", O_WRONLY | O_CREAT));
        /* Перенаправить открытый файл #1
         * (stdout) в файл ls.log */

        execl("/bin/sh", "sh", "-c", "ls", "-l", 0);

        /* Сюда мы попадаем только при ошибке! */
        /* fall through */
    case -1: /* Ошибка */
        perror("Cannot fork or exec");
        exit(1);
    default: /* Родительский процесс */
        .....
    /* Здесь мы можем ссылаться на порожденный
     * процесс, используя значение pid */
}
```

3.1. Абсолютная загрузка

Самый простой способ загрузки состоит в том, что мы всегда будем загружать программу с одного и того же адреса. Это возможно в следующих случаях:

- система может предоставить каждому процессу свое адресное пространство. Это возможно только на процессорах, осуществляющих трансляцию виртуального адреса в физический;
- система может исполнять в каждый момент только один процесс. Так ведет себя СР/М, так же устроено большинство загрузочных мониторов для самодельных компьютеров. Похожим образом устроена система RT-11, но о ней далее.

Загрузочный файл, используемый при таком способе, называется *абсолютным загрузочным модулем*. При создании такого модуля компилятор или компоновщик должен знать адрес, по которому будет происходить загрузка — проще всего это обеспечить при выполнении указанных ранее условий. Компоновщик сразу размещает все объекты (переменные и участки кода) так, как они должны быть расположены после загрузки. Поскольку во время сборки становится известно окончательное размещение всех объектов, то в это же время можно окончательно разрешить все адресные ссылки, как в коде, так и в сегменте данных. Разрешенные ссылки записываются в адресные поля команд и в переменные, которым программист при описании присвоил значения типа адрес.

Ни в какой дополнительной настройке в момент загрузки абсолютный модуль не нуждается.

Начальное содержимое образа процесса формируется путем простого копирования загрузочного модуля в память. В системе RT-11 такие файлы имеют расширение sav (от saved — сохраненный).

Формат загрузочного модуля a.out

В системе UNIX на 32-разрядных машинах также используется абсолютная загрузка. Загружаемый файл формата a.out (современные версии Unix используют более сложный формат загружаемого модуля и более сложную схему загрузки, которая будет обсуждаться в разд. 5.4) начинается с заголовка (рис. 3.1), который содержит:

- "магическое число" — признак того, что это именно загружаемый модуль, а не что-то другой;
- число TEXT_SIZE — длину области кода программы (TEXT);
- DATA_SIZE — длину области инициализированных данных программы (DATA);
- BSS_SIZE — длину области неинициализированных данных программы (BSS);
- стартовый адрес программы.

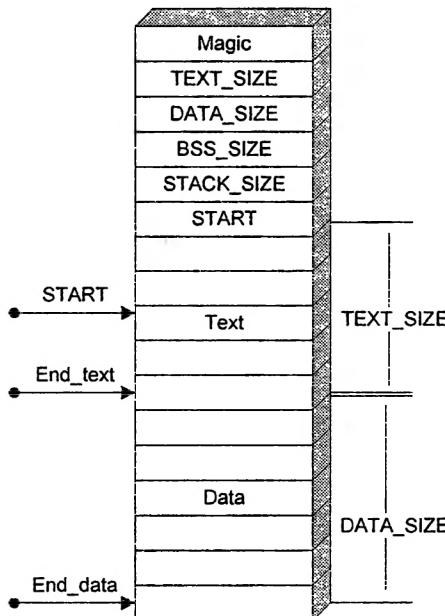


Рис. 3.1. Загрузочный модуль a.out

За заголовком следует содержимое областей TEXT и DATA. Затем может следовать отладочная информация. Она нужна символьным отладчикам, но самой программой не используется.

При загрузке система выделяет процессу **TEXT_SIZE** байтов виртуальной памяти, доступной для чтения/исполнения, и копирует туда содержимое сегмента TEXT. Затем отсчитывается **DATA_SIZE** байтов памяти, доступной для чтения/записи, и туда копируется содержимое сегмента DATA. Затем отсчитывается еще **BSS_SIZE** байтов памяти, доступной для чтения/записи, которые прописываются нулями.

Очистка выделяемой памяти нужна не столько для удобства программиста, сколько по соображениям безопасности: перед вновь загружаемым процессом эту память могли занимать (а при сколько-нибудь длительной работе системы почти наверняка занимали) другие процессы, которые могли использовать эту память для хранения важных и секретных данных, например паролей или ключей шифрования.

После этого выделяется пространство под стек, в стек помещаются позиционные аргументы и среда исполнения (environment) далее управление передается на стартовый адрес. Процесс начинает исполняться.

3.2. Разделы памяти

Одним из способов обойти невозможность загружать более одной программы при абсолютной загрузке являются *разделы памяти*. В наше время этот метод

практически не применяется, но в машинах второго поколения использовался относительно широко и часто описывается в старой литературе.

Идея метода состоит в том, что мы задаем несколько допустимых стартовых адресов для абсолютной загрузки. Каждый такой адрес определяет раздел памяти (рис. 3.2). Процесс может размещаться в одном разделе или, если это необходимо — т. е. если образ процесса слишком велик — в нескольких, что позволяет загружать несколько процессов одновременно, сохраняя при этом преимущества абсолютной загрузки.

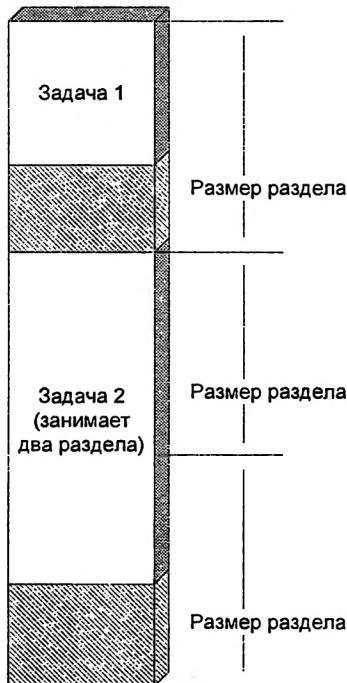


Рис. 3.2. Разделы памяти

Если мы не знаем, в какой из разделов пользователь вынужден будет загружать нашу программу, мы должны предоставить по отдельному загрузочному модулю на каждый из допустимых разделов. Понятно, что это не очень практично, потому разделы были вытеснены более удобными схемами управления памятью.

3.3. Относительная загрузка

Относительный способ загрузки состоит в том, что мы загружаем программу каждый раз с нового адреса. При этом в момент компиляции и сборки мы не

можем знать адреса, с которого будет происходить загрузка. Максимум, что мы можем определить в момент сборки, — это размещение объектов друг относительно друга. Если при загрузке весь модуль будет загружен в непрерывную область памяти, то относительное размещение можно сохранить. Благодаря этому, при сборке программы мы знаем разности адресов объектов (или, что то же самое, их смещения друг относительно друга), но не можем знать их окончательных адресов. При загрузке такой программы мы должны настроить ее на новые адреса, а для этого нам надо вспомнить материал предыдущей главы и понять, что же именно в программе привязано к адресу загрузки.

При использовании в коде программы абсолютной адресации мы должны найти адресные поля всех команд, использующих такую адресацию, и пересчитать эти адресные поля с учетом реального адреса загрузки (рис. 3.3). Если в коде программы применялись косвенно-регистровый, базовый и базово-индексный режимы адресации, следует найти те места, где в регистр загружается значение адреса (рис. 3.4).

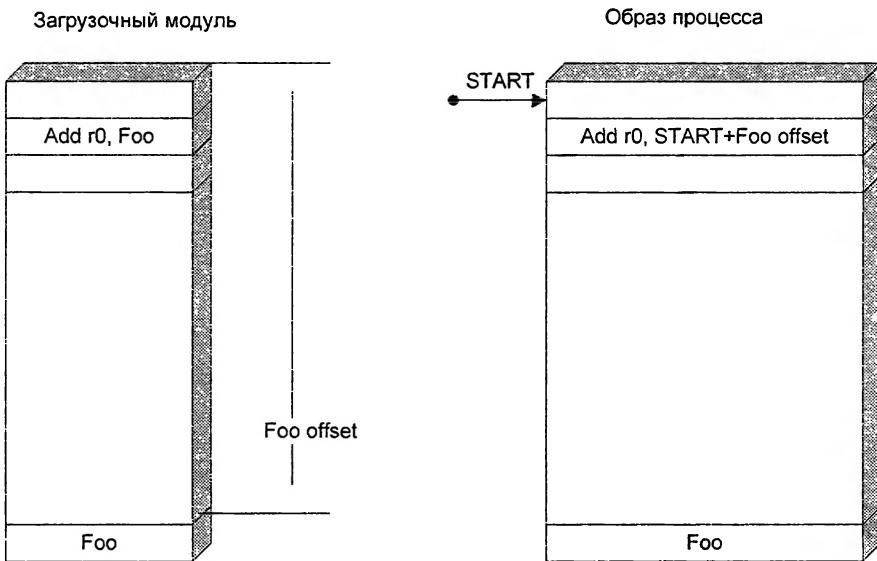


Рис. 3.3. Перемещение кода, использующего абсолютную адресацию

Сложность здесь в том, что если абсолютные адресные поля можно найти анализом кодов команд (деассемблированием), то значение в адресный регистр может загружаться задолго до собственно адресации, причем, как мы видели в примерах кода для процессора SPARC, формирование значения регистра может происходить и по частям. Без помощи программиста или ком-

пилота (в этой главе мы не будем различать написанный на ассемблере или компилированный код, а того, кто генерировал код, будем называть программистом) решить вопрос о том, какая из команд загружает в регистр скалярное значение, а какая — будущий адрес или часть адреса, невозможно. Та же проблема возникает в случае, если мы используем в качестве указателя ячейку статически инициализированных данных (пример 3.3).

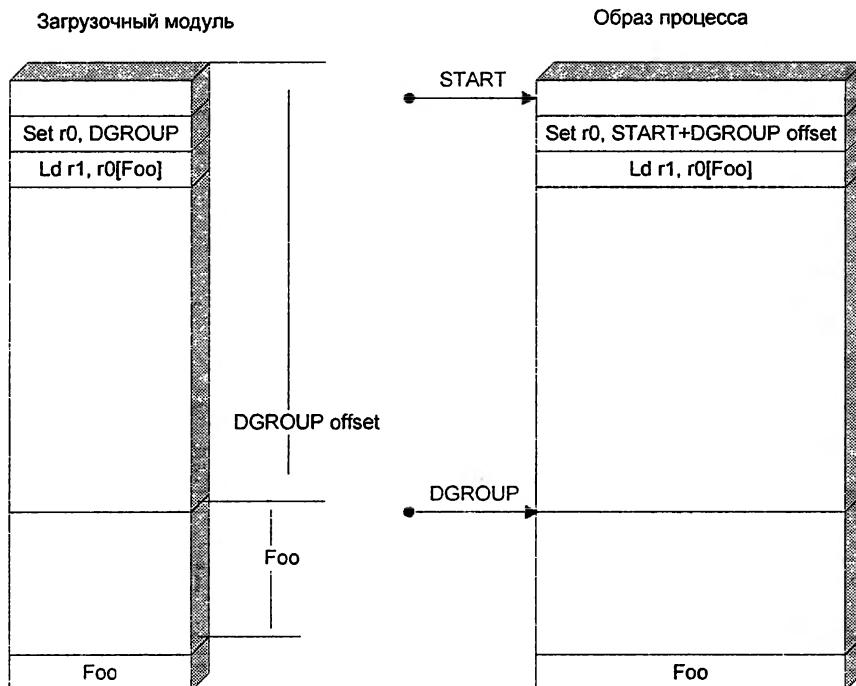


Рис. 3.4. Перемещение кода, самостоятельно перезагружающего базовые регистры

Пример 3.3. Примеры статически инициализированных указателей в С

```
int buf[20], *bufptr=buf;
char * message="No message defined yet\n";
void do_nothing_hook(int);
void (*hook)(int)=do_nothing_hook;
```

Довольно легко построить и пример кода, в котором адресация происходит вообще без явного использования каких-либо регистров, во всяком случае, без загрузки в них значений (пример 3.4).

Пример 3.4. Реализация косвенного перехода по адресу addr

```
push addr ; Это и будет ссылкой на абсолютный адрес
ret
```

На практике содействие программиста загрузчику состоит в том, что программист старается без необходимости не использовать в адресных полях и в качестве значений адресных регистров произвольные значения (необходимость в этом может возникать при адресации системных структур данных или внешних устройств, расположенных по фиксированным адресам). Вместо этого, программист применяет ассемблерные символы, соответствующие адресам.

Ассемблер при каждой ссылке на такой символ генерирует не только "заготовку" адреса в коде, но и запись в *таблице перемещений* (relocation table). Эта запись хранит место ссылки на такой символ в коде или данных. Если в ссылке используется только часть адреса, как в командах `sethi %10, %hi(addr)` процессора SPARC, или `move ax, segment addr` процессора 8086, мы запоминаем и этот факт (тогда формат записи в таблице перемещений, разумеется, значительно усложняется). При настройке программы на реальный адрес загрузки нам, таким образом, необходимо пройти по всем объектам, перечисленным в таблице перемещений, и переместить каждую из ссылок — сформировать из заготовки адрес. В качестве "заготовки" адреса обычно используется смещение адресуемого объекта от начала программы и для перемещения оказывается достаточно добавить к "заготовке" адрес загрузки.

Файл, содержащий таблицу перемещений, гораздо сложнее абсолютного загружаемого модуля и носит название относительного или перемещаемого загрузочного модуля. Именно такой формат имеют EXE-файлы в системе MS-DOS (пример 3.5).

Пример 3.5. Заголовок EXE-файла MS-DOS. Цитируется по WIN32.H из поставки MS Visual C++ v6.0 (перевод комментариев автора)

```
#define IMAGE_DOS_SIGNATURE      0x4D5A // MZ
typedef struct _IMAGE_DOS_HEADER {        // Заголовок DOS .EXE
    WORD e_magic;                      // Магическое число (сигнатура)
    WORD e_cblp;                       // Длина последней страницы файла в байтах
    WORD e_cp;                         // Количество страниц в файле
    WORD e_crlc;                       // Количество перемещений
    WORD e_cparhdr;                    // Размер заголовка в параграфах
    WORD e_minalloc;                   // Минимальное количество дополнительных параграфов
```

```

WORD e_maxalloc;           // Максимальное количество дополнительных параграфов
WORD e_ss;                 // Начальное (относительное) значение SS
WORD e_sp;                 // Начальное значение SP
WORD e_csum;               // Контрольная сумма
WORD e_ip;                 // Начальное значение IP
WORD e_cs;                 // Начальное (относительное) значение CS
WORD e_lfarlc;             // Адрес таблицы перемещений в файле
WORD e_ovno;                // Номер перекрытия
WORD e_res[4];              // Зарезервировано
WORD e_oemid;               // OEM идентификатор (для e_oeminfo)
WORD e_oeminfo;              // Информация OEM; специфично для e_oemid
WORD e_res2[10];             // Зарезервировано
LONG e_lfanew;              // Адрес следующего заголовка в файле
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;

```

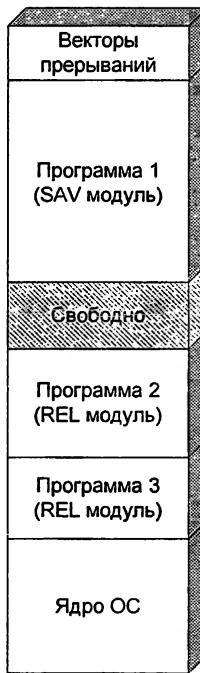


Рис. 3.5. Распределение памяти в RT-11 с одним загруженным SAV-файлом и двумя REL-файлами

Наиболее поучительна в этом отношении система RT-11, в которой поддерживаются загружаемые модули обоих типов. Обычные программы имеют расширение sav, представляют собой абсолютные загружаемые модули и гру-

зятся всегда с адреса 01000. Ниже этого магического адреса находятся векторы прерываний и стек программы. Сама операционная система вместе с драйверами размещается в верхних адресах памяти. Естественно, вы не можете загрузить одновременно два SAV-файла.

Однако, если вам обязательно нужно исполнять одновременно две программы, вы можете собрать вторую из них в виде относительного модуля: файла с расширением rel. Такая программа будет загружаться в верхние адреса памяти, каждый раз разные, в зависимости от конфигурации ядра системы, количества загруженных драйверов устройств и других REL-модулей (рис. 3.5).

Многие из современных относительных загрузчиков допускают еще более сложный формат загрузочных модулей, когда модуль может содержать несколько независимо перемещаемых секций. Например, модуль может содержать сегмент кода и сегмент данных, которые при загрузке размещаются по совершенно разным адресам. Благодаря этому образ загруженной программы не обязан занимать непрерывное пространство в памяти; вместо этого он может быть раскидан по нескольким участкам, количество которых совпадает с количеством секций.

3.4. Базовая адресация

Впрочем, если уж мы полагаемся на содействие программиста, можно пойти в этом направлении дальше. Мы объявляем один или несколько регистров процессора базовыми (несколько регистров могут использоваться для адресации различных сегментов программы, например, один — для кода, другой — для статических данных, третий — для стека) и договариваемся, что значения этих регистров программист принимает как данность и никогда сам не модифицирует, зато все адреса в программе он вычисляет на основе значений этих регистров (рис. 3.6).

В этом случае для перемещения программы нам нужно только изменить значения базовых регистров, и программа даже не узнает, что загружена с другого адреса. Статически инициализированными указателями в этом случае пользоваться либо невозможно, либо необходимо всегда прибавлять к ним значения базовых регистров.

Именно так происходит загрузка СОМ-файлов в MS-DOS. Система выделяет свободную память, настраивает для программы базовые регистры ds и cs, которые почему-то называются сегментными, и передает управление на стартовый адрес. Ничего больше делать не надо.

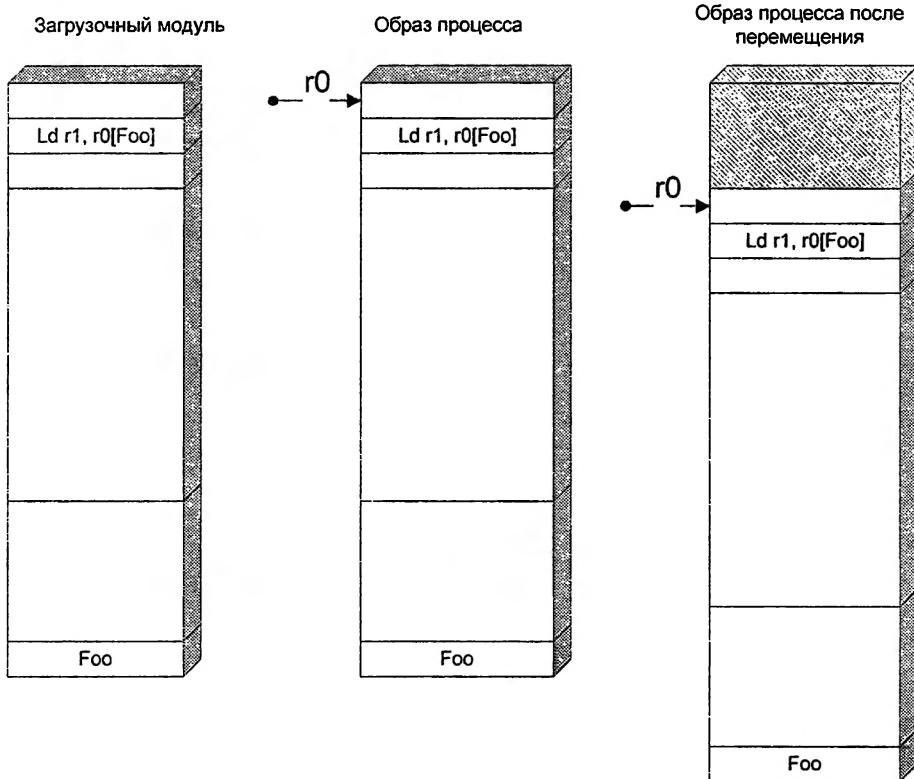


Рис. 3.6. Перемещение кода, использующего базовую адресацию

3.5. Позиционно-независимый код

За всеми этими разговорами мы чуть было не забыли о третьем способе формирования адреса в программе. Это относительная адресация, когда адрес получается сложением адресного поля команды и адреса самой этой команды — значения счетчика команд. Код, в котором используется только такая адресация, можно загружать с любого адреса без всякой перенастройки. Такой код называется *позиционно-независимым* (*position-independent*).

Позиционно-независимые программы очень удобны для загрузки, но, к сожалению, при их написании следует соблюдать довольно жесткие ограничения, накладываемые на используемые в программе методы адресации. Например, нельзя пользоваться статически инициализированными переменными указательного типа, нельзя делать на ассемблере фокусы, вроде того, который был приведен в примере 3.5, и т. д. Возникают сложности при сборке программы из нескольких модулей.

К тому же, на многих процессорах, например, на Intel 8080/8085 или многих современных RISC-процессорах, описанная ранее реализация позиционно-независимого кода вообще невозможна, т. к. эти процессоры не поддерживают соответствующий режим адресации для данных. На процессорах гарвардской архитектуры адресовать данные относительно счетчика команд вообще невозможно — команды находятся в другом адресном пространстве.

Поэтому этот стиль программирования используют только в особых случаях. Например, многие вирусы для MS-DOS и драйверы для RT-11 написаны именно таким образом.

Любопытное наблюдение

В эпоху RT-11 хакеры писали драйверы. Сейчас они пишут вирусы. Еще любопытнее, что для некоторых персональных платформ, например Amiga, вирусов почти нет. Хакеры считают более интересным писать игры или демонстрационные программы для Amiga. Похоже, общение с IBM PC порождает у программиста какие-то агрессивные комплексы. Наблюдение это принадлежит не мне: см. [КомпьютерПресс 1993].

Позиционно-независимый код в современных Unix-системах

Компиляторы современных систем семейства UNIX (GNU C или стандартный C-компилятор UNIX SVR4) имеют ключ `-f PIC` (Position-Independent Code). Впрочем, код, порождаемый при использовании данного ключа, не является позиционно-независимым в указанном ранее смысле: этот код все-таки содержит перемещаемые адресные ссылки. Задача состоит не в том, чтобы избавиться от таких ссылок полностью, а лишь в том, чтобы собрать все эти ссылки в одном месте и разместить их, по возможности, отдельно от кода. Какая от этого польза, мы поймем несколько позже, в разд. 5.4, а сейчас обсудим технические приемы, используемые для решения этой задачи.

Код, генерируемый GNU C, использует базовую адресацию: в начале функции адрес точки ее входа помещается в один из регистров, и далее вся адресация других функций и данных осуществляется относительно этого регистра. На процессоре x86 используется регистр `%ebx`, а загрузка адреса осуществляется командами, вставляемыми в пролог каждой функции (пример 3.6).

На процессорах, где разрешен прямой доступ к счетчику команд, соответствующий код выглядит проще, но принцип сохраняется: компилятор занимает один регистр и благодаря этому упрощает работу загрузчику.

Как мы видим в примере 3.7, на самом деле адресация происходит не относительно точки входа в функцию, а относительно некоторого объекта, называемого GOT или `GLOBAL_OFFSET_TABLE`. Счетчик команд используется для вычисления адреса этой таблицы, а не сам по себе. Подробнее мы разберемся с логикой работы данного кода (и заодно с тем, что означает еще один непонятный символ — PLT) в разд. 5.4.

Компилированный таким образом код предназначен в первую очередь для разделяемых библиотек формата ELF (Executable and Linking Format, формат исполняемых и собираемых [модулей], используемый большинством современных систем семейства Unix).

Пример 3.6. Получение адреса точки входа в позиционно-независимую подпрограмму

```
call L4
L4:
popl %ebx
```

Пример 3.7. Позиционно-независимый код, порождаемый компилятором GNU C

```
/* strerror.c (emx+gcc) -- Copyright (c) 1990-1996 by Eberhard Mattes */

#include <stdlib.h>
#include <string.h>
#include <emx/thread.h>

char *strerror (int errnum)
{
    if (errnum >= 0 && errnum < _sys_nerr)
        return (char *)_sys_errlist [errnum];
    else
    {
        static char msg[] = "Unknown error ";
#if defined (_MT_)
        struct _thread *tp = _thread();
#define result (tp->_th_error)
#else
        static char result[32];
#endif
        memcpy (result, msg, sizeof (msg) - 1);
        _itoa (errnum, result + sizeof (msg) - 1, 10);
        return result;
    }
}

gcc -f PIC -S strerror.c

.file "strerror"
gcc2_compiled.:
__gnu_compiled_c:
.data
```

```
_msg.2:
    .ascii "Unknown error \0"
.lcomm _result.3,32
.text
    .align 2,0x90
.globl _strerror
_strerror:
    pushl %ebp
    movl %esp, %ebp
    pushl %ebx
    call L4
L4:
    popl %ebx
    addl $_GLOBAL_OFFSET_TABLE_+[.-L4], %ebx
    cmpl $0, 8(%ebp)
    j1 L2
    movl __sys_nerr@GOT(%ebx), %eax
    movl 8(%ebp), %edx
    cmpb %edx, (%eax)
    jle L2
    movl 8(%ebp), %eax
    movl %eax, %edx
    leal 0(%edx, 4), %eax
    movl __sys_errlist@GOT(%ebx), %edx
    movl (%edx, %eax), %eax
    jmp L1
    .align 2,0x90
    jmp L3
    .align 2,0x90
L2:
    pushl $14
    leal _msg.2@GOTOFF(%ebx), %edx
    movl %edx, %eax
    pushl %eax
    leal _result.3@GOTOFF(%ebx), %edx
    movl %edx, %eax
    pushl %eax
    call _memcpy@PLT
    addl $12, %esp
    pushl $10
```

```

leal _result.3@GOTOFF(%ebx),%edx
leal 14(%edx),%eax
pushl %eax
movl 8(%ebp),%eax
pushl %eax
call __itoa@PLT
addl $12,%esp
leal _result.3@GOTOFF(%ebx),%edx
movl %edx,%eax
jmp L1
.align 2,0x90

L3:
L1:
    movl -4(%ebp),%ebx
    leave
    ret

```

3.6. Оверлеи (перекрытия)

Еще более интересный способ загрузки программы — это *оверлейная загрузка* (overlay — дословно, лежащий сверху) или, как это называли в старой русскоязычной литературе, *перекрытие*. Смысл оверлея состоит в том, чтобы не загружать программу в память целиком, а разбить ее на несколько модулей и помещать их в память по мере необходимости. При этом на одни и те же адреса в различные моменты времени будут отображены разные модули (рис. 3.7). Отсюда и название.

Потребность в таком способе загрузки появляется, если у нас виртуальное адресное пространство мало, например 1 Мбайт или даже всего 64 Кбайт (на некоторых машинах с RT-11 бывало и по 48 Кбайт — многие полезные программы нормально работали!), а программа относительно велика. На современных 32-разрядных системах виртуальное адресное пространство обычно измеряется гигабайтами, и большинству программ этого хватает, а проблемы с нехваткой можно решать совсем другими способами. Тем не менее существуют различные системы, даже и 32-разрядные, в которых нет устройства управления памятью, и размер виртуальной памяти не может превышать объема микросхем ОЗУ, установленных на плате. Пример такой системы — упоминавшийся ранее Transputer.

Важно подчеркнуть, что, несмотря на определенное сходство между задачами, решаемыми механизмом перекрытий и виртуальной адресацией, одно ни в коем случае не является разновидностью другого. При виртуальной адреса-

ции мы решаем задачу отображения большого адресного пространства на ограниченную оперативную память. При использовании оверлея мы решаем задачу отображения большого количества объектов в ограниченное адресное пространство.

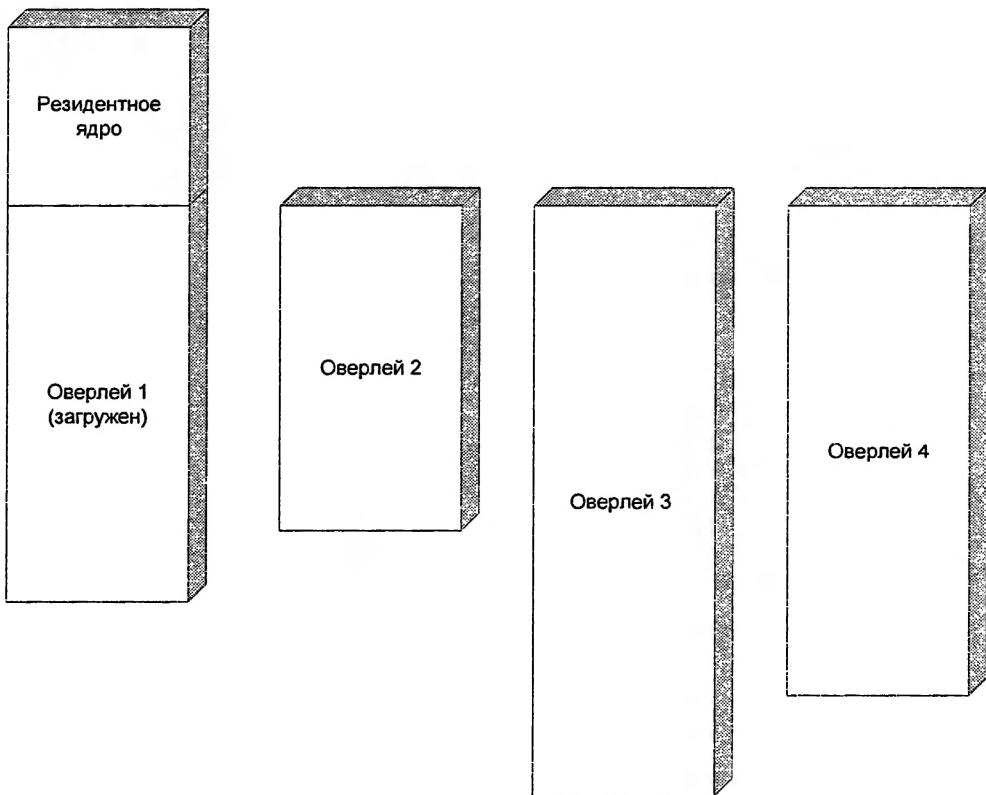


Рис. 3.7. Образ процесса с несколькими оверлеями

Основная проблема при оверлейной загрузке состоит в следующем: прежде чем ссылаться на оверлейный адрес, мы должны понять, какой из оверлейных модулей в данный момент там находится. Для ссылок на функции это просто: вместо точки входа функции мы вызываем некую процедуру, называемую *менеджером перекрытий* (*overlay manager*). Эта процедура знает, какой модуль куда загружен, и при необходимости "подкачивает" то, что загружено не было. Перед каждой ссылкой на оверлейные данные мы должны выполнять аналогичную процедуру, что намного увеличивает и замедляет программу. Иногда такие действия возлагаются на программиста (Win16, Mac OS до версии 10 — подробнее управление памятью в этих системах описывается в разд. 4.4.1), иногда — на компилятор (handle pointer в Zortech C/C++ для

MS-DOS), но чаще всего с оверлейными данными вообще предпочитают не иметь дела. В таком случае оверлейным является только код.

В старых учебниках по программированию и руководствах по операционным системам уделялось много внимания тому, как распределять процедуры между оверлейными модулями. Действительно, загрузка модуля с диска представляет собой довольно длительный процесс, поэтому хотелось бы минимизировать ее. Для этого нужно, чтобы каждый оверлейный модуль был как можно более самодостаточным. Если это невозможно, стараются вынести процедуры, на которые ссылаются из нескольких оверлесов, в отдельный модуль, называемый *резидентной частью* или *резидентным ядром*. Это модуль, который всегда находится в памяти и не разделяет свои адреса ни с каким другим оверлеем. Естественно, оверлейный менеджер должен быть частью этого ядра.

Каждый оверлейный модуль может быть как абсолютным, так и перемещаемым. От этого несколько меняется устройство менеджера, но не более того. На архитектурах типа i80x86 можно делать оверлейные модули, каждый из которых адресуется относительно значения базового регистра CS и ссылается на данные, статически размещенные в памяти, относительно постоянного значения регистра DS. Такие модули можно загружать в память с любого адреса, может быть, даже в перемешку с данными. Именно так и ведут себя оверлейные менеджеры компиляторов Borland и Zortech для MS/DR DOS.

3.7. Сборка программ

Он был ловкий и весь такой собранный джентльмен, а одет — в самые лучшие и дорогие одежды; и все у него было подобрано и пригнано, даже части тела.

А. Тумуола

В предыдущем разделе шла речь о типах исполняемых модулей, но не говорилось ни слова о том, каким образом эти модули получаются. Вообще говоря, способ создания загружаемого модуля различен в различных ОС, но в настоящее время во всех широко распространенных системах этот процесс выглядит примерно одинаково. Это связано, прежде всего, с тем, что такие системы используют одни и те же языки программирования и правила межмодульного взаимодействия, в которых явно или неявно определяют логику раздельной компиляции и сборки.

В большинстве современных языков программирования программа состоит из отдельных слабо связанных модулей. Как правило, каждому такому модулю соответствует отдельный файл исходного текста. Эти файлы независимо обрабатываются языковым процессором (компилятором), и для каждого из

них генерируется отдельный файл, называемый *объектным модулем*. Затем запускается программа, называемая *редактором связей, компоновщиком* или *линкером* (*linker* — тот, кто связывает), которая формирует из заданных объектных модулей цельную программу.

Первый компоновщик

Первый компоновщик для компьютерных программ был разработан в 1950—51 годах Грэйс Хоппер, которая была в это время сотрудникой компании Remington Rand.

Хоппер начинала свою карьеру программиста на IBM ASSC (Harvard Mark I) во время Второй Мировой войны. Mark I использовал программную память с последовательным доступом на основе перфолент (первоначально гарвардской архитектурой называлось именно это, а не раздельные адресные пространства команд и данных). Для того чтобы реализовать на такой машине цикл с известным числом итераций (например, разностную схему для решения системы дифференциальных уравнений), необходимо было многократно повторить тело цикла в программе.

Важным достоинством этой архитектуры был тот факт, что программы не имели в коде адресных ссылок на код. Поэтому все программы для Mark I были позиционно независимыми. Для переноса куска кода из одного места в другое можно было воспользоваться просто ножницами и kleem.

В действительности, считывание программы компьютером происходило на довольно высокой скорости, поэтому лента подвергалась довольно высоким механическим нагрузкам. Поэтому в Mark I нельзя было заправлять перфоленту, склеенную из кусков.

Процедура сборки программы для Mark I напоминала процедуру монтажа пленочного кинофильма. Программист использовал станок, который мог дублировать перфоленту. На вход этого станка подавались куски перфоленты с исходными модулями, а станок дублировал их на новую ленту. Поскольку чтение ленты станком происходило с гораздо меньшей скоростью, чем ее протягивал компьютер, исходная лента вполне могла быть склеенной. Если необходимо было сделать цикл, исходный кусок ленты замыкался в кольцо (по одной из легенд, отсюда происходит одно из названий циклов в англоязычной терминологии — *loop*, кольцо).

Присоединившись к команде программистов Mark I, Хоппер внесла в процесс разработки ряд полезных усовершенствований. Так, по ее инициативе была создана библиотека переиспользуемых подпрограмм. Поэтому Хоппер часто приписывают изобретение подпрограммы.

Когда Хоппер в 1949 году присоединилась к Эккерту и Мочли в их работе над компьютером UNIVAC, она быстро обнаружила, что приемы повторного использования кода, пригодные для Mark I, совершенно не годятся для машин с адресуемой программной памятью. Некоторое время она пыталась организовать работу по ручному пересчету адресов в библиотечных программах, но все заинтересованные быстро пришли к выводу, что так жить нельзя. В результате, Хоппер написала одну из первых в мире инструментальных программ, предназначенных не для решения прикладной задачи, а для упрощения разработки других программ.

Раннее связывание предполагает однократное разрешение всех символов и допускает отбрасывание необходимой для этого информации (таблиц символов и перекрестных ссылок); впрочем, при относительной загрузке мы вынуждены сохранять значительную часть таблицы перекрестных ссылок под названием таблицы перемещений.

Объектный модуль отчасти похож по структуре на перемещаемый загрузочный модуль. Дело в том, что сборку программы из нескольких модулей можно уподобить загрузке в память нескольких программ. При этом возникает та же задача перенастройки адресных ссылок, что и при загрузке относительно-загрузочного файла (рис. 3.8). Поэтому объектный модуль должен в той или иной форме содержать таблицу перемещений. Можно, конечно, потребовать, чтобы весь модуль был позиционно-независимым, но это, как говорилось ранее, накладывает очень жесткие ограничения на стиль программирования, нередко требует выделения регистров процессора, а на некоторых архитектурах и просто невозможно.

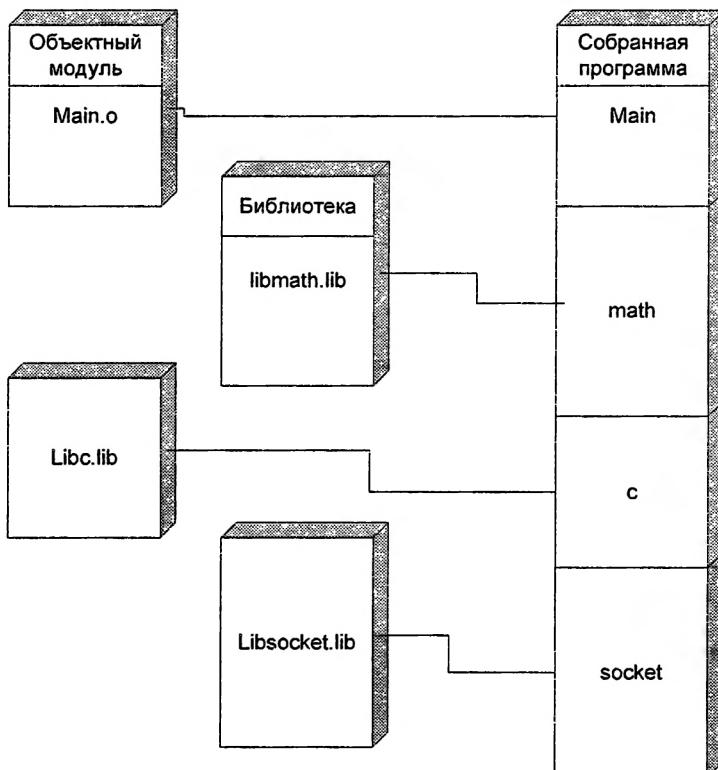


Рис. 3.8. Сборка программы

Кроме ссылок на собственные метки, объектный модуль имеет право ссылаться на символы, определенные в других модулях. Типичный пример такой ссылки — вызов функции, которая определена в другом файле исходного текста (рис. 3.9 и 3.10).

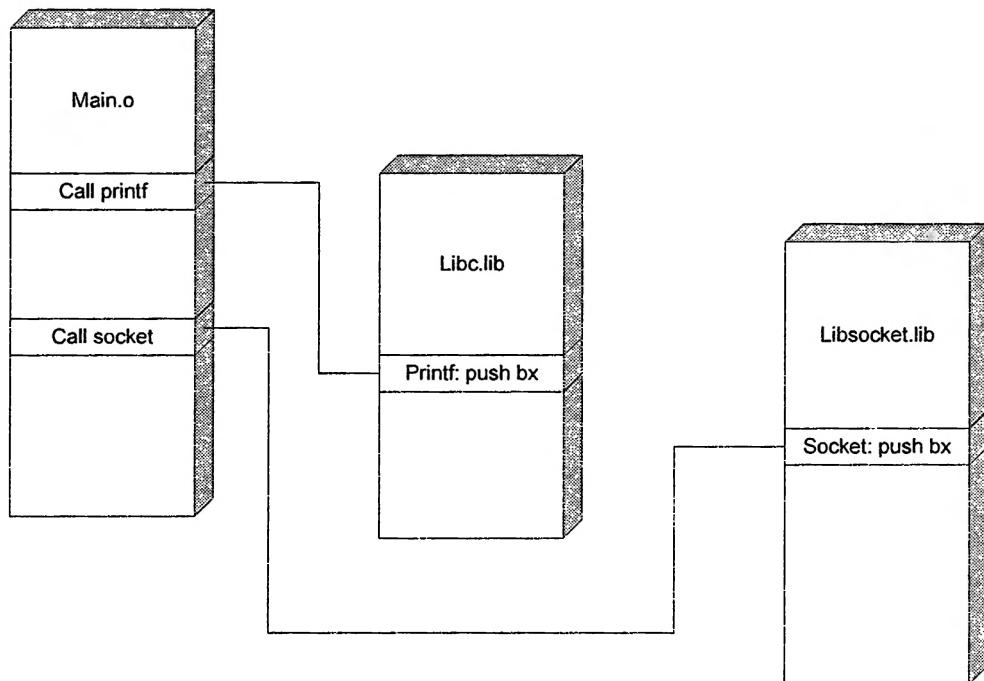


Рис. 3.9. Разрешение внешних ссылок (объектный модуль)

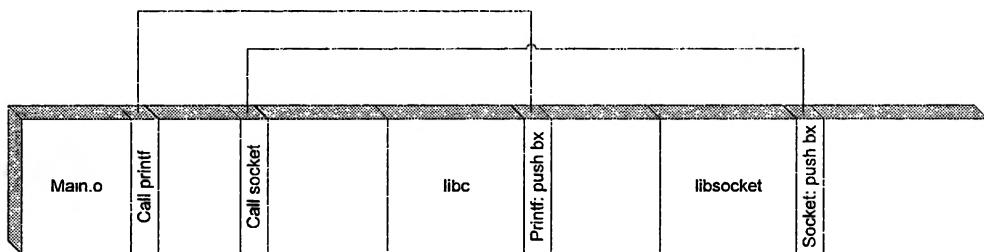


Рис. 3.10. Разрешение внешних ссылок (собранная программа)

Для разрешения внешних ссылок мы должны создать две таблицы: в одной перечислены внешние объекты, на которые ссылается модуль, в другой — объекты, определенные внутри модуля, на которые можно ссылаться извне.

Обычно с каждым таким объектом ассоциировано имя, называемое *глобальным символом*. Как правило, это имя совпадает с именем соответствующей функции или переменной в исходном языке, однако бывают и исключения. Так, C++ поддерживает перегруженные функции и методы — одноименные функции, различающиеся по количеству и типам параметров, поэтому компиляторы C++ кодируют в имени символа набор параметров функции или метода, а для методов также имя класса, к которому этот метод принадлежит. Это приводит к значительному увеличению длины имени символа, так что C++ потребовал переделки редакторов связей и форматов объектных модулей.

Для каждой ссылки на внешний символ мы должны уметь определить, является эта ссылка абсолютной или относительной, либо это вообще должна быть разность или сумма двух или даже более адресов, и т. д. Для определения объекта, с другой стороны, мы должны уметь указать, что это абсолютный или перемещаемый символ, либо что он равен другому символу плюс заданное смещение, и т. д.

Кроме того, в объектных файлах может содержаться отладочная информация, формат которой может быть очень сложным. Следовательно, объектный файл представляет собой довольно сложную и рыхлую структуру. Размер собранной программы может оказаться в два или три раза меньше суммы длин объектных модулей.

Типичный объектный модуль содержит следующие структуры данных:

- таблицу перемещений, т. е. таблицу ссылок на перемещаемые объекты внутри модуля;
- таблицу ссылок на внешние объекты. Иногда это называется таблицей или списком импорта;
- таблицу определенных в этом модуле объектов, на которые можно ссылаться из других модулей. В некоторых случаях ее называют *списком экспорта*. Иногда таблицы экспорта и импорта объединяют и называют *таблицей глобальных символов*. В этом случае для каждого символа приходится указывать, определен он в данном модуле или нет, а если определен, то как;
- различную служебную информацию, такую как имя модуля, программу, которая его создала (например, строка "gcc compiled");
- отладочную информацию, которая, в свою очередь, обычно содержит перемещаемые и внешние адресные ссылки (см. разд. 2.10.1);
- Собственно код и данные модуля.

Как правило, код и данные разбиты на именованные секции. В ассемблерах с синтаксисом Intel для x86 (masm, tasm) такие секции называются сегментами,

в ассемблерах DEC и ассемблерах с синтаксисом AT&T — *программными секциями* (см. разд. П2.4.3). В готовой программе все участки кода или данных, описанные в разных модулях, но принадлежащие к одной секции, собираются вместе. Например, в системах семейства Unix программы, написанные на языке C, состоят из минимум трех программных секций:

- `text` — исполняемый код (современные компиляторы иногда помещают в эту секцию и данные, описанные как `const`);
- `data` — статически инициализированные данные;
- `bss` — неинициализированные данные.

В качестве упражнения читателю предлагается найти эти секции в примере 3.7 и других примерах ассемблерного кода, приводимых в этой книге.

Некоторые форматы объектных модулей, в частности ELF (Executable and Linking Format — формат исполняемых и собираемых [модулей], используемый современными системами семейства Unix), предоставляют особый тип глобального символа — *слабый (weak) символ* (пример 3.8). При сборке программы компоновщик не выдает сообщения об ошибке, если обнаруживает два различных определения символа, при условии, что одно из определений является слабым — таким образом, слабый символ может быть легко переопределен при необходимости. Особенно полезен этот тип при помещении объектного модуля в библиотеку.

Пример 3.8. Структуры данных объектного модуля ELF (цитируется по elf.h из поставки Linux 2.2.16, перевод комментариев автора)

```
/* Заголовок файла ELF. Находится в начале каждого файла ELF. */

#define EI_NIDENT (16)

typedef struct
{
    unsigned char e_ident[EI_NIDENT]; /* Магическое число и другая
информация */
    Elf32_Half e_type;           /* Тип объектного файла */
    Elf32_Half e_machine;        /* Архитектура */
    Elf32_Word e_version;        /* Версия объектного файла */
    Elf32_Addr e_entry;          /* Виртуальный адрес точки входа */
    Elf32_Off e_phoff;           /* Смещение таблицы заголовков программы */
                                /* в файле */
    Elf32_Off e_shoff;           /* Смещение таблицы заголовков секций в файле */
    Elf32_Word e_flags;          /* Процессорно-зависимые флаги */
```

```

Elf32_Half e_ehsize;      /* Размер заголовка ELF в байтах */
Elf32_Half e_phentsize; /* Размер элемента */
                           /* таблицы заголовка программы */
Elf32_Half e_phnum;     /* Счетчик элементов таблицы заголовка программы */
Elf32_Half e_shentsize; /* Размер элемента таблицы заголовков секций */
Elf32_Half e_shnum;    /* Счетчик элементов таблицы заголовков программ */
Elf32_Half e_shstrndx; /* Индекс таблицы имен секций
                           /* в таблице заголовков секций */

} Elf32_Ehdr;

/* Поля в массиве e_indent. Макросы EI_* суть индексы в этом массиве.
Макросы, следующие за каждым определением EI_*, суть значения, которые
соответствующий байт может принимать. */

#define EI_MAG0 0      /* Индекс нулевого байта сигнатуры1 */
#define ELF MAG0 0x7f /* Значение нулевого байта сигнатуры */

#define EI_MAG1 1      /* Индекс первого байта сигнатуры */
#define ELF MAG1 'E'   /* Значение первого байта сигнатуры */

#define EI_MAG2 2      /* Индекс второго байта сигнатуры */
#define ELF MAG2 'L'   /* Значение второго байта сигнатуры */

#define EI_MAG3 3      /* Индекс третьего байта сигнатуры */
#define ELF MAG3 'F'   /* Значение третьего байта сигнатуры */

/* Объединение идентификационных байтов, для сравнения по словам */
#define ELF MAG "\177ELF"
#define SELFMAG 4

#define EI_CLASS 4      /* Индекс байта, указывающего класс файла */
#define ELFCLASSNONE 0  /* Не определено */
#define ELFCLASS32 1    /* 32-разрядные объекты */
#define ELFCLASS64 2    /* 64-разрядные объекты */
#define ELFCLASSNUM 3

#define EI_DATA 5       /* Индекс байта кодировки данных */
#define ELF DATA NONE 0 /* Не определена кодировка данных */

```

¹ В данном случае — это "магическое число", код, размещаемый в определенном месте (обычно в начале) файла и подтверждающий, что это файл данного формата.

```
#define ELFDATA2LSB 1 /* Двоичные дополнительные, младший байт первый */
#define ELFDATA2MSB 2 /* Двоичные дополнительные, старший байт первый */
#define ELFDATANUM 3

#define EI_VERSION 6 /* Индекс байта версии файла */
/* Значение должно быть EV_CURRENT */

#define EI_OSABI 7 /* идентификатор OS ABI */
#define ELFOSABI_SYSV 0 /* UNIX System V ABI */
#define ELFOSABI_HPUX 1 /* HP-UX */
#define ELFOSABI_ARM 97 /* ARM */
#define ELFOSABI_STANDALONE 255 /* Самостоятельное (встраиваемое)
приложение */

#define EI_ABIVERSION 8 /* версия ABI */

#define EI_PAD 9 /* Индекс байтов выравнивания */

/* Допустимые значения для e_type (тип объектного файла). */

#define ET_NONE 0 /* Не указан тип */
#define ET_REL 1 /* Перемещаемый файл */
#define ET_EXEC 2 /* Исполнимый файл */
#define ET_DYN 3 /* Разделяемый объектный файл */
#define ET_CORE 4 /* Образ задачи */
#define ET_NUM 5 /* Количество определенных типов */
#define ET_LOPROC 0xff00 /* Специфичный для процессора */
#define ET_HIPROC 0xffff /* Специфичный для процессора */

/* Допустимые значения для e_machine (архитектура). */

#define EM_NONE 0 /* Не указана машина */
#define EM_M32 1 /* AT&T WE 32100 */
#define EM_SPARC 2 /* SUN SPARC */
#define EM_386 3 /* Intel 80386 */
#define EM_68K 4 /* Motorola m68k family */
#define EM_88K 5 /* Motorola m88k family */
#define EM_486 6 /* Intel 80486 */
#define EM_860 7 /* Intel 80860 */
#define EM_MIPS 8 /* MIPS R3000 big-endian */
#define EM_S370 9 /* Amdahl */
#define EM_MIPS_RS4_BE 10 /* MIPS R4000 big-endian */
#define EM_RS6000 11 /* RS6000 */
```

```
#define EM_PARISC 15 /* HPPA */
#define EM_nCUBE 16 /* nCUBE */
#define EM_VPP500 17 /* Fujitsu VPP500 */
#define EM_SPARC32PLUS 18 /* Sun's "v8plus" */
#define EM_960 19 /* Intel 80960 */
#define EM_PPC 20 /* PowerPC */

#define EM_V800 36 /* NEC V800 series */
#define EM_FR20 37 /* Fujitsu FR20 */
#define EM_RH32 38 /* TRW RH32 */
#define EM_MMA 39 /* Fujitsu MMA */
#define EM_ARM 40 /* ARM */
#define EM_FAKE_ALPHA 41 /* Digital Alpha */
#define EM_SH 42 /* Hitachi SH */
#define EM_SPARCV9 43 /* SPARC v9 64-bit */
#define EM_TRICORE 44 /* Siemens Tricore */
#define EM_ARC 45 /* Argonaut RISC Core */
#define EM_H8_300 46 /* Hitachi H8/300 */
#define EM_H8_300H 47 /* Hitachi H8/300H */
#define EM_H8S 48 /* Hitachi H8S */
#define EM_H8_500 49 /* Hitachi H8/500 */
#define EM_IA_64 50 /* Intel Merced */
#define EM_MIPS_X 51 /* Stanford MIPS-X */
#define EM_COLDFIRE 52 /* Motorola Coldfire */
#define EM_68HC12 53 /* Motorola M68HC12 */
#define EM_NUM 54
```

/* Если необходимо выделить неофициальное значение для EM_*, пожалуйста, выделите большие случайные числа (0x8523, 0xa7f2, etc.), чтобы уменьшить вероятность пересечения с официальными или не-GNU неофициальными значениями. */

```
#define EM_ALPHA 0x9026

/* Допустимые значения для e_version (версия). */

#define EV_NONE 0 /* Недопустимая версия ELF */
#define EV_CURRENT 1 /* Текущая версия */
#define EV_NUM 2
```

```
/* Элемент таблицы символов. */

typedef struct
{
    Elf32_Word st_name;      /* Имя символа (индекс в таблице строк) */
    Elf32_Addr st_value;    /* Значение символа */
    Elf32_Word st_size;     /* Размер символа */
    unsigned char st_info;   /* Тип и привязка символа */
    unsigned char st_other;  /* Значение не определено, 0 */
    Elf32_Section st_shndx; /* Индекс секции */
} Elf32_Sym;

/* Секция syminfo, если присутствует, содержит дополнительную информацию
о каждом динамическом символе. */

typedef struct
{
    Elf32_Half si_boundto; /* Прямая привязка, символ, к которому привязан */
    Elf32_Half si_flags;   /* Флаги символа */
} Elf32_Syminfo;

/* Допустимые значения для si_boundto. */
#define SYMINFO_BT_SELF 0xffff      /* Символ привязан к себе */
#define SYMINFO_BT_PARENT 0xffffe   /* Символ привязан к родителю */
#define SYMINFO_BT_LOWRESERVE 0xff00 /* Начало зарезервированных записей */

/* Возможные битовые маски для si_flags. */
#define SYMINFO_FLG_DIRECT 0x0001   /* Прямо привязываемый символ */
#define SYMINFO_FLG_PASSTHRU 0x0002 /* Промежуточный символ для
транслятора */
#define SYMINFO_FLG_COPY 0x0004    /* Символ предназначен для
перемещения копированием */
#define SYMINFO_FLG_LAZYLOAD 0x0008 /* Символ привязан к объекту
с отложенной загрузкой */

/* Значения версии Syminfo. */
#define SYMINFO_NONE 0
#define SYMINFO_CURRENT 1
#define SYMINFO_NUM 2

/* Как извлекать информацию из поля st_info. */

#define ELF32_ST_BIND(val) (((unsigned char) (val)) >> 4)
```

```

#define ELF32_ST_TYPE(val) ((val) & 0xf)
#define ELF32_ST_INFO(bind, type) (((bind) << 4) + ((type) & 0xf))
/* Допустимые значения для под поля ST_BIND поля st_info
(привязка символов). */

#define STB_LOCAL 0 /* Локальный символ */
#define STB_GLOBAL 1 /* Глобальный символ */
#define STB_WEAK 2 /* Слабый символ */
#define STB_NUM 3 /* Кол-во определенных типов. */
#define STB_LOOS 10 /* Начало ОС-зависимых значений */
#define STB_HIOS 12 /* Конец ОС-зависимых значений */
#define STB_LOPROC 13 /* Начало процессорно-зависимых значений */
#define STB_HIPROC 15 /* Конец процессорно-зависимых значений */

/* Допустимые значения для под поля ST_TYPE поля st_info (тип символа). */

#define STT_NOTYPE 0 /* Не указан */
#define STT_OBJECT 1 /* Символ -- объект данных */
#define STT_FUNC 2 /* Символ -- объект кода */
#define STT_SECTION 3 /* Символ связан с секцией */
#define STT_FILE 4 /* Имя символа -- имя файла */
#define STT_NUM 5 /* Кол-во определенных типов */
#define STT_LOOS 11 /* Начало ОС-зависимых значений */
#define STT_HIOS 12 /* Конец ОС-зависимых значений */
#define STT_LOPROC 13 /* Начало процессорно-зависимых значений */
#define STT_HIPROC 15 /* Конец процессорно-зависимых значений */

/* Индексы таблицы символов размещены в группах и цепочках хэша в секции
хэш-таблицы символов. Это специальное значение индекса указывает на
конец цепочки, и означает, что в этой группе более нет символов. */

#define STN_UNDEF 0 /* Конец таблицы. */

/* Элемент таблицы перемещений без добавочного значения
(в секциях типа SHT_REL). */

typedef struct
{
    Elf32_Addr r_offset; /* Адрес */
    Elf32_Word r_info; /* Тип перемещения и индекс символа */
} Elf32_Rel;

```

```
/* Элемент таблицы перемещений с добавочным значением
(в секциях типа SHT_REL). */

typedef struct
{
    Elf32_Addr r_offset; /* Адрес */
    Elf32_Word r_info;   /* Тип перемещения и индекс символа */
    Elf32_Sword r_addend; /* Добавочное значение */
} Elf32_Rela;

/* Как извлекать информацию из поля r_info. */

#define ELF32_R_SYM(val) ((val) >> 8)
#define ELF32_R_TYPE(val) ((val) & 0xff)
#define ELF32_R_INFO(sym, type) (((sym) << 8) + ((type) & 0xff))

/* Типы перемещений для i386 (формулы взяты из
[docs.sun.com 816-0559-10] – авт.)
```

А – добавочное значение, используемое при вычислении значения перемещаемого поля.

В – базовый адрес, начиная с которого разделяемый объект загружается в память при исполнении [программы]. Обычно разделяемый объект строится с базовым виртуальным адресом, равным 0, но адрес при исполнении иной.

Г – смещение записи в глобальной таблице смещений, где адрес перемещаемого символа находится во время исполнения.

GOT – адрес глобальной таблицы смещений.

Л – местоположение (смещение в секции или адрес) записи символа в процедурной таблице связывания (PLT). PLT перенаправляет вызов функции по настоящему адресу. Редактор связей создает начальную таблицу, а редактор связей времени исполнения модифицирует записи во время исполнения.

Р – местоположение (смещение в секции или адрес) перемещаемого элемента памяти (вычисляется с использованием r_offset).

С – значение символа, индекс которого находится в элементе таблицы перемещений. */

#define R_386_NONE 0 /* Не перемещать */
#define R_386_32 1 /* Прямое 32-разрядное – S + A */
#define R_386_PC32 2 /* 32-разрядное относительно PC – S + A – P */

```

#define R_386_GOT32 3      /* 32-разрядный элемент GOT - G + A */
#define R_386_PLT32 4      /* 32-разрядный адрес PLT - L + A - P */
#define R_386_COPY 5       /* Копировать символ при исполнении */
#define R_386_GLOB_DAT 6   /* Создать запись GOT - S */
#define R_386 JMP_SLOT 7   /* Создать запись PLT - S */
#define R_386_RELATIVE 8   /* Сдвинуть относительно базы программы -
                           B + A */
#define R_386_GOTOFF 9     /* 32-разрядное смещение GOT - S + A - GOT */
#define R_386_GOTPC 10     /* 32-разрядное смещение GOT относительно
                           PC - S + A - GOT */

/* Должна быть последняя запись. */
#define R_386_NUM 11

```

3.8. Объектные библиотеки

Крупные программы часто состоят из сотен и тысяч отдельных модулей. Кроме того, существуют различные пакеты подпрограмм, также состоящие из большого количества модулей. Один из таких пакетов используется практически в любой программе на языке высокого уровня — это так называемая стандартная библиотека. Для решения проблем, возникающих при поддержании порядка в наборах из большого количества объектных модулей, еще на заре вычислительной техники были придуманы *библиотеки объектных модулей*.

Библиотека, как правило, представляет собой последовательный файл, состоящий из заголовка, за которым последовательно располагаются объектные модули (рис. 3.11). В заголовке содержится следующая информация:

- список всех объектных модулей, со смещением каждого модуля от начала библиотеки. Смещение нужно для того, чтобы можно было легко найти требуемый модуль;
- список всех глобальных символов, определенных в каждом из модулей, с указанием, в каком именно модуле он был определен.

Линкер (рис. 3.12) обычно собирает в программу все объектные модули, которые были ему заданы в командной строке, даже если на этот модуль не было ни одной ссылки. С библиотечными модулями он ведет себя несколько иначе.

Встретив ссылку на глобальный символ, компоновщик ищет определение этого символа во всех модулях, которые ему были заданы. Если там такого символа нет, то линкер ищет этот символ в заголовке библиотеки. Если его

нет и там, компоновщик сообщает: "Не определен символ SYMBOL", — и завершает работу. Некоторые редакторы связей, правда, могут продолжить работу и даже собрать загружаемый модуль, но, как правило, таким модулем пользоваться нельзя, т. к. в нем содержится ссылка на некорректный адрес. Если же определение символа в библиотеке есть, компоновщик "вытаскивает" соответствующий модуль и дальше работает так, будто этот модуль был задан ему наравне с остальными объектными файлами. Этот процесс повторяется до тех пор, пока не будут разрешены все глобальные ссылки, в том числе и те, которые возникли в библиотечных модулях, или пока не будет обнаружен неопределенный символ. Благодаря такому алгоритму в программу включаются только те модули из библиотеки, которые нужны.

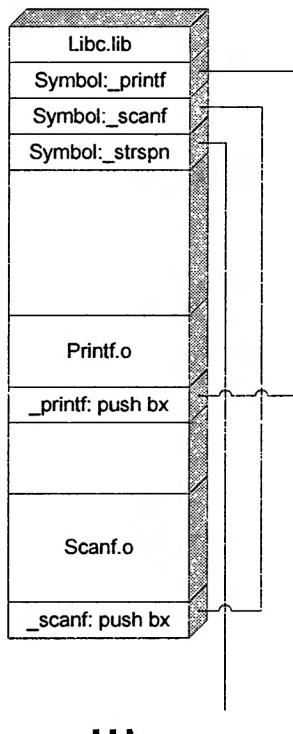


Рис. 3.11. Объектная библиотека

В системах семейства Unix библиотеки такой структуры называются *архивными библиотеками*, чтобы отличить их от разделяемых библиотек, которые рассматриваются в *разд. 3.10 и 5.4*.

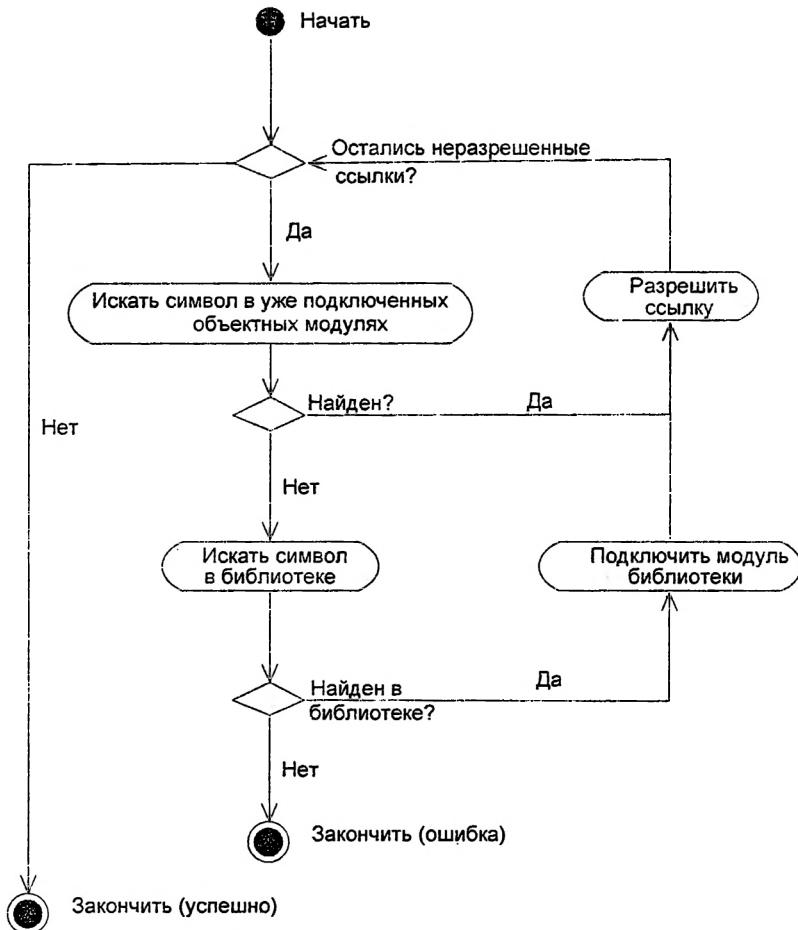


Рис. 3.12. Блок-схема работы редактора связей

3.9. Сборка в момент загрузки

...как только они вошли в Бесконечный Лес, собранный лягушельмен стал разбираться на части и принялся выплачивать арендные деньги. Сначала он отправился к ногозанимодавцам и пришел туда, где нанял левую ногу; он отдал ее владельцу, и заплатил за аренду, и запрыгал к хозяину правой ноги; когда он вернул ее и полностью расплатился, то перевернулся вниз головой и поскакал на руках.

А. Тутуола

Как мы видели в предыдущем разделе, объектные модули и библиотеки содержат достаточно информации, чтобы собирать программу не только заранее.

нее, но и непосредственно в момент загрузки. Этот способ, безусловно, требует больших затрат процессорного времени, чем загрузка заранее собранного кода, но дает и некоторые преимущества.

Главное преимущество состоит в том, что, если мы загружаем несколько программ, использующих одну и ту же библиотеку, мы можем настроить их на работу с одной копией кода библиотеки, таким образом, сэкономив память. Разделение кода привлекательно и с функциональной точки зрения, поэтому сборка в момент загрузки находит широкое применение в самых разнообразных ситуациях.

Примером такой сборки является широко используемая в Windows всех версий и OS/2 технология DLL (на самом деле, DLL обеспечивают сборку не только в момент загрузки, но и после нее — возможность подключить дополнительный модуль к уже загруженной программе), которая будет более подробно обсуждаться далее. В качестве других примеров можно привести Novell Netware, OS-9, VxWorks и т. д. Впрочем, если мы говорим о системах, предназначенных для использования во встроенных приложениях (той же VxWorks), вопрос о том, является ли сборка перед прошивкой в ПЗУ сборкой в момент загрузки или сборкой заранее, носит схоластический характер.

Сборка при загрузке замедляет процесс загрузки программы (впрочем, для современных процессоров это замедление вряд ли имеет большое значение), но упрощает, с одной стороны, разделение кода, а с другой стороны — разработку программ. Действительно, из классического цикла внесения изменения в программу — редактирование текста — перекомпиляция — пересборка — перезагрузка (программы, не обязательно всей системы) — выпадает целая фаза. В случае большой программы это может быть длительная фаза. В случае Novell Netware решающим оказывается первое преимущество (рис. 3.13), в случае систем реального времени одинаково важны оба.

В большинстве современных ОС, в действительности, сборка в момент загрузки происходит не из объектных модулей, а из предварительно собранных *разделяемых библиотек*. Такие библиотеки отличаются от обсуждавшихся в разд. 3.8, во-первых, тем, что из них невозможно извлечь отдельный модуль: все межмодульные ссылки внутри такой библиотеки разрешены, и ее необходимо всегда загружать как целое; и, во-вторых, тем, что список символов, экспортруемых такой библиотекой, не является объединением списков экспорта составляющих ее объектных модулей. При сборке такой библиотеки необходимо указать, какие из символов будут экспортироваться. Некоторые редакторы связей позволяют на этом этапе создавать дополнительные символы.

Некоторые системы команд поддерживают динамически пересобираемые программы, у которых вся настройка модуля вынесена в отдельную таблицу.

В этом случае модуль может быть подключен одновременно к нескольким программам, использовать одновременно разные копии сегмента данных, и каждая используемая копия модуля при этом даже не будет подозревать о существовании других. Примером такой архитектуры является Pascal-система *Lilith*, разработанная Н. Виртом, и ее наследники *Kronos/N9000*.

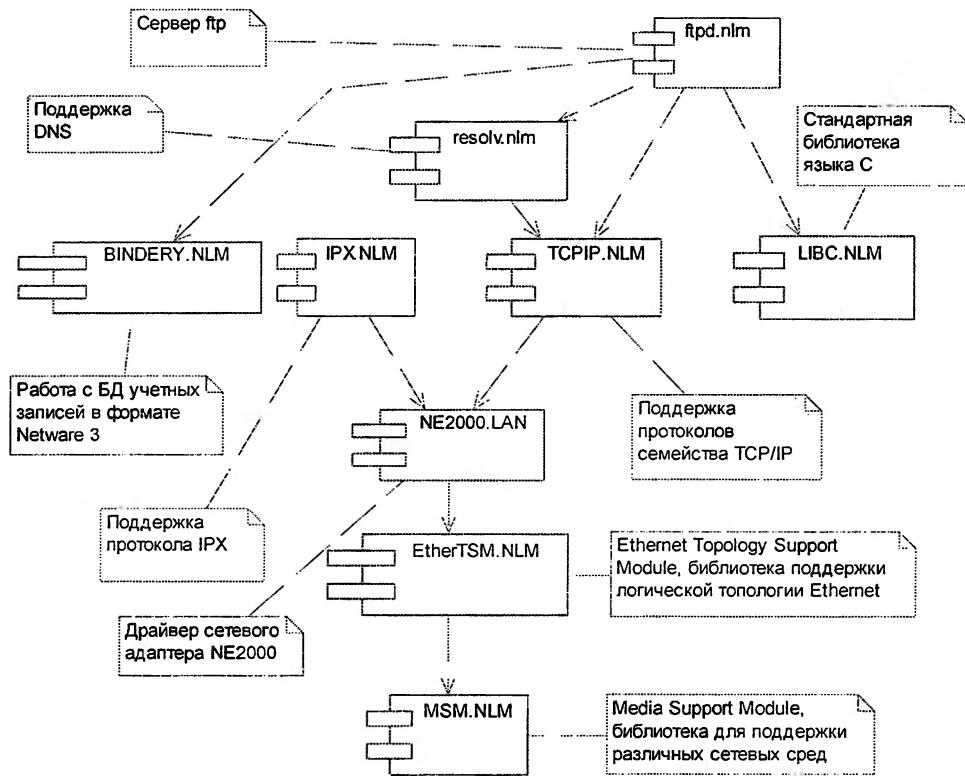


Рис. 3.13. Фрагмент структуры взаимозависимостей между NLM (Netware Loadable Module) сервера Netware 4.11

Программные модули в N9000

В этих архитектурах каждый объектный модуль соответствует одному модулю в смысле языка высокого уровня Oberon (или NIL — N9000 Instrumental Language). Далее мы будем описывать архитектуру системы N9000, поскольку я с ней лучше знаком.

Модуль может иметь не более 256 процедур, не более 256 переменных и ссылаться не более чем на 256 других модулей. Код модуля является позиционно-независимым. Данные модуля собраны в отдельный сегмент, и для каждой используемой копии модуля, т. е. для каждой программы, которая этот модуль использует, создается своя копия сегмента данных. В начале сегмента находится

таблица переменных. Строки этой таблицы содержат либо значения — для скалярных переменных, таких как целое число или указатель, либо адреса в сегменте данных. Кроме того, в сегменте данных есть ссылка на сегмент кода. Этот сегмент кода содержит таблицу адресов точек входа всех определенных в нем функций (рис. 3.14).

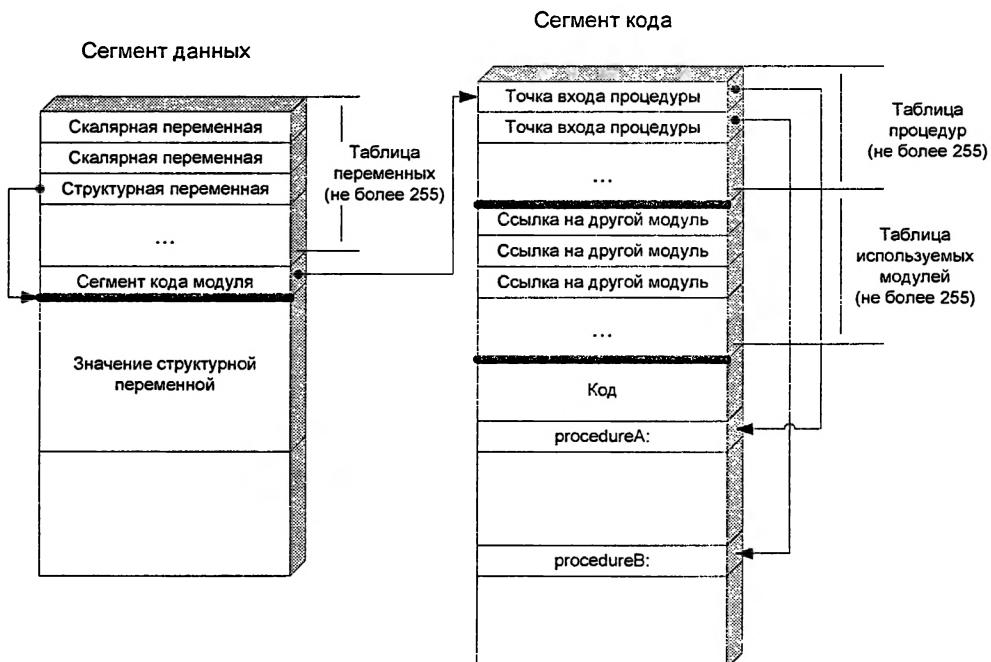


Рис. 3.14. Модуль N9000

Ссылки на все внешние модули собраны в таблицу, которая также содержится в сегменте данных. Внешний модуль определяется началом его сегмента данных.

Все ссылки на объекты в данном модуле осуществляются через индекс в соответствующей таблице. Ссылки на внешние модули имеют вид индекс модуля: индекс объекта.

Сегмент данных не может содержать никаких статически инициализированных данных. Инициализация производится специальной процедурой, которая вызывается при каждом новом использовании модуля. Все эти свойства реализованы в системе команд, потому накладные расходы относительно невелики. Точнее, они невелики по сравнению с Intel 80286, но уже великоваты по сравнению с i386, а по сравнению с современными RISC-процессорами или системами типа транспьютера (Transputer) они становятся недопустимыми. Впрочем, в разд. 5.4 мы увидим, как подобная структура используется и на "обычных" процессорах.

Видно, что в системе может существовать несколько программ, обращающихся к одним и тем же модулям и использующих одну и ту же копию кода модуля.

Проблем с абсолютной/относительной загрузкой вообще не возникает. Операционная система ТС для Н9000 была (я не уверен, существует ли в настоящее время хотя бы одна работоспособная машина этой архитектуры) основана на сборке программ в момент загрузки. В системе имелась специальная команда `load` — "загрузить все модули, используемые программой, и разместить для них сегменты данных, но саму программу не запускать". В памяти могло находиться одновременно несколько программ; при этом модули, используемые несколькими из них, загружались в одном экземпляре. Это значительно ускоряло работу. Например, можно было загрузить в память текстовый редактор, и запуск его занимал бы доли секунды вместо десятков секунд, которые нужны для загрузки с жесткого диска фирмы ИЗОТ.

Любопытно, что когда началась реализация системы программирования на языке С для этой машины, по ряду причин было решено не связываться с динамической сборкой, а собирать обычные перемещаемые загрузочные модули.

На практике, подобная архитектура более характерна для байт-кодов — пре-компилированных представлений программы, предназначенных для дальнейшей обработки интерпретатором — Java Virtual Machine, интерпретатором Smalltalk и т. д., чем для аппаратно реализованных систем команд. В таких системах команд порой используются и более экстравагантные решения.

Архитектура AS/400

Система команд AS/400 (сервер баз данных среднего уровня, производимый IBM) представляет собой машинно-независимый байт-код. При загрузке программы этот байт-код компилируется в бинарный код "реального" процессора подобно тому, как это делается в большинстве современных реализаций Java Virtual Machine (JVM). Точнее, наоборот, успех AS/400 был одним из важных факторов, которые подвигли фирму Sun на разработку Java, поэтому правильнее говорить, что современные JVM основаны на том же принципе компиляции при загрузке, что и AS/400.

Это решение обеспечивает невысокую стоимость аппаратуры (современные AS/400 основаны на микропроцессорах архитектуры Power PC; их более высокая по сравнению с машинами, основанными на процессорах x86, цена обусловлена более производительными системной шиной и периферией), высокую производительность и возможность заменять архитектуру "реального" процессора без перекомпиляции пользовательского программного обеспечения. За время выпуска машин этой серии такая замена происходила дважды.

С другой стороны, отсутствие необходимости думать о том, как та или иная возможность может быть реализована аппаратно, позволила принимать весьма авангардистские решения, на которые не решался никто из разработчиков аппаратно реализованных CISC-архитектур, таких как VAX, Nova Eclipse и даже апофеоза CISC, Intel 432.

AS/400 имеет единое адресное пространство в том смысле, что адресуемыми объектами являются не только сегменты кода и скалярных данных, но и объекты реляционной СУБД, такие как таблицы, индексы, курсоры и т. д.

Фактически, адресации подлежит вся память системы — как оперативная, так и дисковая. Адрес имеет два представления: его сегментная часть может хранить имя адресуемого объекта (в контексте этой главы это можно уподобить неразрешенной внешней ссылке) или собственно адрес, 64-битовое бинарное значе-

ние. Перед тем как обратиться к объекту, адрес-имя надо преобразовать в бинарный формат, для чего существуют специальные команды [redbooks.ibm.com sg242222].

Механизм этого преобразования выполняет работу и файловой системы, и редактора связей, в том смысле, что и файловый доступ, и сборка программы содержат важную фазу преобразования имен (соответственно, имен файлов и имен внешних символов) в адреса, по которым можно осуществлять доступ.

3.10. Динамические библиотеки

В Windows и OS/2 используется комбинация предварительной сборки программ и сборки в момент загрузки. Исполняемый модуль в этих системах собирается из объектных модулей, но содержит неразрешенные ссылки на другие модули, называемые DLL (Dynamically Loadable Library, динамически загружаемая библиотека или, как это расшифровывают в более современных документах, Dynamic-Link Library, динамически связываемая библиотека). Фактически, каждый модуль в этих системах обязан содержать хотя бы одну ссылку на DLL, потому что интерфейс к системным вызовам в этих ОС также реализован в виде DLL.

DLL представляют собой библиотеки в том смысле, что обычно они также собираются из нескольких объектных модулей. Но, в отличие от архивных библиотек, из DLL нельзя извлечь отдельный модуль, при присоединении библиотеки к программе она присоединяется и загружается целиком. Таким образом, динамический сборщик оперирует не большим числом независимых объектных модулей, а относительно небольшим количеством библиотек.

Главное отличие DLL от поддерживаемых многими другими ОС разделяемых библиотек состоит в том, что модуль (как основной, так и библиотечный) по собственному желанию может выбирать различные библиотеки, подгружая их уже после своей собственной загрузки. При этом нет даже строгого ограничения на совместимость этих библиотек по вызовам (две библиотеки совместимы по вызовам, если они имеют одинаковые точки входа с одинаковой семантикой): загрузчик предоставляет возможность просмотреть список глобальных символов, определенных в библиотеке, и получить указатель на каждый символ, обратившись к нему по имени (впрочем, количество и типы параметров или тип переменной, а тем более их семантику, загрузчик не сообщает — эту информацию надо получать из других источников, например из списка зарегистрированных в системе объектов СОМ). Таким образом, DLL позволяют использовать как раннее (во время загрузки), так и позднее (во время исполнения) связывание.

Особенно удобна возможность вызывать любую функцию по имени при обращении к внешним модулям из интерпретируемых языков — особенно в

ситуации, когда интерпретатор языка с поздним связыванием (Basic или Perl) написан на языке с ранним связыванием (C/C++).

В примере 3.9 для подключения внешних библиотек (в данном случае это стандартная библиотека RexxUtil и библиотека доступа к сетевым сервисам rxSock) применяются две процедуры: сначала RxFuncAdd с тремя параметрами: имя символа REXX, который будет использоваться для обращения к вызываемой функции, имя DLL и имя символа в этой DLL, а потом специальная функция, предоставляемая модулем (SysLoadFunc и SockLoadFunc соответственно), которая регистрирует в интерпретаторе REXX остальные функции модуля.

Пример 3.9. Пример использования динамической библиотеки
(здесь — REXX Socket) в интерпретируемом языке

```
*****  
ПРОСТОЙ HTTP клиент на REXX  
Dmitry Maximovich 2:5030/544.60 aka maxim@pabl.ru  
*****  
PARSE VALUE ARG(1) WITH A1 A2  
  
IF A1 = '' THEN  
DO  
  SAY 'USAGE: wwwget hostname[/path] [port]'  
  EXIT  
END  
ELSE  
DO  
  PARSE VALUE A1 WITH B1'/'B2  
  sServer = B1  
  IF B2 = '' THEN  
    DO  
      sRequest = 'GET / HTTP 1.0'||"0D0A0D0A"x  
      SAY 'Requesting /'  
    END  
  ELSE  
    DO  
      sRequest = 'GET /'||B2||' HTTP 1.0'||"0D0A0D0A"x  
      SAY 'Requesting /'||B2  
    END  
  END  
END
```

```
IF A2 <> '' THEN
DO
nPortNumber = A2
END
ELSE
DO
nPortNumber = 80
END

/* Загрузить REXX Socket Library если еще не загружена*/
IF RxFuncQuery("SockLoadFuncs") THEN
DO
rc = RxFuncAdd("SockLoadFuncs", "rxSock", "SockLoadFuncs")
rc = SockLoadFuncs()
END

IF RxFuncQuery("SysLoadFuncs") THEN
DO
rc = RxFuncAdd( "SysLoadFuncs", "RexxUtil", "SysLoadFuncs")
rc = SysLoadFuncs()
END

rc=SockGetHostByName(sServer, "host.")

IF rc <> 1 THEN
DO
SAY 'CANNOT RESOLVE HOSTNAME TO ADDRESS: 'sServer
EXIT -1
END

SAY 'Trying server:'host.name', address:'host.addr', port:'nPortNumber

socket = SockSocket('AF_INET', 'SOCK_STREAM', 0)
IF socket < 0 THEN
DO
SAY 'UNABLE TO CREATE A SOCKET'
EXIT -1
END
```

```
address.family = 'AF_INET'
address.port = nPortNumber
address.addr = host.addr

rc = SockConnect(socket,'address.')
IF rc < 0 THEN
  DO
    SAY 'UNABLE TO CONNECT TO SERVER:'address.addr
    SIGNAL DO
  END

rc = SockSend(socket, sRequest)

SAY 'REQUEST:*****'

Resp = ''
DO FOREVER
  rc = SockRecv(socket,"sReceive",256)
  IF rc <= 0 THEN
    LEAVE
  Resp = Resp || sReceive
END

SAY ''

/* CR -> CRLF */
nStart = 1
nStop = pos(X2C("0A"), Resp)
do while nStop > 0
  SAY SUBSTR(Resp,nStart,nStop-nStart)
  nStart = nStop + 1
  nStop = pos(X2C("0A"), Resp, nStart)
end

DO:
rc = SockShutDown(socket,2)
rc = SockClose(socket)
```

При сборке DLL из нескольких объектных модулей программист должен предоставить DEF-файл (пример 3.10). В этом файле содержится перечисление символов, экспортируемых библиотекой (в отличие от обычных, "архив-

ных" библиотек, набор этих символов не обязательно равен объединению наборов экспортных символов всех включенных в библиотеку объектов), а также некоторые другие параметры. Например, можно указать, что DLL имеет функции инициализации и терминации. Эти функции могут запускаться как при первой загрузке библиотеки (`INITGLOBAL`), так и при подключении библиотеки очередной программой (`INITINSTANCE`). Можно также управлять разделением сегмента данных DLL — применять общий сегмент данных для всех программ, использующих библиотеку, или создавать свою копию для каждой программы.

Пример 3.10. DEF-файл из примеров кода VisualAge C++ V3.0

```
LIBRARY REXXUTIL INITINSTANCE LONGNAMES
PROTMODE
DESCRIPTION 'REXXUTIL Utilities - (c) Copyright IBM Corporation 1991'
DATA MULTIPLE NONSHARED
STACKSIZE 32768
EXPORTS
    SYSCLS      = SysCls      @1
    SYSCURPOS   = SysCurPos   @2
    SYSCURSTATE = SysCurState @3
    SYSDRIVEINFO = SysDriveInfo @4
    SYSDRIVEMAP = SysDriveMap @5
    SYSDROPFUNCS = SysDropFuncs @6
    SYSFILEDELETE = SysFileDelete @7
    SYSFILESEARCH = SysFileSearch @8
    SYSFILETREE = SysFileTree @9
    SYSGETMESSAGE = SysGetMessage @10
    SYSINI       = SysIni       @11
    SYSLOADFUNCS = SysLoadFuncs @12
    SYSMKDIR     = SysMkDir     @13
    SYSOS2VER    = SysOS2Ver    @14
    SYSRMDIR    = SysRmDir    @15
    SYSSEARCHPATH = SysSearchPath @16
    SYSSLEEP     = SysSleep     @17
    SYSTEMPFILENAME = SysTempFileName @18
    SYSTEXTSCREENREAD = SysTextScreenRead @19
    SYSTEXTSCREENSIZE = SysTextScreenSize @20
    SYSGETEA     = SysGetEA     @21
    SYSPUTEA     = SysPutEA     @22
    SYSWAITNAMEDPIPE = SysWaitNamedPipe @23
```

Современные версии компиляторов для Win32, например, Microsoft Visual C, позволяют управлять сборкой DLL без создания DEF-файлов, используя при описании функций и переменных нестандартные ключевые слова. Так, чтобы функция экспорттировалась из DLL, при ее описании необходимо использовать ключевое слово `__declspec(dllexport)`.

DLL являются удобным средством разделения кода и создания отдельно загружаемых программных модулей, но их использование сопряжено с определенной проблемой, которая будет подробнее объясняться в разд. 5.4. Забегая вперед, скажем, что концепция разделяемых DLL наилучше естественна в системах, где все задачи используют единое адресное пространство — но при этом ошибка в любой из программ может привести к порче данных или кода другой задачи. Стандартный же способ борьбы с этой проблемой — выделение каждому процессу своего адресного пространства — значительно усложняет разделение кода.

Другая проблема, обусловленная широким использованием разделяемого кода, состоит в слежении за версией этого кода. Действительно, представим себе жизненную ситуацию: в системе одновременно загружены тридцать программ, использующих библиотеку LIBC.DLL. При этом десять из них разрабатывались и тестировались с версией 1.0 этой библиотеки, пять — с версией 1.5 и пятнадцать — с версией 1.5a. Понятно, что рассчитывать на устойчивую работу всех тридцати программ можно только при условии, что все три версии библиотеки полностью совместимы снизу вверх не только по набору вызовов и их параметров, но и по точной семантике каждого из этих вызовов. Последнее требование иногда формулируют как *bug-for-bug compatibility* (корректно перевести это словосочетание можно так: *полная совместимость не только по спецификациям, но и по отклонениям от них*).

Казалось бы, исправление ошибок должно лишь улучшать работу программ, использующих исправленный код. На практике же бывают ситуации, когда код основной программы содержит собственные обходные пути, компенсирующие ошибки в библиотеке. Эти обходы могут быть как внесены сознательно (когда поставщик библиотеки исправит, еще неизвестно, а программа нужна заказчику сейчас), так и получиться сами собой (арифметический знак, перепутанный четное число раз, и т. д.). В этих случаях исправление ошибки может привести к труднопредсказуемым последствиям. Нельзя также забывать и о возможности внесения новых ошибок при исправлении старых, поэтому при разработке и эксплуатации сложных программных систем необходимо тщательно следить за тем, что именно и где изменилось.

Требование "совместимости с точностью до ошибок" — это лишь полемически заостренная формулировка требования контролируемости поведения кода. Из приведенных ранее соображений понятно, что нарушения такой кон-

тролируемости представляют собой проблему, которая, не будучи так или иначе разрешена, может серьезно усложнить работу администраторов системы и приложений.

Разделяемый код в системах семейства Windows

Катастрофические масштабы эта проблема принимает в системах семейства Windows, где принято помещать в дистрибутивы прикладных программ все потенциально разделяемые модули, которые этой программе могут потребоваться, — среда исполнения компилятора и т. д. При этом каждое приложение считает своим долгом поместить свои разделяемые модули в C:\WINDOWS\SYSTEM32 (в Windows NT/2000/XP это заодно приводит к тому, что установка самой безобидной утилиты требует администраторских привилегий). Средств же проследить за тем, кто, какую версию, чего, куда и зачем положил, практически не предоставляется.

В лучшем случае установочная программа спрашивает: "Тут вот у вас что-то уже лежит, перезаписать?". Стандартный деинсталлятор содержит список DLL, которые принадлежат данному приложению, и осознает тот факт, что эти же DLL используются кем-то еще, но не предоставляет (и, по-видимому, не пытается собрать) информации о том, кем именно они используются. Наличие реестра объектов COM не решает проблемы, потому что большая часть приносимого каждым приложением "разделяемого" кода (кавычки стоят потому, что значительная часть этого кода никому другому, кроме принесшего его приложения, не нужна) не является сервером COM.

В результате, когда, например, после установки MS Project 2000 перестает работать MS Office 2000 [MSkb RU270125], это никого не удивляет, а конфликты между приложениями различных разработчиков или разных "поколений" считаются неизбежными. Установить же в одной системе и использовать хотя бы по-переменно две различные версии одного продукта просто невозможно — однако, когда каждая версия продукта использует собственный формат данных, а конверсия между ними неидеальна, это часто оказывается желательно.

Разработчики же и тестеры, которым надо обеспечить совместимость с различными версиями существующих приложений, при этом просто оказываются в безвыходной ситуации. Неслучайно поставщики VMWare и других систем виртуальных машин для x86, как одно из главных достоинств своих систем, часто рекламируют возможность держать несколько копий Windows одновременно загруженными на одной машине.

Привлекательный путь решения этой проблемы — давать каждому приложению возможность указывать, какие именно DLL ему нужны и где их искать, и позволять одновременно загружать одноименные DLL с разной семантикой — на самом деле вовсе не прост как с точки зрения реализации, так и с точки зрения управления системой. Системы с виртуальной памятью предлагают некоторые подходы к реализации данного пути, но это будет обсуждаться в разд. 5.4.

Современные версии Windows, начиная с Windows 2000, и OS/2, начиная приблизительно с MCP2 (точнее, в ядрах начиная с версии 14.093; это поведение включается установкой недокументированной переменной среды _LIBPATHSTRICT), допускают загрузку одноименных DLL, размещенных в разных каталогах. DLL при этом различаются по их полным путевым именам. Впрочем, в Windows это не решает всех проблем — в частности, зарегистрировать два одноименных

СОМ-сервера с интерфейсами разных версий в системе по прежнему невозможно, так что одновременно установить две разные версии MS Office или MS Internet Explorer по прежнему не получается.

3.11. Загрузка самой ОС

... тогда я, задыхаясь, схватил себя за волосы и рванул... И мы взлетели над осокой!

Г. Горин, к. ф. "Тот самый Мюнхгаузен"

При загрузке самой ОС возникает специфическая проблема: в пустой машине, скорее всего, нет программы, которая могла бы это сделать.

В системах, в которых программа находится в ПЗУ (или другой энергонезависимой памяти), этой проблемы не существует: при включении питания программа в памяти уже есть и сразу начинает исполняться. При включении питания или аппаратном сбросе процессор исполняет команду, находящуюся по определенному адресу, например, 0xFFFFFFF4. Если там находится ПЗУ, а в нем записана программа, она и начинает исполняться.

Компьютеры общего назначения также не могут обойтись без ПЗУ. Программа, записанная в нем, называется *загрузочным монитором*. Стартовая точка этой программы должна находиться как раз по тому адресу, по которому процессор передает управление в момент включения питания. Эта программа производит первичную инициализацию процессора, тестирование памяти и обязательного периферийного оборудования и, наконец, начинает загрузку системы. В компьютерах, совместимых с IBM PC, загрузочный монитор известен как BIOS.

На многих системах в ПЗУ бывает прошито нечто большее, чем первичный загрузчик. Это может быть целая контрольно-диагностическая система, называемая *консольным монитором*. Подобная система есть на всех машинах линии PDP-11/VAX и на VME-системах, рассчитанных на OS-9 или VxWorks. Такой монитор позволяет вам просматривать содержимое памяти по заданному адресу, записывать туда данные, запускать какую-то область памяти как программу и многое другое. Он же позволяет выбирать устройство, с которого будет производиться дальнейшая загрузка. В PDP-11/VAX на мониторе можно даже писать программы почти с таким же успехом, как на ассемблере. Нужно только уметь считать в уме в восьмеричной системе счисления.

На машинах фирмы Sun в качестве консольного монитора используется интерпретатор языка Forth. На ранних моделях IBM PC в ПЗУ был прошит интерпретатор BASIC. Именно поэтому клоны IBM PC имеют огромное количество плохо используемого адресного пространства выше сегмента 0xC000. Вы можете убедиться в том, что BASIC там должен быть, вызвав под DOS

прерывание 0x60. Вы получите на мониторе сообщение вроде: NO ROM BASIC. PRESS ANY KEY TO REBOOT. Вообще говоря, BASIC не является консольным монитором в строгом смысле этого слова, т. к. получает управление не перед загрузкой, а лишь после того, как загрузка со всех устройств завершилась неудачей.

После запуска консольного монитора и инициализации системы вы можете приказать системе начать собственно загрузку ОС. На IBM PC такое приказание отдается автоматически, и часто загрузка производится вовсе не с того устройства, с которого хотелось бы. На этом и основан жизненный цикл загрузочных вирусов, который рассматривается далее в этом разделе

Чтобы загрузочный монитор смог что бы то ни было загрузить, он должен уметь проинициализировать устройство, с которого предполагается загрузка, и считать с него загружаемый код. Поэтому загрузочный монитор обязан содержать модуль, способный управлять загрузочным устройством. Например, типичный BIOS PC-совместимого компьютера содержит модули управления гибким и жестким дисками с интерфейсом Seagate 506 (в современных компьютерах это обычно интерфейс и диск EIDE/ATA или SATA, отличающиеся от Seagate 506 конструктивом, но программно совместимые с ним сверху вниз, см. раздел 9.6).

Кроме того, конструктивы многих систем допускают установку ПЗУ на платах контроллеров дополнительных устройств. Это ПЗУ должно содержать программный модуль, способный проинициализировать устройство и произвести загрузку с него (рис. 3.15).

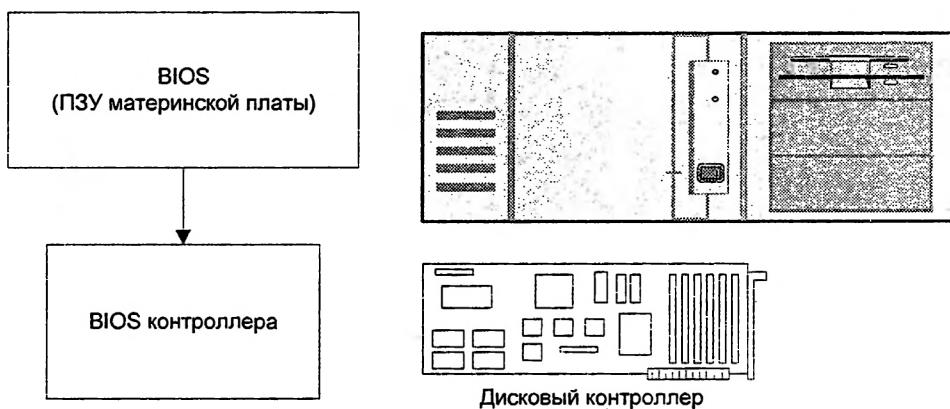


Рис. 3.15. Системное ПЗУ и BIOS дискового контроллера

Как правило, сервисы загрузочного монитора доступны загружаемой системе. Так, модуль управления дисками BIOS PC-совместимых компьютеров

предоставляет функции считывания и записи отдельных секторов диска. Доступ к функциям ПЗУ позволяет значительно сократить код первичного загрузчика ОС и, нередко, сделать его независимым от устройства.

Проще всего происходит загрузка с различных последовательных устройств — лент, перфолент, магнитофонов, перфокарточных считывателей и т. д. Загрузочный монитор считывает в память все, что можно считывать с данного устройства, и передает управление на начало той информации, которую прочитал.

В современных системах такая загрузка практически не используется. В них загрузка происходит с устройств с произвольным доступом, как правило — с дисков. При этом обычно в память считывается нулевой сектор нулевой дорожки диска. Содержимое этого сектора называют *первичным загрузчиком*. В IBM PC этот загрузчик называют *загрузочным сектором*, или *boot-сектором*.

Как правило, первичный загрузчик, пользуясь сервисами загрузочного монитора, ищет на диске начало файловой системы своей родной ОС, находит в этой файловой системе файл с определенным именем, считывает его в память и передает данному файлу управление. В простейшем случае такой файл и является ядром операционной системы.

Размер первичного загрузчика ограничен чаще всего размером сектора на диске, т. е. 512 байтами. Если файловая система имеет сложную структуру, иногда первичному загрузчику приходится считывать *вторичный*, размер которого может быть намного больше. Из-за большего размера этот загрузчик намного умнее и в состоянии разобраться в структурах файловой системы. В некоторых случаях используются *третичные* загрузчики.

Это последовательное исполнение втягивающих друг друга загрузчиков возрастающей сложности называется *бутстрэпом* (bootstrap), что можно перевести как "втягивание [себя] за ремешки на ботинках".

Порядок загрузки РС-совместимых компьютеров и загрузочные вирусы

В BIOS старых РС-совместимых компьютеров порядок перебора устройств при загрузке был жестко задан и включал в себя первый накопитель для гибких магнитных дисков и первый жесткий диск. Если пользователь случайно забывал дискету в накопителе, загрузка происходила с нее.

Существовал целый класс вирусов, которые записывали себя в загрузочные секторы жестких и гибких дисков. Такие вирусы так и называются — загрузочные вирусы (*boot virus*). При этом они сохраняли старый загрузочный сектор с помощью различных приемов, которые отличались большим разнообразием. При загрузке с зараженного диска такой вирус копирует себя в верхнюю часть оперативной памяти и корректирует показания BIOS, указывая, что памяти в системе меньше, чем на самом деле, или тем или иным способом маскирует

себя под системное ПЗУ. Кроме этого, вирус перехватывает сервисы int13 так, что все обращения к диску проходят через него и лишь затем передаются BIOS. Затем он загружает старый загрузочный сектор, и загрузка системы происходит обычным порядком — но вирус остается в памяти.

Примитивные вирусы такого типа легко обнаружимы простым просмотром загрузочного сектора диска и сравнением содержимого с тем, что там должно находиться. К сожалению, большинство распространенных "загрузчиков" достаточно умны и перехватывают попытки обращения к загрузочному сектору, подменяя его содержимое сохраненной копией. Поэтому, пользуясь средствами системы, зараженной таким вирусом, практически невозможно обнаружить факт ее заражения.

При установке в накопитель новой дискеты, вирус записывает себя в ее загрузочный сектор, так что все дискеты, с которыми работали на зараженной системе, также оказываются заражены. При обычной работе такие дискеты практически безопасны (впрочем, есть примеры так называемых "поливалентных" вирусов, которые умеют размножаться как через загрузочный сектор, так и другими средствами), но если по ошибке с нее загрузиться, то машина, с которой это произошло, также окажется заражена. В те времена, когда дискеты служили основным средством обмена данными между компьютерами, подобные вирусы представляли очень большую опасность.

Номенклатура загрузочных вирусов очень широка, а их функциональность отличается большим разнообразием и варьирует в диапазоне от простого саморазмножения (как, например, у классического вируса *Stone*, который, по видимому, был написан с единственной целью распространения лозунга о легализации марихуаны) до весьма изощренных разрушительных действий, обычно происходящих при наступлении определенной даты (как у столь же известного вируса *Chih/C Chernobyl*, который уничтожает системные структуры данных на диске 26 мая каждого года). Подавляющее большинство подобных вирусов полагается на то, что загружаемая система будет пользоваться сервисами BIOS при работе с диском, как это делает DOS, поэтому для современных ОС они не очень опасны, хотя и могут нарушить процесс их загрузки. Тем не менее жизнеспособные популяции таких вирусов существуют и по сей день — в 2004 году мне доводилось видеть дискету, зараженную в интернет-кафе какой-то загрузочной дрянью, которую я, к сожалению, не смог идентифицировать.

Вирус, представляющий опасность для современных ОС, должен был бы быть системно-зависимым. Такой вирус должен модифицировать дисковый образ системы — скорее всего, загрузочные модули ядра или драйверов дисковых устройств и файловых систем. Разумеется, этот вирус намного сложнее в реализации, чем примитивные вирусы для DOS. Он должен иметь гораздо больший объем кода, чем загрузочные вирусы предыдущих поколений, и, возможно, он должен использовать многоступенчатую схему загрузки, аналогичную загрузке полноценной ОС. Так что подобному вирусу, наверное, довольно сложно было бы скрыть свое наличие на диске и процесс размножения. К тому же, в современных условиях действует ряд факторов, ограничивающих размножение "загрузчиков", поэтому сколько-нибудь серьезных эпидемий вирусов данного типа на сегодня неизвестно. В частности, BIOS современных РС предоставляют некоторые средства, затрудняющие распространение таких вирусов.

Во-первых, практически все современные BIOS позволяют задавать, иногда в довольно широких пределах, список устройств, с которых возможна загрузка, и

порядок, в котором загрузка должна происходить (переход к следующему устройству происходит, если загрузка с текущего завершилась неудачей, скорее всего потому, что в накопителе не оказалось носителя данных). При нормальной работе из этого списка необходимо исключить все устройства со сменными носителями, такие как дискеты, CD и устройства USB. Разумеется, при установке ОС или для загрузки с восстановительного диска одно из этих устройств необходимо будет вновь включить, но вряд ли эти операции следует считать частью нормальной работы.

Во-вторых, многие BIOS поддерживают специальную функцию, которую обычно так и называют "антивирусной защитой", а именно, они блокируют запись в загрузочные секторы всех дисков. Эта защита эффективна только при работе под DOS (более современные ОС работают с диском самостоятельно, без использования BIOS) и мешает установке ОС, поэтому на время установки ее также необходимо выключить. Поскольку DOS используется все реже и реже, эффективность этого средства относительно невелика. К тому же, такие средства не могут защитить компакт-диски, USB-накопители и др.

В современных условиях дискеты используются относительно редко; по идее, большую опасность должны были бы представлять загрузочные вирусы, распространяющиеся на CD. Впрочем, жизнеспособных вирусов такого рода до сих пор неизвестно; возможно, дело в том, что запись компакт-дисков осуществляется не встроенными средствами ОС, а внешними программами, при том что универсальных средств "встраивания" в такие программы и контролируемой модификации их поведения практически нет.

Можно ожидать также распространения загрузочных вирусов на накопителях (как дисковых, так и электронных) с интерфейсом USB; возможно, главным фактором, останавливающим это, служит то, что многие современные BIOS не умеют загружать систему с USB-устройств. По мере того как BIOS, поддерживающие подобную загрузку, будут получать распространение, положение может измениться. Поэтому тот факт, что жизнеспособные загрузочные вирусы для современных ОС не известны, вовсе не означает, что можно расслабиться и потерять контроль над тем, кто и с какого устройства может загрузить вашу вычислительную систему.

Большую практическую роль играет еще один способ загрузки — загрузка по сети. Она происходит аналогично загрузке с диска: ПЗУ, установленное на сетевой карте, посыпает в сеть пакет стандартного содержания, который содержит запрос к серверу удаленной загрузки. Этот сервер передает по сети вторичный загрузчик и т. д. Такая технология незаменима при загрузке бездисковых рабочих станций. Централизованное размещение загрузочных образов рабочих станций на сервере упрощает управление ими, защищает настройки ОС от случайных и злонамеренных модификаций и существенно удешевляет эксплуатацию больших парков настольных компьютеров, поэтому по сети нередко загружаются и машины, имеющие жесткий диск.

Проще всего происходит загрузка систем, ядро которых вместе со всеми дополнительными модулями (драйверами устройств, файловых систем и др.) собрано в единый загрузочный модуль. Например, в системах семейства Unix

ядро так и называется /unix (в FreeBSD — /vmunix, в Linux — /vmlinuX, или, в случае упакованного ядра, /vmlinuz).

При переконфигурации системы, добавлении или удалении драйверов и других модулей необходима пересборка ядра, которая может производиться либо стандартным системным редактором связей, либо специальными утилитами *генерации системы*. Для такой пересборки в поставку системы должны входить либо исходные тексты (как у Linux, BSD Unix, RSX-11, VMS), либо объектные модули ядра, как у коммерческих Unix-систем. Сборка ядра из объектных модулей на современных системах занимает несколько минут. Полная перекомпиляция ядра из исходных текстов, конечно, продолжается существенно дольше.

На случай, если системный администратор ошибется и соберет неработоспособное ядро, вторичный загрузчик таких систем часто предоставляет возможность выбрать файл, который следует загрузить. Ядро таких систем часто не использует никаких конфигурационных файлов — все настройки также задаются при генерации.

Большинство современных ОС используют более сложную форму загрузки, при которой дополнительные модули подгружаются уже после старта самого ядра. В терминах предыдущих разделов это называется "сборка в момент загрузки". Список модулей, которые необходимо загрузить, а также параметры настройки ядра, собраны в специальном файле или нескольких файлах. У DOS и OS/2 этот файл называется CONFIG.SYS, у Win32-систем — реестром (*registry*).

Сложность при таком способе загрузки состоит в том, что ядро, еще полностью не проинициализовавшись, уже должно быть способно работать с файловой системой, находить в ней файлы и считывать их в память.

Особенно сложен этот способ тогда, когда драйверы загрузочного диска и загрузочной файловой системы сами являются подгружаемыми модулями. Обычно при этом ядро пользуется функциями работы с файловой системой, предоставляемыми вторичным (или третичным, в общем, последним по порядку) загрузчиком, до тех пор, пока не проинициализирует собственные модули. Вторичный загрузчик обязан уметь читать загрузочные файлы, иначе он не смог бы найти ядро. Если поставщики ОС не удосужились написать соответствующий вторичный загрузчик, а предоставили только драйвер файловой системы, ОС сможет работать с такой файловой системой, но не сможет из нее загружаться.

Некоторые системы, например DOS, могут грузиться только с устройств, поддерживаемых BIOS, и только из одного типа файловой системы (ФС) — FAT, драйвер которой скомпонован с ядром. Любопытное развитие этой идеи представляет Linux, модули которой могут присоединяться к ядру как стати-

чески, так и динамически. Динамически могут подгружаться любые модули, кроме драйверов загрузочного диска и загрузочной ФС.

Преимущества, которые дает динамически собираемое в момент загрузки ядро, не так уж велики по сравнению с системами, в которых ядро собирается статически. Впрочем, ряд современных систем (Solaris, Linux, Netware, Windows 2000 и следующие версии Windows NT) идут в этом направлении дальше и позволяют подгружать модули уже после загрузки и даже выгружать их. Такая архитектура предъявляет определенные требования к интерфейсу модуля ядра (он должен уметь не только инициализировать сам себя и, если это необходимо, управлять им устройство, но и корректно освобождать все занятые им ресурсы при выгрузке), но дает значительные преимущества.

Во-первых, это допускает подгрузку модулей по запросу. При этом подсистемы, нужные только иногда, могут не загрузиться вообще. Даже те модули, которые нужны всегда, могут проинициализироваться, только когда станут нужны, уменьшив тем самым время от начала загрузки до старта некоторых сервисов.

Во-вторых, пожалуй, даже более важное при реальной эксплуатации преимущество состоит в возможности реконфигурировать систему без перезагрузки, что особенно полезно для систем коллективного пользования и совершенно незаменимо, если система допускает подключение внешних устройств "на ходу" (так называемое hot plug, дословно "горячее включение"). В современных персональных компьютерах включение на ходу поддерживаются для устройств с интерфейсами USB, FireWire (IEEE 1394), PCMCIA и т. д. Существует также редакция спецификаций периферийной шины PCI, допускающая горячую замену; это требует определенных изменений как от материнской платы, так и от плат устройств. Такие материнские платы применяются главным образом на серверах и позволяют заменять отказавшие компоненты без перезагрузки сервера (ценность этой возможности для системы коллективного пользования трудно переоценить). Конструктивы серверов высших ценовых категорий допускают также горячую замену и установку блоков ОЗУ и даже процессорных модулей (разумеется, последнее возможно только в многопроцессорных системах). Впервые такие возможности появились в больших компьютерах компании IBM в 70-е годы.

И, наконец, возможность выгрузки модулей ядра иногда (но не всегда, а лишь если поломка не мешает драйверу корректно освободить ресурсы) позволяет корректировать работу отдельных подсистем — опять-таки без перезагрузки всей ОС и пользовательских приложений.

Однако динамическая загрузка модулей ядра сопряжена и с весьма серьезной опасностью. В многопользовательских ОС право загрузки модулей ядра

обычно теми или иными средствами ограничивается. Часто такая загрузка допускается лишь от имени администратора системы. Но если злоумышленник сможет получить к системе административный доступ или сможет теми или иными способами спровоцировать администратора системы исполнить свой код, то такой злоумышленник сможет загрузить и модуль ядра. Это гораздо опаснее, чем запуск произвольного кода в пользовательском режиме, ведь троянский модуль ядра может перехватывать системные вызовы. Помимо прочего, это означает, что такой троянский модуль может перехватить все системные вызовы, при помощи которых можно обнаружить его наличие в системе (вызовы для просмотра списка активных процессов, просмотра файлов, просмотра конфигурационного файла или ветвей реестра системы). Такой модуль будет практически необнаружим. Особенно легко и надежно такой модуль может спрятаться себя от автоматических антивирусных сканеров — для этого ему достаточно перехватить системные вызовы, используемые этим сканером. Задача проверки такой системы на целостность средствами самой системы неразрешима — примерно по тем же причинам, по которым человек не может самостоятельно (без внешних референсов) убедиться в том, что он не сумасшедший.

Троянские программы, скрывающие свое наличие в системе, называются руткитами (rootkit). Особенно большую опасность такие программы представляют для Windows 2000/XP/2003, в которых большинство интерактивных пользователей работают с полномочиями локального администратора. В Windows NT 3x-4.0 была невозможна загрузка модулей ядра без перезагрузки ОС, а в Windows Vista предпринят ряд мер, позволяющих работать в системе без административных полномочий — впрочем, эффективность этих мер еще предстоит проверить на практике. Многие реальные троянские программы — черви, боты, spyware — являются руткитами и не обнаруживаются большинством автоматических сканеров.

Оказавшись в памяти и, так или иначе, подтянув все необходимые дополнительные модули, ядро запускает их подпрограммы инициализации. При динамической подгрузке инициализация модулей часто происходит по мере их загрузки. Обычно инициализация ядра завершается тем, что оно загружает определенную программу, которая продолжает инициализацию уже не ядра, но системы в целом. Например, системы семейства UNIX имеют специальную инициализационную программу, которая так и называется — `init`. Эта программа запускает различные процессы-демоны, например `cron`, — программу, которая умеет запускать другие заданные ей программы в заданные моменты времени, различные сетевые сервисы, программы, которые ждут ввода с терминалных устройств (`getty`), и т. д. Набор запускаемых программ задается в файле `/etc/inittab` (в разных версиях системы данный файл может иметь разные имена; `/etc/inittab` используется в System V и Linux). Админист-

ратор системы может редактировать этот файл и устанавливать те сервисы, которые в данный момент нужны, избавляясь от тех, которые не требуются, и т. д.

Программа `init` остается запущенной все время работы системы. Она, как правило, следит за дальнейшей судьбой запущенных ею процессов. В зависимости от заданных в файле `/etc/inittab` параметров, она может либо перезапускать процесс после его завершения, либо не делать этого.

Аналогичный инициализационный сервис в той или иной форме предоставляют все современные операционные системы.

Загрузка Sun Solaris на компьютерах x86

Полный цикл загрузки Solaris 9 (версия Unix System V Release 4, поставляемая фирмой Sun) на компьютерах x86 происходит в шесть этапов (в более новой версии, Solaris 10, последовательность загрузки несколько изменена; вместо нестандартного MBR и DCU используется ставший стандартом дефакто свободно распространяемый вторичный загрузчик GRUB).

Первые три этапа стандартны для всех ОС, работающих на iBM PC-совместимой технике. При включении компьютера запускается прошитый в ПЗУ BIOS. Он проводит тестирование процессора и памяти и инициализацию машины, в том числе, на современных машинах, назначение адресов и линий запроса прерывания устройствам PCI и ISA PnP. В процессе инициализации BIOS устанавливает обработчик прерывания Int 13h. Этот обработчик умеет считывать и записывать отдельные секторы жестких и гибких дисков и производить некоторые другие операции над дисковыми устройствами. Первичные загрузчики ОС обычно пользуются этим сервисом. Некоторые ОС, например MS/DR-DOS, используют данный сервис не только при загрузке, но и при работе, и потому могут не иметь собственного модуля управления дисками.

Если загрузка происходит с жесткого диска, BIOS загружает в память и запускает нулевой сектор нулевой дорожки диска. Этот сектор обычно содержит не первичный загрузчик операционной системы, а MBR (Master Boot Record, главная загрузочная запись). Данная программа обеспечивает разбиение физического жесткого диска на несколько логических разделов (partition) и возможность попеременной загрузки различных ОС, установленных в этих разделах (рис. 3.16).

Разбиение физического диска на логические программы MBR осуществляет на основе содержащейся в ее теле таблицы разделов (partition table), которая содержит границы и типы разделов. MBR перехватывает прерывание Int 13h и транслирует обращения к дисковой подсистеме так, что обращения к логическому диску N преобразуются в обращения к N-ному разделу физического диска.

Один из разделов диска должен быть помечен как активный или загрузочный. MBR загружает начальный сектор данного раздела — обычно это и есть первичный загрузчик ОС. Многие реализации MBR, в том числе и поставляемая с Solaris, могут предоставлять пользователю выбор раздела, с которого следует начинать загрузку. Выбор обычно предоставляется в форме паузы, в течение которой пользователь может нажать какую-то клавишу или комбинацию клавиш. После этого пользователю будет выдан список разделов в виде меню. Если ничего не будет нажато, начнется загрузка с текущего активного раздела.

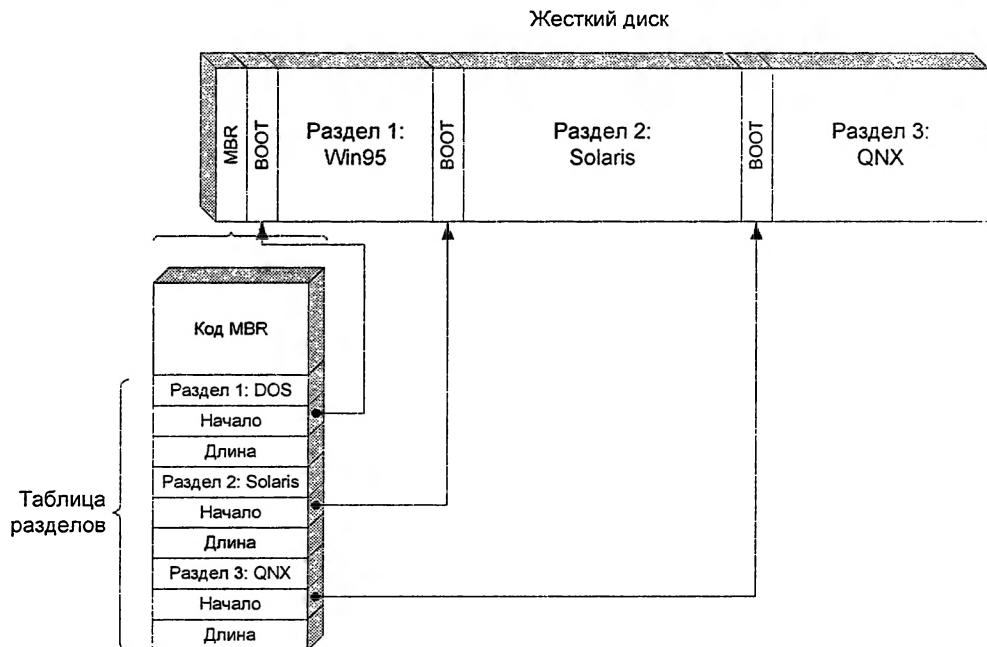


Рис. 3.16. MBR и таблица разделов

Так или иначе, но загрузочный сектор — по совместительству, первичный загрузчик Solaris — оказывается в памяти и начинает исполняться. Исполнение его состоит в том, что он загружает — нет, еще не ядро, а специальную программу, называемую DCU (Device Configuration Utility, утилита конфигурации устройств). Основное назначение этой программы — имитация сервисов консольного монитора компьютеров фирмы Sun на основе процессоров SPARC.

DCU производит идентификацию установленного в машине оборудования. Пользователь может вмешаться в этот процесс и, например, указать системе, что такого-то устройства в конфигурации нет, даже если физически оно и существует, или установить драйверы для нового типа устройств. Драйверы, используемые DCU, отличаются от драйверов, используемых самим Solaris, называются они BEF (Boot Executable File) и начинают исполнение, как и сама DCU, в реальном режиме процессора x86.

Найдя все необходимое оборудование, DCU запускает вторичный загрузчик Solaris. Логический диск, выделенный Solaris, имеет внутреннюю структуру и также разбит на несколько разделов (рис. 3.17). Чтобы не путать эти разделы с разделами, создаваемыми MBR, их называют *слайсами* (slice). Загрузочный диск Solaris должен иметь минимум два слайса — Root (корневая файловая система) и Boot, в котором и размещаются вторичный загрузчик и DCU.

Вторичный загрузчик, пользуясь BEF-модулем загрузочного диска для доступа к этому диску, считывает таблицу слайсов и находит корневую файловую систему. В этой файловой системе он выбирает файл /kernel/unix, который и является ядром Solaris. В действительности, вторичный загрузчик исполняет командный файл, в котором могут присутствовать условные операторы, и, в зависимо-

сти от тех или иных условий, в качестве ядра могут быть использованы различные файлы. `/kernel/unix` используется по умолчанию.

Кроме того, пользователю предоставляется пауза (по умолчанию 5 секунд), в течение которой он может прервать загрузку по умолчанию и приказать загрузить какой-то другой файл или тот же файл, но с другими параметрами.

Будучи так или иначе загружено, ядро, пользуясь сервисами вторичного загрузчика, считывает файл `/etc/system`, в котором указаны параметры настройки системы. Затем, пользуясь информацией, предоставленной DCU, ядро формирует дерево устройств — список установленного в системе оборудования, и в соответствии с этим списком начинает подгружать модули, управляющие устройствами, — драйверы. Подгрузка по-прежнему происходит посредством сервисов вторичного загрузчика — ведь все драйверы размещены на загрузочном диске и в корневой файловой системе, в том числе и драйверы самого этого диска и этой файловой системы.

Раздел Solaris

Слайд 2

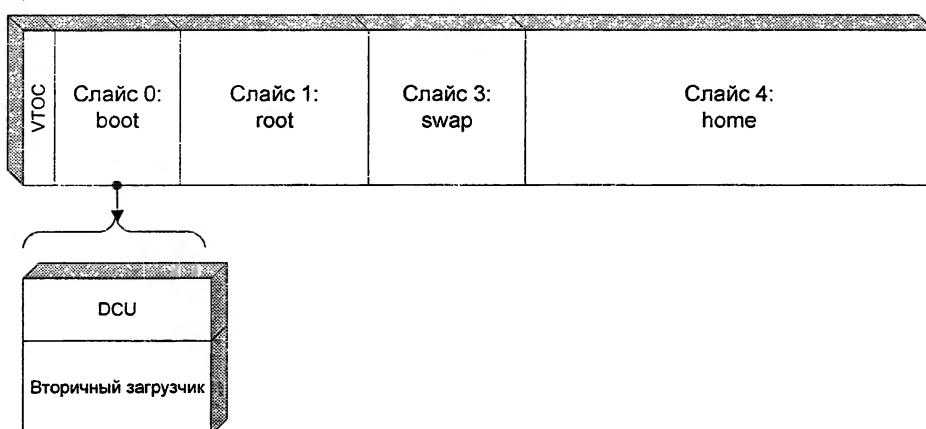


Рис. 3.17. Структура раздела Solaris

Загрузив драйверы всех дисковых устройств и файловых систем (а при загрузке из сети — также сетевых контроллеров и сетевых протоколов), ядро начинает их инициализацию. С этого момента использовать сервисы вторичного загрузчика становится невозможно, но они уже и не нужны. Проинициализировав собственный драйвер загрузочного диска и корневой файловой системы, ядро запускает программу `/sbin/init`, которая подключает остальные диски и файловые системы, если они есть, указывает параметры сетевых устройств и инициализирует их, запускает обязательные сервисы, в общем, производит всю остальную стартовую настройку системы.

Загрузка Windows NT

Первые этапы загрузки Windows NT/2000/XP/2003 полностью аналогичны первым этапам загрузки Solaris/x86. При включении питания процессор запускает

прошитый в ПЗУ BIOS, BIOS инициализирует машину, находит загрузочное устройство и копирует в память его загрузочный сектор или MBR жесткого диска, который, в свою очередь, копирует в память и запускает загрузочный сектор активного раздела.

Загрузочный сектор раздела находит в корневом каталоге файл NTLDLR (официальная расшифровка названия этого файла неизвестна, по-видимому, это NT LoaDeR), являющийся вторичным загрузчиком ОС. Вторичный загрузчик читает из корневого каталога текстовый файл BOOT.INI, в котором перечислены все доступные для загрузки ОС. К сожалению, NTLDLR Windows XP/2003 и более ранних версий может загружать только версии Windows NT или DOS (а также системы, использующие DOS в качестве вторичного загрузчика, такие, как Windows 95/98/Me).

В Windows Vista этот недостаток в значительной мере исправлен. В ряде источников, например, [port25.technet.com/vista/boot], описывается процедура, при помощи которой можно заставить Windows Boot Manager (вторичный загрузчик Vista) загружать Linux и, вообще говоря, любые ОС, загружаемые при помощи GRUB (FreeBSD, Solaris 10). К сожалению, эта процедура включает в себя копирование загрузочного сектора GRUB в файл и выполнение сложных обрядов, а по некоторым данным, также и приношение кровавых жертв.

Формат файла BOOT.INI в основном должен быть понятен из примера 3.11. Частичное описание его синтаксиса можно найти во встроенной документации Windows, а полное — по адресу [www.microsoft.com/boot.ini]. Файл начинается с секции настройки загрузчика, где указан интервал времени, в течении которого пользователь может вмешаться в процесс загрузки, и ОС, которую следует грузить, если пользователь ничего не выберет (в том числе если пользователь не успеет ничего выбрать, потому что указан слишком маленький интервал ожидания), а также ряд других параметров.

Для каждой версии Windows NT указаны или могут быть указаны:

- раздел, с которого следует начинать загрузку;
- "системный" каталог на этом разделе, в котором расположен реестр, системные программы и DLL;
- имя файла, в котором находится ядро (по умолчанию \NTKRNL для Windows NT и Windows 2000 и \Windows\system32\ntoskrnl.exe для Windows XP); в примере 3.11 этот параметр не указан;
- параметры, которые следует передать ядру;
- имя, под которым эта ОС будет представлена в стартовом меню.

Для версий DOS можно указать только загрузочный раздел и имя в меню.

При загрузке DOS или Windows 95/98/ME NTLDLR имитирует работу загрузочного сектора раздела DOS, т. е. загружает в память файлы MSDOS.SYS и IO.SYS из корневого каталога и передает управление на стартовый адрес файла MSDOS.SYS. При загрузке ядра какой-либо из версий Windows NT NTLDLR загружает в память ядро и предоставляет ему сервисы доступа к файловой системе — во всяком случае, до тех пор, пока ядро не прочитает реестр, не загрузит и не проинициализирует драйверы дисковых устройств и файловых систем. Если у читателя есть время и ресурсы для экспериментов, я советую установить какую-либо из версий NT на разные файловые системы, например на

FAT32 и NTFS. При этом можно будет убедиться в том, что файлы NTLDR на разных ФС будут разными, как по размеру, так и для побайтового сравнения. Действительно, ведь они должны содержать драйверы разных ФС.

Оказавшись в памяти, ядро выводит на экран графический логотип, читает реестр и загружает минимальное подмножество драйверов, необходимых для продолжения загрузки, а именно, уже упоминавшиеся драйверы дисковых устройств и файловых систем. Эти операции ядро осуществляет с использованием сервисов вторичного загрузчика. Затем ядро инициализирует эти драйверы. С этого момента вторичный загрузчик больше не может использоваться, ведь его операции над дисковыми контроллерами могут конфликтовать с операциями драйверов ядра — но сервисы вторичного загрузчика более не нужны. После этого ядро считывает весь реестр и начинает загружать остальные драйверы и указанные в реестре программы. Программы, в свою очередь, делятся на фоновые сервисные процессы (их список доступен в контрольной панели через иконку **Сервисы**; разумеется, при загрузке стартуют только те сервисы, для которых указан автоматический старт) и интерактивную "оболочку" (Shell), которая устанавливает графический режим, выдает пользователю окно с приглашением ввести имя и пароль и запускает собственно пользовательскую оболочку, Проводник (Explorer).

В специальных конфигурациях системы, например в той, которая загружается с дистрибутивного CD, при установке системы вместо графической оболочки используется специализированная программа, работающая в текстовом режиме. Эта программа задает пользователю разные вопросы, позволяет разбить жесткий диск на разделы, создать файловую систему и скопировать туда файлы, необходимые для загрузки с диска и продолжения установки. На дистрибутивных дисках Windows XP вместо этой установочной оболочки можно также выбрать загрузку так называемой "консоли восстановления", которая позволяет выполнить несколько простых восстановительных операций над образом системы на жестком диске.

Пример 3.11 Файл BOOT.INI

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINNT
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINNT="Microsoft Windows 2000
Professional" /fastdetect
```

Существуют ОС, которые не умеют самостоятельно выполнять весь цикл бутстрапа. Они используют более примитивную операционную систему, которая исполняет их вторичный (или какой это уже будет по счету) загрузчик, и помогает данному загрузчику поместить в память ядро ОС. На процессорах x86 в качестве стартовой системы часто используется MS/DR-DOS, а загрузчик новой ОС оформляется в виде EXE-файла.

Таким образом устроены системы MS Windows 1.x—3.x, Windows 95/98/ME, DesqView и ряд других "многозадачников" для MS-DOS. Так же загружается сервер Nowell Netware, система Oberon для x86, программы, написанные для различных расширителей DOS (DOS extenders), и т. д. Многие из перечисленных систем, например Windows (версии младше 3.11 — в обязательном порядке, а 3.11 и 95/98/ME только в определенных конфигурациях), используют DOS и во время работы в качестве дисковой подсистемы. Тем не менее эти программные пакеты умеют самостоятельно загружать пользовательские программы и выполнять все перечисленные во введении функции и должны, в соответствии с нашим определением, считаться полноценными операционными системами.

Вопросы для самопроверки

1. Чем абсолютный загрузочный модуль отличается от относительного? Укажите отличия как с функциональной точки зрения, так и с точки зрения внутренней структуры модуля.
2. Как вы считаете, почему загрузка с помощью разделов памяти была названа непрактичной?
3. Какая информация хранится в записях таблицы перемещений относительного загрузочного модуля?
4. Приведите примеры языков программирования с поздним и ранним связыванием.
5. Какая информация хранится в записях таблицы экспорта объектного модуля? В записях таблицы импорта?
6. Почему объектный модуль должен содержать таблицу перемещения?
7. Рассмотрим сборку загрузочного модуля из нескольких объектных модулей и архивной библиотеки. Что должно произойти, если один из объектных модулей определяет символ (например, функцию языка С), который совпадает по имени с одной из функций, определенных в библиотеке. К какой информации вам не хватает для однозначного ответа на этот вопрос? (Ответ может иметь довольно большую практическую ценность. Действительно, таким способом можно было бы переопределять некоторые из функций стандартной библиотеки языка С, сохраняя при этом возможность использовать стандартные определения других функций.)
8. Какие именно внешние и перемещаемые ссылки содержатся в отладочной информации? (Для ответа на этот вопрос необходимо знакомство с материалом предыдущей главы.)

9. Каких проблем можно ожидать при загрузке одноименных DLL, различаемых по их путевым именам? Достаточно ли эти проблемы серьезны, чтобы в OS/2 такое поведение по умолчанию было выключено?
10. В разд. 3.11 неоднократно встречается утверждение, что драйверы загрузочного диска и загрузочной файловой системы могут инициализироваться только тогда, когда сервисы вторичного загрузчика, через которые ядро осуществляло доступ к загрузочному диску, уже не нужны. Почему это так?
11. (Вопрос знатокам ОС Linux.) Верно ли утверждение, что при загрузке Linux драйвер загрузочного устройства должен быть статически собран с ядром? Возможны ли другие решения, и если да, то в чем они состоят? В каких условиях эти решения могли бы быть целесообразны?

ГЛАВА 4



Управление оперативной памятью

Что-то с памятью моей
стало...

P. Рождественский

Основной ресурс системы, распределением которого занимается ОС, — оперативная память. Поэтому организация памяти оказывает большое влияние на структуру и возможности ОС. Во введении мы видели, что основное различие между двумя большими классами систем, которые в данной книге называются ОС и ДОС, состоит в том, используют эти системы средства трансляции адресов при доступе к памяти (так называемую виртуальную память) или нет. В настоящее время сложилась даже более интересная ситуация — переносимая операционная система UNIX, рассчитанная на машины со страничным диспетчером памяти, произвела жесткий отбор, и теперь практически все машины общего назначения, начиная от x86 и заканчивая суперкомпьютерами или, скажем, процессором Alpha, имеют именно такую организацию адресного пространства.

Самый простой вариант задачи управления памятью — отсутствие диспетчера памяти, т. е. совпадение физического и логического адресных пространств. В этих условиях система не может контролировать доступ пользовательских программ к памяти; если мы определяем процесс как единицу исполнения, обладающую собственным адресным пространством, то вся система вместе с пользовательскими задачами представляет собой единый процесс. Управление памятью со стороны ОС ограничивается только решением вопроса, какая память занята кодом и данными программы, а какая — свободна. Даже в этих условиях управление памятью представляет собой непростую задачу, для решения которой придумано много различных (и, в ряде отношений, не совсем удачных) решений.

В системах с виртуальной памятью мы сталкиваемся с вариантом той же самой задачи при управлении виртуальным адресным пространством в пределах процесса. Действительно, наличие диспетчера памяти дает довольно большую свободу во время отображения адресного пространства на физическую память, но при распределении самого адресного пространства мы сталкиваемся с той же самой задачей, что и при управлении открытой памятью: необходимо гарантировать, что ни один вновь создаваемый объект не будет конфликтовать по [виртуальным] адресам с уже существовавшими объектами. Поэтому многие из алгоритмов, первоначально придуманных для управления открытой физической памятью, находят применение при управлении виртуальным адресным пространством в пределах задачи.

4.1. Открытая память

Рассмотрим простейшую однозадачную ОС, в которой нет никакого диспетчера памяти и допускается работа только одной задачи. Именно так работают CP/M и RT-11 SJ (Single-Job, однозадачная). В этих системах программы загружаются с фиксированного адреса `PROG_START`. В CP/M это `0x100`; в RT-11 — `01000`. По адресам от 0 до начала программы находятся векторы прерываний, а в RT-11 — также и стек программы. Операционная система размещается в старших адресах памяти. Адрес `SYS_START`, с которого она начинается, зависит от количества памяти у машины и от конфигурации ОС (рис. 4.1).



Рис. 4.1. Управление памятью в однопроцессной ОС с открытой памятью

В этом случае управление памятью со стороны системы состоит в том, что загрузчик проверяет, поместится ли загружаемый модуль в пространство от `PROG_START` до `SYS_START`. Если объем памяти, который использует программа, не будет меняться во время ее исполнения, то на этом все управление и заканчивается.

Однако программа может использовать динамическое управление памятью, например, функцию `malloc()` или что-то в этом роде. В таком случае уже код `malloc()` должен следить за тем, чтобы не залезть в системные адреса. Как правило, динамическая память начинает размещаться с адреса `PROG_END =`

`PROG_START + PROG_SIZE`. `PROG_SIZE` в данном случае обозначает полный размер программы, т. е. размер ее кода, статических данных и области, выделенной под стек.

Функция `malloc()` поддерживает некоторую структуру данных, следящую за тем, какие блоки памяти из уже выделенных были освобождены, так называемый *пул* (pool, от англ. лужа, бассейн) или *кучу* (heap). При каждом новом запросе она сначала ищет блок подходящего размера в пуле, и только когда этот поиск завершится неудачей, просит новый участок памяти у системы. Для этого используется переменная, которая в библиотеке языка С называется `brk_addr` (рис. 4.2).

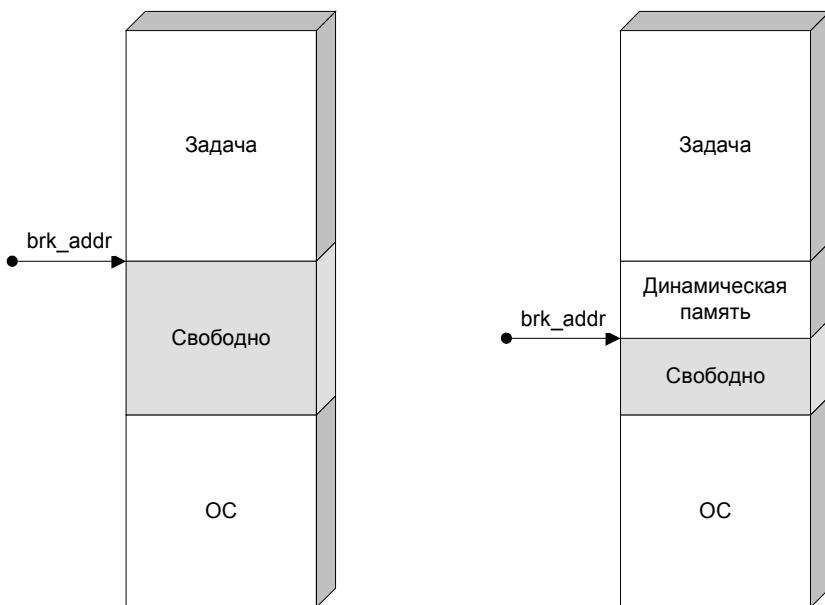


Рис. 4.2. Загруженная программа и `brk_addr`

Изначально эта переменная равна `PROG_END`, ее значение увеличивается при выделении новых блоков, но в некоторых случаях может и уменьшаться. Это происходит, когда программа освобождает блок, который заканчивается на текущем значении `brklevel`.

Аналогичным образом происходит выделение адресного пространства в многозадачных ОС с виртуальной памятью. Задача обычно занимает начальные адреса, но не с самого начала — например, в Unix первые 8 Мбайт виртуального адресного пространства заняты так называемой *сторожевой зоной* (guard area), доступ к которой запрещен. Затем идут код и статические данные задачи, затем ее динамическая память. Окончание динамической памяти

отмечается значением `brk_addr`. Как и в простых системах с открытой памятью, задача просит у ОС дополнительной памяти, вызывая системный вызов `sbrk` (пример 4.1).

Пример 4.1. Выделение дополнительной памяти в GNU LibC для Linux

morecore.c:

```
/* Copyright (C) 1991, 1992 Free Software Foundation, Inc.
```

Этот файл является частью библиотеки С проекта GNU (GNU C Library).

GNU C Library является свободным программным обеспечением;
вы можете передавать и/или модифицировать ее в соответствии
с положениями GNU General Public License версии 2 или (по вашему выбору)
любой более поздней версии.

Библиотека GNU C распространяется в надежде, что она будет полезна, но
БЕЗ КАКИХ-ЛИБО ГАРАНТИЙ; даже без неявно предполагаемых гарантий

КОММЕРЧЕСКОЙ ЦЕННОСТИ или ПРИГОДНОСТИ ДЛЯ КОНКРЕТНОЙ ЦЕЛИ.

Подробнее см. GNU General Public License.

Вы должны были получить копию GNU General Public License вместе с
GNU C Library; см. файл COPYING. Если вы ее не получили, напишите по
адресу: Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA.
*/

```
#ifndef      _MALLOC_INTERNAL
#define       _MALLOC_INTERNAL
#include <malloc.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <sys/syscall.h>
#endif

#ifndef      __GNU_LIBRARY__
#define      __sbrk      sbrk
#endif

extern void * __brk_addr;
extern __ptr_t __sbrk __P ((int increment));
```

```
extern int __init_brk __P ((void));  
  
#ifndef NULL  
#define NULL 0  
#endif  
  
/* Выделить еще INCREMENT байтов пространства данных  
и возвратить начало пространства данных или NULL при ошибках.  
Если INCREMENT отрицателен, сжать пространство данных. */  
  
__ptr_t  
__default_morecore (ptrdiff_t increment)  
{  
    __ptr_t result = __sbrk ((int) increment);  
    if (result == (__ptr_t) -1)  
        return NULL;  
    return result;  
}  
  
/* Эта функция почти полностью аналогична __default_morecore ().  
* Но она вызывается только однажды через __morecore.  
*/  
  
__ptr_t  
__default_morecore_init (ptrdiff_t increment)  
{  
    __ptr_t result;  
  
    if ( __init_brk() != 0)  
        return NULL;  
  
    if (__morecore == __default_morecore_init)  
        __morecore = __default_morecore;  
  
    result = __sbrk ((int) increment);  
    if (result == (__ptr_t) -1)  
        return NULL;  
    return result;  
}
```

В простейшем случае, все остальное адресное пространство доступно задаче; ОС при этом находится в отдельном адресном пространстве и задача ее не видит. Благодаря этому, с одной стороны, задача не может повредить код и данные ОС, и, с другой стороны, задача может использовать больше памяти.

В более сложных случаях код и данные задачи занимают несколько несмежных областей в ОЗУ (виртуальном или физическом). Это бывает в многозадачных системах с открытой памятью (Win16, старые версии Mac OS, Novell Netware и др.); в системах с виртуальной памятью это получается из-за использования разделяемых библиотек, разделяемой памяти и отображения файлов на память. В таких системах пул динамической памяти обычно оказывается разбит на несмежные области. В последующих разделах данной главы мы увидим, что большинство стратегий управления пулом не требуют, чтобы пул был непрерывен.

4.2. Алгоритмы динамического управления памятью

Герой имел привычку складывать окурки в кожаный кипет и употреблять их для изготовления новых самокруток. Таким образом, согласно велению неумолимого закона средних чисел, какую-то часть этого табака он курил в течение многих лет.

T. Пратчетт

При *динамическом выделении памяти* запросы на выделение памяти формируются во время исполнения задачи. Динамическое выделение, таким образом, противопоставляется *статическому*, когда запросы формируются на этапе компиляции программы. В конечном итоге, и те, и другие запросы нередко обрабатываются одним и тем же алгоритмом выделения памяти в ядре ОС. Но во многих случаях статическое выделение можно реализовать намного более простыми способами, чем динамическое. Главное отличие здесь в том, что при статическом выделении кажется неестественной — и поэтому редко требуется — возможность отказаться от ранее выделенной памяти. При динамическом же распределении часто требуется предоставить возможность отказываться от запрошенных блоков так, чтобы освобожденная память могла использоваться для удовлетворения последующих запросов. Таким образом, динамический распределитель (*аллокатор*, от англ. allocator) вместо простой границы между занятой и свободной памятью (которой достаточно в простых случаях статического распределения) вынужден хранить список возможно несвязных областей свободной памяти, называемый *пулом* (pool) или *кучей* (heap).

Многие последовательности запросов памяти и отказов от нее могут привести к тому, что вся доступная память будет разбита на блоки маленького размера, и попытка выделения большого блока завершится неудачей, даже если сумма длин доступных маленьких блоков намного больше требуемой. Это явление называется *фрагментацией памяти* (рис. 4.3). Иногда используют более точный термин — *внешняя фрагментация* (что такое внутренняя фрагментация, будет рассказано далее). Кроме того, большое количество блоков требует длительного поиска. Существует также много мелких трудностей разного рода. К счастью, человечество занимается проблемой распределения памяти уже давно и найдено много хороших или приемлемых решений.

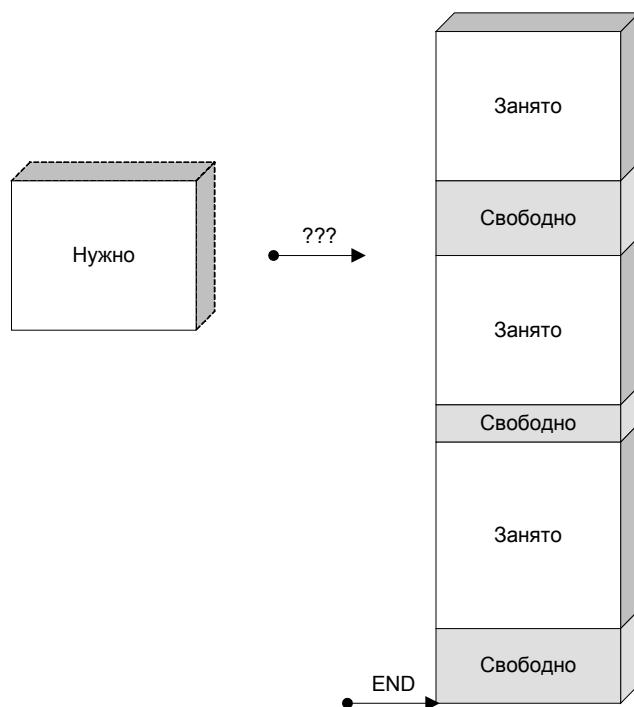


Рис. 4.3. Внешняя фрагментация

В зависимости от решаемой задачи используются различные стратегии поиска свободных блоков памяти. Например, программа может выделять блоки одинакового размера или нескольких фиксированных размеров. Это сильно облегчает решение задач дефрагментации и поиска свободных участков ОЗУ.

Возможны ситуации, когда блоки освобождаются в порядке, обратном тому, в котором они выделялись. Это позволяет свести выделение памяти к стеко-

вой структуре, т. е. фактически вернуться к простому запоминанию границы между занятой и свободной памятью.

Возможны также ситуации, когда некоторые из занятых блоков можно переместить по памяти — тогда есть возможность проводить *дефрагментацию памяти*, перемещение занятых блоков памяти с целью объединить свободные участки. Например, функцию `realloc()` в ранних реализациях системы UNIX можно было использовать именно для этой цели.

В стандартных библиотечных функциях языков высокого уровня, таких как `malloc/free/realloc` в C, `new/dispose` в Pascal и т. д., как правило, используются алгоритмы, рассчитанные на наиболее общий случай: программа запрашивает блоки случайного размера в случайному порядке и освобождает их также случайнным образом.

Впрочем, случайные запросы — далеко не худший вариант. Даже не зная деталей стратегии управления кучей, довольно легко построить программу, которая "испортит жизнь" многим распространенным алгоритмам (пример 4.2). Доказано [Кнут 2000], что для любого алгоритма динамического выделения памяти, который не допускает перемещения выделенных блоков, можно построить такую последовательность запросов на выделение и освобождение памяти, которая приведет к блокировке алгоритма — невозможности удовлетворить запрос, несмотря на то, что общий объем доступной памяти превышает запрашиваемый.

Пример 4.2. Пример последовательности запросов памяти

```
while(TRUE) {  
    void * b1 = malloc(random(10));  
    /* Случайный размер от 0 до 10 байт */  
    void * b2 = malloc(random(10)+10);  
    /* ..... от 10 до 20 байт */  
  
    if(b1 == NULL && b2 == NULL) /* Если памяти нет */  
        break;      /* Выйти из цикла */  
  
    free(b1);  
}  
void * b3 = malloc(150);  
  
/* Скорее всего, память не будет выделена */
```

В результате исполнения такой программы вся доступная память будет "порезана на лапшу": между любыми двумя свободными блоками будет размещен занятый блок меньшего размера (рис. 4.4).

К счастью, пример 4.2 имеет искусственный характер. В реальных программах такая ситуация встречается редко, и часто оказывается проще исправить программу, чем вносить изменения в универсальный алгоритм управления кучей.

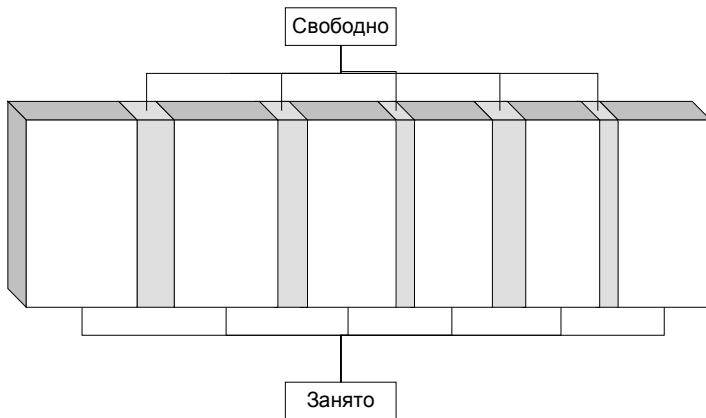


Рис. 4.4. Результат работы программы примера 4.2

Впрочем, необходимо еще раз подчеркнуть, что задача динамического управления памятью (как физическим ОЗУ, так и виртуальным адресным пространством) в общем случае неразрешима: если мы требуем, чтобы выделяемые блоки памяти занимали непрерывный диапазон и не можем перемещать занятые блоки, мы всегда можем построить последовательность запросов, которая заведет наш аллокатор в тупик. При этом оба допущения (непрерывность блоков и невозможность их перемещения) вполне реалистичны. Так, стандартная схема индексации в массивах (сложение индекса и начального адреса) предполагает, что массив непрерывен. Эта стандартная схема привлекательна тем, что обращение к элементам массива происходит за фиксированное (и весьма малое) время. Существуют различные способы построения динамических массивов, способных занимать несмежные участки памяти, но время обращения к элементу такого массива зависит от количества фрагментов, на которые разбит массив, а это далеко не всегда приемлемо.

Далее, выделение памяти всегда предполагает последующее обращение к выделенной памяти, т. е. сохранение указателей на нее. В большинстве языков программирования, в том числе в C/C++ и Pascal, невозможно проследить дальнейшую судьбу этих указателей; даже в тех языках, где указатели во вре-

мя исполнения отличаются от скалярных данных (например, в Java/C#), такое прослеживание оказывается недопустимо дорогой операцией, даже при сборке мусора (*см. разд. 4.3*) не собирается всей необходимой для этого информации. Если выделенный блок памяти используется для хранения кода, то код в момент загрузки будет настроен на эти адреса; большинство загрузчиков отбрасывают таблицу перемещений после загрузки и не допускают последующего перемещения однажды настроенного кода.

Пример 4.2 построен на том предположении, что система выделяет нам блоки памяти, размер которых соответствует запрошенному с точностью до байта. Если же минимальная единица выделения равна 32 байтам, никакой внешней фрагментации наш пример не вызовет: на каждый запрос будет выделяться один блок. Но при этом мы столкнемся с обратной проблемой, которая называется *внутренней фрагментацией*: если система умеет выделять только блоки, кратные 32 байтам, а нам реально нужно 15 или 47 байт, то 17 байт на блок окажутся потеряны (рис. 4.5).

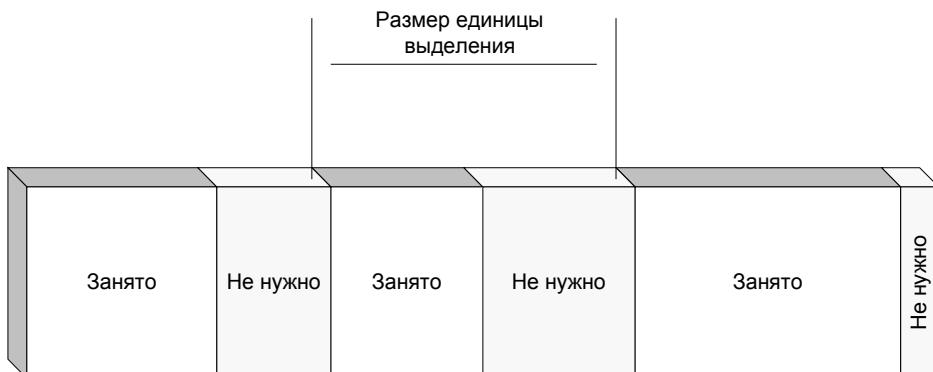


Рис. 4.5. Внутренняя фрагментация

Чем больше размер единицы выделения, тем меньше нам грозит фрагментация внешняя, и тем большие потери обеспечивает фрагментация внутренняя. Величина потерь зависит от среднего размера запрашиваемого блока. Грубая оценка свидетельствует о том, что в каждом блоке в среднем теряется половина единицы выделения, т. е. отношение занятой памяти к потерянной будет $\frac{1}{2} \frac{Ns}{N\bar{l}}$, где N — количество выделенных блоков, s — размер единицы выделения, а \bar{l} — средний размер блока. Упростив эту формулу, мы получим вы-

ражение для величины потерь: $\frac{s}{2l}$, т. е. потери линейно растут с увеличением размера единицы выделения.

Если средний размер блока сравним с единицей выделения, наша формула теряет точность, но все равно дает хорошую оценку порядка величины потерь. Так, если $s = l$, наша формула дает 50% потерь, что вполне согласуется со здравым смыслом: если запрашиваемый блок чуть короче минимально возможного, теряется только это "чуть"; зато если он чуть длиннее, то для него отводится два минимальных блока, один из которых теряется почти весь. Точная величина потерь определяется распределением запрашиваемых блоков по длине (для ее вычисления требуется брать интеграл от функции этого распределения), но я предпочитаю оставить вывод точной формулы любопытному читателю.

Все системы управления памятью (не только оперативной, но и дисковой), которые в той или иной форме допускают размещение объектов в несмежных областях памяти и их перемещение (именно это делают сегментные и страницные диспетчеры памяти, файловые системы и некоторые другие системы управления памятью, которые не будут рассматриваться в этой книге, например, алгоритмы размещения таблиц и индексов реляционных СУБД), также в той или иной форме предполагают наличие минимального блока памяти, который подвергается переадресации и/или перемещению. Поэтому ни одно известное средство борьбы с внешней фрагментацией не может избавиться от внутренней фрагментации.

Варианты алгоритмов распределения памяти исследовались еще в 50-е годы XX века. Итоги многолетнего изучения этой проблемы приведены в [Кнут 2000] и многих других учебниках, и они в определенном смысле неутешительны: хотя есть стратегии управления памятью, которые в каких-то отношениях лучше других, но универсального алгоритма, который не страдал бы от того или иного вида фрагментации и был предсказуем в других отношениях, не существует. Как уже отмечалось, во многих случаях оказывается проще исправить программу (привести ее в соответствие с ожиданиями алгоритма распределения памяти), чем подбирать "правильный" аллокатор. Это один из примеров "самосбывающейся" статистики, когда ориентация ОС или стандартных библиотек языков программирования на определенную модель поведения пользовательских программ приводит к распространению программ, более или менее соответствующих этой статистике.

Обычно все свободные блоки памяти объединяются в двунаправленный связанный список. Список должен быть двунаправленным для того, чтобы из него в любой момент можно было извлечь любой блок. Впрочем, если все

действия по извлечению блока производятся после поиска, то можно слегка усложнить процедуру поиска и всегда сохранять указатель на предыдущий блок. Это решает проблему извлечения и можно ограничиться односторонним списком. Беда только в том, что многие алгоритмы при объединении свободных блоков извлекают их из списка в соответствии с адресом, поэтому для таких алгоритмов двунаправленный список остро необходим.

Поиск в списке может вестись тремя способами: до нахождения *первого подходящего* (first fit) блока, до блока, размер которого ближе всего к заданному — *наиболее подходящего* (best fit), и, наконец, до нахождения самого большого блока, *наименее подходящего* (worst fit).

Использование стратегии worst fit имеет смысл разве что в сочетании с сортировкой списка по убыванию размера. Это может ускорить выделение памяти (всегда берется первый блок, а если он недостаточно велик, мы с чистой совестью можем сообщить, что свободной памяти нет), но создает проблемы при освобождении блоков: время вставки в отсортированный список пропорционально $O(n)$, где n — размер списка.

Помещать блоки в отсортированный массив еще хуже — время вставки становится $O(n + \log(n))$ и появляется ограничение на количество блоков. Использование хэш-таблиц или двоичных деревьев требует накладных расходов и усложнений программы, которые себя в итоге не оправдывают.

В действительности, можно использовать для организации кучи своеобразную структуру данных, которая также называется кучей (heap), — это своего рода дерево, в котором самые большие элементы размещаются не в самой правой ветви (как в сортированных деревьях), а ближе к корню дерева. Такие структуры данных используются в качестве промежуточных при сортировке. Не исключено, что когда-то аллокаторы были устроены именно так; возможно даже, что название "куча" происходит именно от использования таких структур. На практике стратегия worst fit используется при размещении пространства в файловых системах, например в HPFS, но ни одного примера ее использования для распределения оперативной памяти в современных ОС и средах исполнения ЯВУ мне не известно.

Чаще всего применяют несортированный список. Для нахождения наиболее подходящего мы обязаны просматривать весь список, в то время как первый подходящий может оказаться в любом месте, и среднее время поиска будет меньше. Насколько меньше — зависит от отношения количества подходящих блоков к общему количеству. (Читатели, знакомые с теорией вероятности, могут самостоятельно вычислить эту зависимость.)

В общем случае best fit увеличивает фрагментацию памяти. Действительно, если мы нашли блок с размером больше заданного, мы должны отделить "хвост" и пометить его как новый свободный блок. Понятно, что в случае best fit средний размер этого "хвоста" будет маленьким, и мы в итоге получим

большое количество мелких блоков, которые невозможно объединить, т. к. пространство между ними занято.

В тех ситуациях, когда мы размещаем блоки нескольких фиксированных размеров, этот недостаток роли не играет и стратегия best fit может оказаться оправданной. Однако библиотеки распределения памяти рассчитывают на общий случай, и в них обычно используются алгоритмы first fit.

При использовании first fit с линейным двунаправленным списком возникает специфическая проблема. Если каждый раз просматривать список с одного и того же места, то большие блоки, расположенные ближе к началу, будут чаще удаляться. Соответственно, мелкие блоки будут скапливаться в начале списка, что увеличит среднее время поиска (рис. 4.6). Простой способ борьбы с этим явлением состоит в том, чтобы просматривать список то в одном направлении, то в другом. Более радикальный и еще более простой метод заключается в следующем: список делается кольцевым, и каждый поиск начинается с того места, где мы остановились в прошлый раз. В это же место добавляются освободившиеся блоки. В результате список очень эффективно перемешивается и никакой "антисортировки" не возникает.

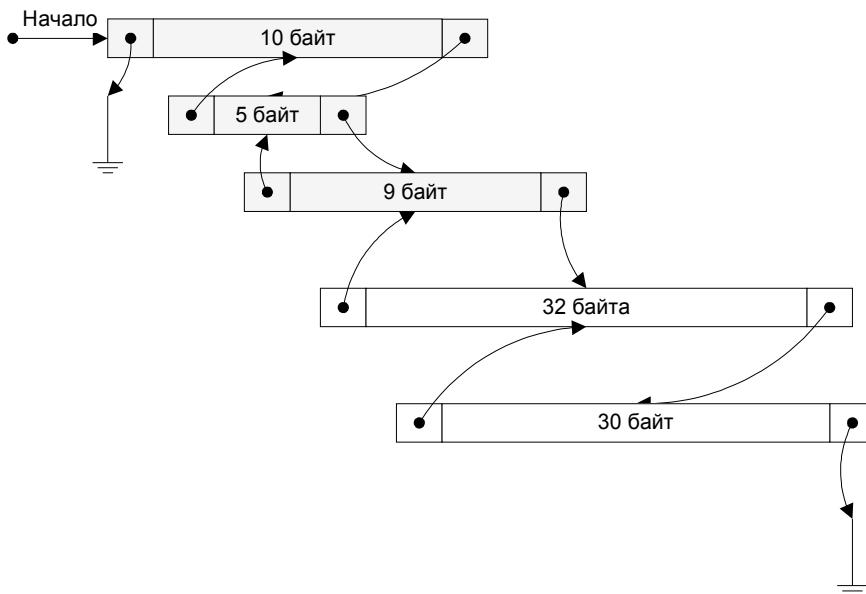


Рис. 4.6. Антисортировка

Разработчик программы динамического распределения памяти обязан решить еще одну важную проблему, а именно — объединение свободных блоков. Действительно, обидно, если мы имеем сто свободных блоков по одному ки-

лобайту и не можем сделать из них один блок в сто килобайт. Но если все эти блоки расположены в памяти один за другим, а мы не можем их при этом объединить, — это просто унизительно.

Кроме того, если мы умеем объединять блоки и видим, что объединенный блок ограничен сверху значением `brklevel`, то мы можем, вместо помещения этого блока в список, просто уменьшить значение `brklevel` и, таким образом, вернуть ненужную память системе.

Представим себе для начала, что все, что мы знаем о блоке, — это его начальный адрес и размер. Легко понять, что это очень плохая ситуация. Действительно, для объединения блока с соседями мы должны найти их в списке свободных или же убедиться, что там их нет. Для этого мы должны просмотреть весь список. Как одну из идей мозгового штурма можно выдвинуть предложение сортировать список свободных блоков по адресу.

Гораздо проще запоминать в дескрипторе блока указатели на дескрипторы соседних блоков. Немного разив эту идею, мы приходим к методу, который называется *алгоритмом парных меток* и состоит в том, что мы добавляем к каждому блоку по два слова памяти. Именно слова, а не байта. Дело в том, что требуется добавить достаточно места, чтобы хранить в нем размер блока в байтах или словах. Обычно такое число занимает столько же места, сколько и адрес, а размер слова обычно равен размеру адреса. На x86 в реальном режиме это не так, но это вообще довольно странный процессор.

Итак, мы добавляем к блоку два слова — одно перед ним, другое после него. В оба слова мы записываем размер блока. Получается своеобразный дескриптор, который окружает блок. При этом мы устанавливаем, что значения длин будут положительными, если блок свободен, и отрицательными, если блок занят. Можно сказать и наоборот, важно только потом соблюдать это соглашение (рис. 4.7).

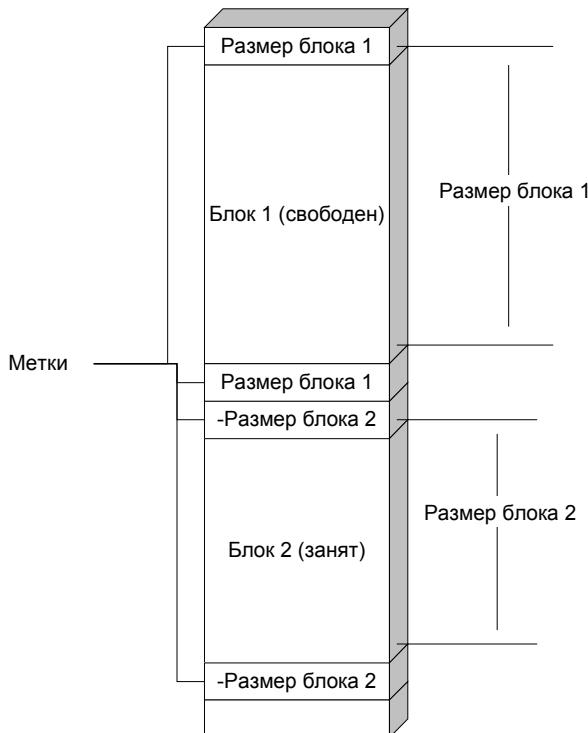


Рис. 4.7. Парные метки

Представим, что мы освобождаем блок с адресом `addr`. Считаем, что `addr` имеет тип `word *`, и при добавлении к нему целых чисел результирующий адрес будет отсчитываться в словах, как в языке С. Для того чтобы проверить, свободен ли сосед перед ним, мы должны посмотреть слово с адресом `addr-2`. Если оно отрицательно, то сосед занят, и мы должны оставить его в покое (рис. 4.8). Если же оно положительно, то мы можем легко определить адрес начала этого блока как `addr-addr[-2]`.



Рис. 4.8. Объединение с использованием парных меток

Определив адрес начала блока, мы можем легко объединить этот блок с блоком `addr`, нам нужно только сложить значения меток-дескрипторов и записать их в дескрипторы нового большого блока. Нам даже не нужно будет добавлять освобождаемый блок в список и извлекать оттуда его соседа!

Похожим образом присоединяется и сосед, стоящий после него. Единственное отличие состоит в том, что этого соседа все-таки нужно извлекать из списка свободных блоков.

Фактически, парные метки можно рассматривать как способ реализации решения, предложенного нами как одна из идей мозгового штурма: двунаправленного списка, включающего в себя как занятые, так и свободные блоки, и отсортированного по адресу. Дополнительное преимущество приведенного алгоритма состоит в том, что мы можем отслеживать такие ошибки, как многократное освобождение одного блока, запись в память за границей блока и иногда даже обращение к уже освобожденному блоку. Действительно, мы в любой момент можем проверить всю цепочку блоков памяти и убедиться в том, что все свободные блоки стоят в списке, что в нем стоят только свободные блоки, что сами цепочка и список не испорчены и т. д.

Примечание

Это действительно большое преимущество, т. к. оно значительно облегчает выявление ошибок работы с указателями, о которых в руководстве по Zortech C/C++ сказано, что "опытные программисты, услышав это слово [ошибка работы с указателями — прим. авт.], бледнеют и прячутся под стол" ([Zortech v3.x]).

Итак, одним из лучших универсальных алгоритмов динамического распределения памяти является *алгоритм парных меток* с объединением свободных блоков в двунаправленный кольцевой список и поиском по принципу first fit. Этот алгоритм обеспечивает приемлемую производительность почти для всех стратегий распределения памяти, используемых в прикладных программах.

Алгоритм парных меток был предложен Дональдом Кнутом в начале 60-х годов XX века.

В третьем издании классической книги [Кнут, 2000] этот алгоритм приводится под названием "освобождения с дескрипторами границ". В современных системах используются и более сложные структуры дескрипторов, но всегда ставится задача обеспечить поиск соседей блока по адресному пространству за фиксированное время. В этом смысле практически все современные подпрограммы динамического выделения памяти (в частности, реализации стандартной библиотеки языка С) используют аналоги алгоритма парных меток или, точнее, отсортированные по адресу двунаправленные списки, в которые включены как свободные, так и занятые блоки. Сортировка по адресу обеспечивает, что блоки, размещенные в памяти рядом, оказываются соседями в этом списке, поэтому поиск соседей освобождаемого блока происходит за константное и, на практике, за очень малое время. Другие известные подходы либо просто хуже, чем этот, либо проявляют свои преимущества только в специальных случаях.

Реализация `malloc` в библиотеке GNU LibC

GNU LibC — это реализация стандартной библиотеки языка С в рамках freeware-проекта GNU Not Unix. Реализация функций `malloc/free` этой библиотеки (пример 4.3) использует смешанную стратегию: блоки размером более 4096 байт выделяются стратегией first fit из двусвязного кольцевого списка с использованием циклического просмотра, а освобождаются с помощью метода, который в указанном ранее смысле похож на алгоритм парных меток. Все выделяемые таким образом блоки будут иметь размер, кратный 4096 байтам.

На каждые 4096 байт пула памяти GNU LibC выделяет специальную запись-дескриптор. Эти записи объединяются в динамический массив `_heapinfo`; поскольку сам этот массив выделяется из той же области памяти, что и пул, управление этим массивом представляет собой довольно сложную задачу, иногда напоминающую вытаскивание себя за волосы из болота в духе барона Мюнхаузена; если читатель желает самостоятельно разобраться в том, как это происходит, я рекомендую самостоятельно проанализировать функцию `morecore` из примера 4.3 и комментарии к нему.

Размещение дескрипторов в `_heapinfo` в точности соответствует размещению соответствующих страниц в пуле. Пересчет указателя в пуле в индекс его дескриптора в примере 4.3 осуществляется макроопределением `BLOCK` по достаточно очевидному алгоритму. Понимание кода несколько затрудняется тем, что слово `block` в идентификаторах и комментариях используется в двух смыслах: для обозначения блоков размером 4096 байт и для обозначения непрерывных областей памяти, образованных такими блоками.

Элементом массива является структура `_malloc_info`, которая требует отдельного обсуждения. Для нефрагментированных блоков эта структура описывает область непрерывной памяти, к которой принадлежит блок: начало и размер этой области и ее статус (занята она или свободна). LibC поддерживает в актуальном состоянии описатели только для первого блока каждой непрерывной области памяти (свободной или занятой). Остальные дескрипторы области используются только как заполнители; впрочем, если область будет поделена на части, то первые из дескрипторов частей будут использоваться по назначению.

Головные дескрипторы всех областей связаны в два двунаправленных списка. В одном списке содержатся все области в пуле (и свободные, и занятые, и фрагментированные). Этот список отсортирован по адресу блока и полностью аналогичен списку, образованному парными метками в классическом алгоритме парных меток; он никогда не просматривается полностью, но используется для поиска соседей блока при его освобождении. Блоки вставляются в этот список при добавлении памяти функцией `morecore` или при делении крупного блока на части, а удаляются — при объединении нескольких соседних свободных блоков.

Второй список кольцевой, несортированный и содержит только свободные блоки. По этому списку при выделении памяти осуществляется поиск по принципу первого подходящего; выделяемые блоки исключаются из этого списка, но если при выделении оказалось необходимо отрезать от блока "хвост", то этот хвост возвращается в список в качестве самостоятельного блока.

Блоки меньшего размера объединяются в очереди с размерами, пропорциональными степеням двойки, как в описанном далее алгоритме близнецов. Элементы этих очередей называются фрагментами (рис. 4.9). В отличие от алгоритма близнецов, мы не объединяем при освобождении парные фрагменты. Вместо этого мы разбиваем 4-килобайтовый блок на фрагменты одинакового размера. Если, например, наша программа сделает запросы на 514 и 296 байт памяти, ей будут переданы фрагменты в 1024 и 512 байт соответственно. Под эти фрагменты будут выделены полные блоки в 4 Кбайта, и внутри них будет выделено по одному фрагменту. При последующих запросах на фрагменты такого же размера будут использоваться свободные фрагменты этих блоков. Пока хотя бы один фрагмент блока занят, весь блок считается занятым. Когда же освобождается последний фрагмент, блок возвращается в пул.

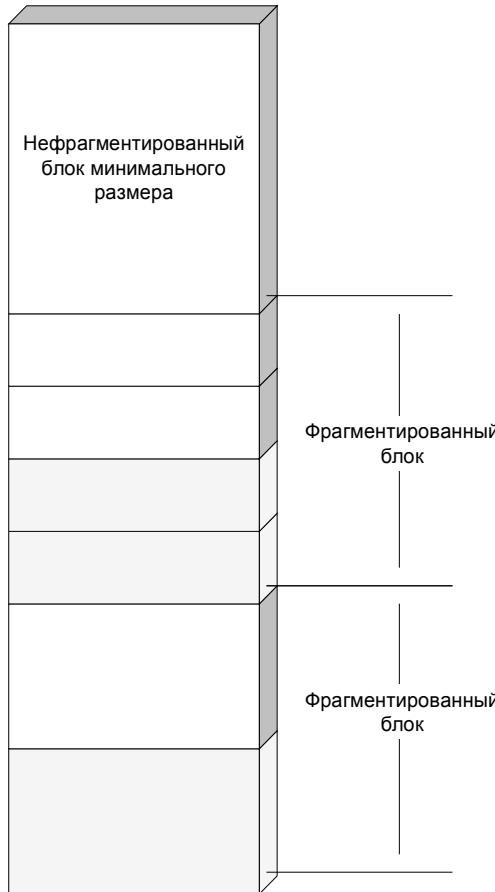


Рис. 4.9. Фрагменты в реализации `malloc` из GNU LibC

Описатели блоков хранятся не вместе с самими блоками, а в отдельном динамическом массиве `_heapinfo`. Описатель заводится не на непрерывную последовательность свободных байтов, а на каждые 4096 байт памяти (в примере 4.3 именно это значение принимает константа `BLOCKSIZE`). Благодаря этому мы можем вычислить индекс описателя в `_heapinfo`, просто разделив на 4096 смещение освобождаемого блока от начала пула.

Для нефрагментированных блоков описатель хранит состояние (занят-свободен) и размер непрерывного участка, к которому принадлежит блок. Благодаря этому, как и в алгоритме парных меток, мы легко можем найти соседей освобождаемого участка памяти и объединить их в большой непрерывный участок.

Для фрагментированных блоков описатель хранит размер фрагмента, счетчик занятых фрагментов и список свободных. Кроме того, все свободные фрагменты одного размера объединены в общий список — заголовки этих списков собраны в массив `_fraghead`.

Используемая структура данных занимает больше места, чем применяемая в классическом алгоритме парных меток, но сокращает объем списка свободных блоков и поэтому имеет более высокую производительность. Средний объем блока, выделяемого современными программами для ОС общего назначения, измеряется многими килобайтами, поэтому в большинстве случаев повышение накладных расходов памяти оказывается терпимо.

**Пример 4.3. Реализация malloc/free в GNU LibC. Функция
__default_morecore приведена в примере 4.1**

malloc.c

/* Распределитель памяти 'malloc'.

Copyright 1990, 1991, 1992 Free Software Foundation

Написана в мае 1989 Mike Haertel.

GNU C Library является свободным программным обеспечением;
вы можете передавать другим лицам и/или модифицировать ее в соответствии
с положениями GNU General Public License версии 2 или (по вашему выбору)
любой более поздней версии.

Библиотека GNU C распространяется в надежде, что она будет полезна, но
БЕЗ КАКИХ-ЛИБО ГАРАНТИЙ; даже без неявно предполагаемых гарантий

КОММЕРЧЕСКОЙ ЦЕННОСТИ или ПРИГОДНОСТИ ДЛЯ КОНКРЕТНОЙ ЦЕЛИ.

Подробнее см. GNU General Public License.

Вы должны были получить копию GNU General Public License вместе с

GNU C Library; см. файл COPYING. Если вы ее не получили, напишите по
адресу: Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA.

С автором можно связаться по электронной почте по адресу
mike@ai.mit.edu, или Mike Haertel c/o Free Software Foundation. */

```
#ifndef _MALLOC_INTERNAL
#define _MALLOC_INTERNAL
#include <malloc.h>
#endif

#ifndef __ELF__
```

```
#pragma weak malloc = __libc_malloc
#endif

/* Как действительно получить дополнительную память. */
__ptr_t (*__morecore) __P ((ptrdiff_t __size))
    = __default_morecore_init;

/* Предоставляемая пользователем отладочная функция (hook) для 'malloc'.
*/
void (*__malloc_initialize_hook) __P ((void));
__ptr_t (*__malloc_hook) __P ((size_t __size));

/* Указатель на основание первого блока. */
char * __heapbase;

/* Таблица информационных записей о блоках. Размещается
через align/_free (не malloc/free). */
malloc_info * __heapinfo;

/* Количество информационных записей. */
static size_t heapsize;

/* Индекс поиска в информационной таблице. */
size_t __heapindex;

/* Ограничитель допустимых индексов в информационной таблице */
size_t __heaplimit;

#if 1          /* Adapted from Mike */
/* Счетчик больших блоков, размещенных для каждого из размеров
фрагментов. */
int _fragblocks[BLOCKLOG];
#endif

/* Списки свободных фрагментов по размерам. */
struct list _fraghead[BLOCKLOG];

/* Инструментальные переменные. */
size_t __chunks_used;
size_t __bytes_used;
size_t __chunks_free;
```

```
size_t _bytes_free;

/* Имеем ли мы опыт? */
int __malloc_initialized;

/* Выровненное размещение. */
static __ptr_t align __P ((size_t));
static __ptr_t
align (size)
    size_t size;
{
    __ptr_t result;
    unsigned long int adj;

    result = (*__morecore) (size);
    adj = (unsigned long int) ((unsigned long int) ((char *) result -
                                                (char *) NULL)) % BLOCKSIZE;
    if (adj != 0)
    {
        adj = BLOCKSIZE - adj;
        (void) (*__morecore) (adj);
        result = (char *) result + adj;
    }
    return result;
}

/* Настроить все и запомнить, что у нас есть. */
static int initialize __P ((void));
static int
initialize ()
{
    if (__malloc_initialize_hook)
        (*__malloc_initialize_hook) ();

    heapsize = HEAP / BLOCKSIZE;
    _heapinfo = (malloc_info *) align (heapsize * sizeof (malloc_info));
    if (_heapinfo == NULL)
        return 0;
    memset (_heapinfo, 0, heapsize * sizeof (malloc_info));
    _heapinfo[0].free.size = 0;
```

```
_heapinfo[0].free.next = _heapinfo[0].free.prev = 0;
_heapindex = 0;
_heapbase = (char *) _heapinfo;
_malloc_initialized = 1;
return 1;
}

/* Получить выровненную память, инициализируя
или наращивая таблицу описателей кучи по мере необходимости. */
static __ptr_t morecore __P ((size_t));
static __ptr_t
morecore (size)
    size_t size;
{
    __ptr_t result;
    malloc_info *newinfo, *oldinfo;
    size_t newsize;

    result = align (size);
    if (result == NULL)
        return NULL;

    /* Проверить, нужно ли нам увеличить таблицу описателей. */
    if ((size_t) BLOCK ((char *) result + size) > heapsize)
    {
        newsize = heapsize;
        while ((size_t) BLOCK ((char *) result + size) > newsize)
            newsize *= 2;
        newinfo = (malloc_info *) align (newsize * sizeof (malloc_info));
        if (newinfo == NULL)
        {
            (*__morecore) (-size);
            return NULL;
        }
        memset (newinfo, 0, newsize * sizeof (malloc_info));
        memcpy (newinfo, _heapinfo, heapsize * sizeof (malloc_info));
        oldinfo = _heapinfo;
        newinfo[BLOCK (oldinfo)].busy.type = 0;
        newinfo[BLOCK (oldinfo)].busy.info.size
            = BLOCKIFY (heapsize * sizeof (malloc_info));
    }
}
```

```
_heapinfo = newinfo;
_free_internal (oldinfo);
heapsize = newsize;
}

_heaplimit = BLOCK ((char *) result + size);
return result;
}

/* Выделить память из кучи. */
__ptr_t
__libc_malloc (size)
    size_t size;
{
    __ptr_t result;
    size_t block, blocks, lastblocks, start;
    register size_t i;
    struct list *next;

/* Некоторые программы вызывают malloc (0). Мы это допускаем. */
#endif 0
if (size == 0)
    return NULL;
#endif

if (!__malloc_initialized)
    if (!initialize ())
        return NULL;

if (__malloc_hook != NULL)
    return (*__malloc_hook) (size);

if (size < sizeof (struct list))
    size = sizeof (struct list);

/* Определить политику размещения на основании размера запроса. */
if (size <= BLOCKSIZE / 2)
{
    /* Маленькие запросы получают фрагмент блока.
Определяем двоичный логарифм размера фрагмента. */

```

```
register size_t log = 1;
--size;
while ((size /= 2) != 0)
    ++log;

/* Просмотреть списки фрагментов на предмет свободного
   фрагмента желаемого размера. */
next = _fraghead[log].next;
if (next != NULL)
{
    /* Найдены свободные фрагменты этого размера.
       Вытолкнуть фрагмент из списка фрагментов и вернуть его.
       Обновить счетчики блока nfree и first. */
    result = (_ptr_t) next;
    next->prev->next = next->next;
    if (next->next != NULL)
        next->next->prev = next->prev;
    block = BLOCK (result);
    if (--_heapinfo[block].busy.info.frag.nfree != 0)
        _heapinfo[block].busy.info.frag.first = (unsigned long int)
            ((unsigned long int) ((char *) next->next - (char *) NULL)
             % BLOCKSIZE) >> log;

    /* Обновить статистику. */
    ++_chunks_used;
    _bytes_used += 1 << log;
    --_chunks_free;
    _bytes_free -= 1 << log;
}
else
{
    /* Нет свободных фрагментов желаемого размера. Следует взять
       новый блок, поделить его на фрагменты и вернуть первый. */
    result = __libc_malloc (BLOCKSIZE);
    if (result == NULL)
        return NULL;
#endif /* Adapted from Mike */
++_fragblocks[log];
#endif /* Связать все фрагменты, кроме первого, в список свободных. */
```

```
for (i = 1; i < (size_t) (BLOCKSIZE >> log); ++i)
{
    next = (struct list *) ((char *) result + (i << log));
    next->next = _fraghead[log].next;
    next->prev = &_fraghead[log];
    next->prev->next = next;
    if (next->next != NULL)
        next->next->prev = next;
}

/* Инициализировать счетчики nfree и first для этого блока. */
block = BLOCK (result);
_heapinfo[block].busy.type = log;
_heapinfo[block].busy.info.frag.nfree = i - 1;
_heapinfo[block].busy.info.frag.first = i - 1;

_chunks_free += (BLOCKSIZE >> log) - 1;
_bytes_free += BLOCKSIZE - (1 << log);
_bytes_used -= BLOCKSIZE - (1 << log);
}

}
else
{
/* Большие запросы получают один или больше блоков.

Просмотреть список свободных циклически, начиная с точки, где мы
были в последний раз. Если мы пройдем полный круг, не обнаружив
достаточно большого блока, мы должны будем запросить еще память
у системы. */

blocks = BLOCKIFY (size);
start = block = _heapindex;
while (_heapinfo[block].free.size < blocks)
{
    block = _heapinfo[block].free.next;
    if (block == start)
    {
        /* Необходимо взять больше [памяти] у системы.

Проверить, не будет ли новая память продолжением
последнего свободного блока; если это так, нам не
надо будет запрашивать так много. */

block = _heapinfo[0].free.prev;
```

```
lastblocks = _heapinfo[block].free.size;
if (_heaplimit != 0 && block + lastblocks == _heaplimit &&
    (*__morecore) (0) == ADDRESS (block + lastblocks) &&
    (morecore ((blocks - lastblocks) * BLOCKSIZE)) != NULL)
{
#endif 1
/* Adapted from Mike */
/* Обратите внимание, что morecore() может изменить
   положение последнего блока, если она перемещает
   таблицу дескрипторов и старая копия таблицы
   слиивается с последним блоком. */
block = _heapinfo[0].free.prev;
_heapinfo[block].free.size += blocks - lastblocks;
#else
_heapinfo[block].free.size = blocks;
#endif
_bytes_free += (blocks - lastblocks) * BLOCKSIZE;
continue;
}
result = morecore (blocks * BLOCKSIZE);
if (result == NULL)
    return NULL;
block = BLOCK (result);
_heapinfo[block].busy.type = 0;
_heapinfo[block].busy.info.size = blocks;
++_chunks_used;
_bytes_used += blocks * BLOCKSIZE;
return result;
}
}

/*
* В этой точке мы [так или иначе] нашли подходящую запись
* в списке свободных. Понять, как удалить то, что нам нужно,
* из списка. */
result = ADDRESS (block);
if (_heapinfo[block].free.size > blocks)
{
/* Блок, который мы нашли, имеет небольшой остаток,
   так что присоединить его задний конец к списку свободных. */
_heapinfo[block + blocks].free.size
= _heapinfo[block].free.size - blocks;
```

```

        _heapinfo[block + blocks].free.next
        = _heapinfo[block].free.next;
    _heapinfo[block + blocks].free.prev
        = _heapinfo[block].free.prev;
    _heapinfo[_heapinfo[block].free.prev].free.next
        = _heapinfo[_heapinfo[block].free.next].free.prev
        = _heapindex = block + blocks;
    }
else
{
    /* Блок точно соответствует нашему запросу, поэтому
       просто удалить его из списка. */
    _heapinfo[_heapinfo[block].free.next].free.prev
        = _heapinfo[block].free.prev;
    _heapinfo[_heapinfo[block].free.prev].free.next
        = _heapindex = _heapinfo[block].free.next;
    --_chunks_free;
}

_heapinfo[block].busy.type = 0;
_heapinfo[block].busy.info.size = blocks;
++_chunks_used;
_bytes_used += blocks * BLOCKSIZE;
_bytes_free -= blocks * BLOCKSIZE;
}

return result;
}

free.c:
/* Освободить блок памяти, выделенный 'malloc'.
   Copyright 1990, 1991, 1992 Free Software Foundation
   Написано в мае 1989 Mike Haertel.
```

GNU C Library является свободным программным обеспечением;
 вы можете перераспространять ее и/или модифицировать ее в соответствии
 с положениями GNU General Public License версии 2 или (по вашему выбору)
 любой более поздней версии.

Библиотека GNU C распространяется в надежде, что она будет полезна, но

БЕЗ КАКИХ-ЛИБО ГАРАНТИЙ; даже без неявно предполагаемых гарантий

КОММЕРЧЕСКОЙ ЦЕННОСТИ или ПРИГОДНОСТИ ДЛЯ КОНКРЕТНОЙ ЦЕЛИ.

Подробнее см. GNU General Public License.

С автором можно связаться по электронной почте по адресу
mike@ai.mit.edu, или Mike Haertel c/o Free Software Foundation. */

```
#ifndef _MALLOC_INTERNAL
#define _MALLOC_INTERNAL
#include <malloc.h>
#endif

#ifndef __ELF__
#pragma weak free = __libc_free
#endif

/* Предоставляемая пользователем отладочная функция (hook) для 'free'. */
void (*__free_hook) __P ((__ptr_t __ptr));

/* Список блоков, выделенных memalign. */
struct alignlist *_aligned_blocks = NULL;

/* Вернуть память в кучу. Аналогична 'free', но не вызывает
   __free_hook, даже если он определен. */
void
__free_internal (__ptr_t ptr)
{
    int type;
    size_t block, blocks;
    register size_t i;
    struct list *prev, *next;

    block = BLOCK (ptr);

    type = _heapinfo[block].busy.type;
    switch (type)
    {
        case 0:
```

```
/* Собрать как можно больше статистики как можно раньше. */
--_chunks_used;
_bytes_used -= _heapinfo[block].busy.info.size * BLOCKSIZE;
_bytes_free += _heapinfo[block].busy.info.size * BLOCKSIZE;

/* Найти свободный кластер, предшествующий этому в списке
свободных.

Начать поиск с последнего блока, к которому было обращение.

Это может быть преимуществом для программ, в которых выделение
локальное. */

i = _heapindex;
if (i > block)
    while (i > block)
        i = _heapinfo[i].free.prev;
else
{
    do
        i = _heapinfo[i].free.next;
    while (i > 0 && i < block);
    i = _heapinfo[i].free.prev;
}

/* Определить, как включить этот блок в список свободных. */
if (block == i + _heapinfo[i].free.size)
{
    /* Слить этот блок с его предшественником. */
    _heapinfo[i].free.size += _heapinfo[block].busy.info.size;
    block = i;
}
else
{
    /* Действительно включить этот блок в список свободных. */
    _heapinfo[block].free.size = _heapinfo[block].busy.info.size;
    _heapinfo[block].free.next = _heapinfo[i].free.next;
    _heapinfo[block].free.prev = i;
    _heapinfo[i].free.next = block;
    _heapinfo[_heapinfo[block].free.next].free.prev = block;
    ++_chunks_free;
}

/* Теперь, поскольку блок включен, проверить, не можем ли мы
```

```
слить его с его последователем (исключая его последователя из списка и складывая размеры). */
```

```
if (block + _heapinfo[block].free.size ==
_heapinfo[block].free.next)
{
    _heapinfo[block].free.size
    += _heapinfo[_heapinfo[block].free.next].free.size;
    _heapinfo[block].free.next
    = _heapinfo[_heapinfo[block].free.next].free.next;
    _heapinfo[_heapinfo[block].free.next].free.prev = block;
    --_chunks_free;
}

/* Проверить, не можем ли мы вернуть память системе. */

blocks = _heapinfo[block].free.size;
if (blocks >= FINAL_FREE_BLOCKS && block + blocks == _heplimit
    && (*__morecore) (0) == ADDRESS (block + blocks))
{
    register size_t bytes = blocks * BLOCKSIZE;
    _heplimit -= blocks;
    (*__morecore) (-bytes);
    _heapinfo[_heapinfo[block].free.prev].free.next
    = _heapinfo[block].free.next;
    _heapinfo[_heapinfo[block].free.next].free.prev
    = _heapinfo[block].free.prev;
    block = _heapinfo[block].free.prev;
    --_chunks_free;
    _bytes_free -= bytes;
}

/* Установить следующий поиск, стартовать с этого блока. */

_heapindex = block;
break;

default:
/* Собрать некоторую статистику. */
--_chunks_used;
_bytes_used -= 1 << type;
```

```
++_chunks_free;
_bytes_free += 1 << type;

/* Получить адрес первого свободного фрагмента в этом блоке. */
prev = (struct list *) ((char *) ADDRESS (block) +
(_heapinfo[block].busy.info.frag.first << type));

#endif /* Adapted from Mike */
if (_heapinfo[block].busy.info.frag.nfree == (BLOCKSIZE >> type) - 1
&& _fragblocks[type] > 1)
#else
if (_heapinfo[block].busy.info.frag.nfree == (BLOCKSIZE >> type) - 1)
#endif
{
    /* Если все фрагменты этого блока свободны, удалить их из
       списка фрагментов и освободить полный блок. */
#if 1 /* Adapted from Mike */
--_fragblocks[type];
#endif
next = prev;
for (i = 1; i < (size_t) (BLOCKSIZE >> type); ++i)
    next = next->next;
prev->prev->next = next;
if (next != NULL)
    next->prev = prev->prev;
_heapinfo[block].busy.type = 0;
_heapinfo[block].busy.info.size = 1;

/* Следим за точностью статистики. */
++_chunks_used;
_bytes_used += BLOCKSIZE;
_chunks_free -= BLOCKSIZE >> type;
_bytes_free -= BLOCKSIZE;

__libc_free (ADDRESS (block));
}

else if (_heapinfo[block].busy.info.frag.nfree != 0)
{
    /* Если некоторые фрагменты этого блока свободны, включить
       этот фрагмент в список фрагментов после первого свободного
       фрагмента этого блока. */
}
```

```
next = (struct list *) ptr;
next->next = prev->next;
next->prev = prev;
prev->next = next;
if (next->next != NULL)
    next->next->prev = next;
++_heapinfo[block].busy.info.frag.nfree;
}
else
{
    /* В этом блоке нет свободных фрагментов, поэтому включить
       этот фрагмент в список фрагментов и анонсировать, что это
       первый свободный фрагмент в этом блоке. */
    prev = (struct list *) ptr;
    _heapinfo[block].busy.info.frag.nfree = 1;
    _heapinfo[block].busy.info.frag.first = (unsigned long int)
        ((unsigned long int) ((char *) ptr - (char *) NULL)
            % BLOCKSIZE >> type);
    prev->next = _fraghead[type].next;
    prev->prev = &_fraghead[type];
    prev->prev->next = prev;
    if (prev->next != NULL)
        prev->next->prev = prev;
}
break;
}

/* Вернуть память в кучу. */
void
__libc_free (ptr)
    __ptr_t ptr;
{
register struct alignlist *l;

if (ptr == NULL)
    return;

for (l = _aligned_blocks; l != NULL; l = l->next)
    if (l->aligned == ptr)
```

```

{
    l->aligned = NULL;
    /* Пометить элемент списка как свободный. */
    ptr = l->exact;
    break;
}

if (_free_hook != NULL)
    (*_free_hook) (ptr);
else
    _free_internal (ptr);
}

#include <gnu-stabs.h>
#ifndef elf_alias
elf_alias (free, cfree);
#endif

```

К основным недостаткам алгоритма first fit относится невозможность оценки времени поиска подходящего блока, что делает его неприемлемым для задач реального времени. Для этих задач требуется алгоритм, который способен за фиксированное (желательно, небольшое) время либо найти подходящий блок памяти, либо дать обоснованный ответ о том, что подходящего блока не существует.

Проще всего решить эту задачу, если нам требуются блоки нескольких фиксированных размеров (рис. 4.10). Мы объединяем блоки каждого размера в свой список. Если в списке блоков требуемого размера ничего нет, мы смотрим в список блоков большего размера. Если там что-то есть, мы разрезаем этот блок на части, одну отдаем запрашивающей программе, а вторую... Правда, если размеры требуемых блоков не кратны друг другу, что мы будем делать с остатком?

Для решения этой проблемы нам необходимо ввести какое-либо ограничение на размеры выделяемых блоков. Например, можно потребовать, чтобы эти размеры равнялись числам Фибоначчи (последовательность целых чисел, в которой $F_{i+1} = F_i + F_{i-1}$). В этом случае, если нам нужно F_i байт, а в наличии есть только блок размера F_{i+1} , мы легко можем получить два блока — один требуемого размера, а другой — F_{i-1} , который тоже не пропадет. Да, любое ограничение на размер приведет к внутренней фрагментации, но так ли велика эта плата за гарантированное время поиска блока?

На практике числа Фибоначчи не используются. Одной из причин, по-видимому, является относительная сложность вычисления такого F_i , которое не меньше требуемого размера блока. Другая причина — сложность объединения свободных блоков со смежными адресами в блок большего размера. Зато широкое применение нашел алгоритм, который ограничивает последовательные размеры блоков более простой зависимостью — степенями числа 2: 512 байт, 1 Кбайт, 2 Кбайт и т. д. Такая стратегия называется *алгоритмом близнецов* (рис. 4.11).

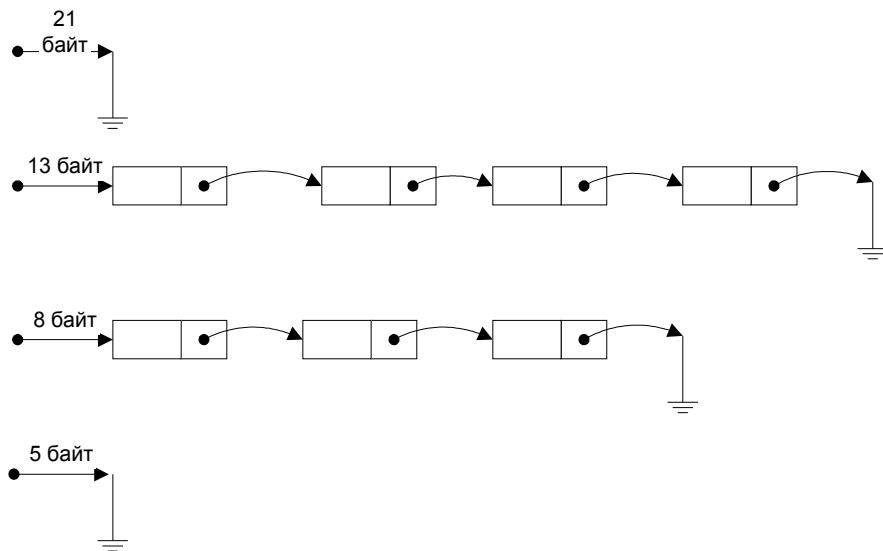


Рис. 4.10. Выделение блоков фиксированных размеров

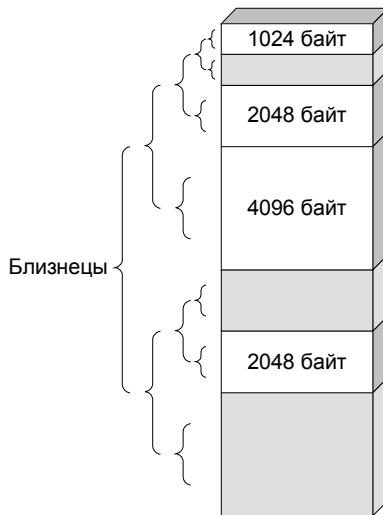


Рис. 4.11. Алгоритм близнецов

Одно из преимуществ этого метода состоит в простоте объединения блоков при их освобождении. Адрес блока-близнеца получается простым инвертированием соответствующего бита в адресе нашего блока. Нужно только проверить, свободен ли этот близнец. Если он свободен, то мы объединяем братьев в блок вдвое большего размера и т. д. Даже в наихудшем случае время поиска не превышает $O(\log(S_{\max}) - \log(S_{\min}))$, где S_{\max} и S_{\min} обозначают, соответственно, максимальный и минимальный размеры используемых блоков. Это делает алгоритм близнецов трудно заменимым для ситуаций, в которых необходимо гарантированное время реакции — например, для задач реального времени. Часто этот алгоритм или его варианты используются для выделения памяти внутри ядра ОС. Например, именно таким способом выделяет память функция `kmalloc` в ядре Linux.

Существуют и более сложные варианты применения описанного ранее подхода. Например, пул свободной памяти Novell Netware состоит из 4 очередей с шагом 16 байт (для блоков размерами 16, 32, 48, 64 байта), 3 очередей с шагом 64 байта (для блоков размерами 128, 192, 256 байт) и 15 очередей с шагом 256 байт (от 512 байт до 4 Кбайт). При запросах большего размера выделяется целиком страница. Любопытно, что возможности работы в режиме реального времени, присущие этой изощренной стратегии, в Netware практически не используются.

Например, если драйвер сетевого интерфейса при получении очередного пакета данных обнаруживает, что у него нет свободных буферов для его приема, он не пытается выделить новый буфер стандартным алгоритмом. Вместо

этого, драйвер просто игнорирует пришедшие данные, лишь увеличивая счетчик потерянных пакетов. Отдельный системный процесс следит за состоянием этого счетчика и только при превышении им некоторого порога за некоторый интервал времени выделяет драйверу новый буфер.

Подобный подход к пользовательским данным может показаться циничным, но надо вспомнить, что при передаче данных по сети возможны и другие причины потери пакетов, например порча данных из-за электромагнитных помех. Поэтому все сетевые протоколы высокого уровня предусматривают средства пересылки пакетов в случае их потери, какими бы причинами эта потеря ни была вызвана. С другой стороны, в системах реального времени игнорирование данных, которые мы все равно не в состоянии принять и обработать, — довольно часто используемая, хотя и не всегда приемлемая стратегия.

Существует еще один способ обеспечить перераспределение памяти между пулами для блоков разного размера — так называемые слабовые аллокаторы [Bonwick 1994]. При их использовании пул разбивают на большие блоки, называемые слабами (slab — англ. бруск, плита), каждый из которых содержит некоторое количество записей требуемого размера. Когда программа запрашивает память, аллокатор пытается найти свободную запись в уже частично занятых слабах; если это невозможно, выделяются новые слабы. Допускается возможность возврата полностью свободных слабов системе, в том числе и их повторное использование для слабов других типов.

Такая стратегия особенно эффективна в условиях, когда программа создает большое число объектов небольшого размера, и при этом размер объекта не кратен степени двойки. Типичными примерами являются буфера под сетевые пакеты (у Ethernet максимальный размер пакета составляет 1500 байт) и другие структуры данных ядра — дескрипторы открытых файлов, дескрипторы процессов и т. д.

Слабовый аллокатор в ядре Linux

Слабы появились в ядре Linux 2.4, несомненно, под впечатлением от результатов, которые дал переход к этой схеме управления памятью в ядре Solaris. Слабовый аллокатор управляет памятью с помощью именованных кэшей (cache), каждый из которых представляет пул блоков памяти определенного типа. Пул создается функцией

```
kmem_cache_t * kmem_cache_create (const char * name, size_t size,  
size_t offset, unsigned long flags, void (*ctor) (void*,  
kmem_cache_t *, unsigned long), void (*dtor) (void*, kmem_cache_t  
*, unsigned long));
```

Смысл большинства параметров очевиден из их названий:

name — имя кэша;

size — размер элемента в байтах;

offset — смещение слаба относительно границы страницы;

flags — дополнительные параметры (флаги);

ctor и dtor — указатели на функции — конструктор и деструктор, которые надо вызвать для каждого элемента слаба. Они вызываются не в момент выделения памяти, а в момент разметки слаба (ctor) и в момент возврата памяти из-под свободного слаба системе (dtor).

Для выделения памяти из кэша используется функция:

```
void * kmem_cache_alloc (kmem_cache_t * cachep, int flags);
```

Параметр flags представляет собой комбинацию нескольких битовых полей, которые позволяют задать приоритет обращения к памяти. Основные уровни приоритета — это GFP_BUFFER, GFP_KERNEL и GFP_ATOMIC. Наиболее высокий уровень приоритета — это GFP_ATOMIC, при котором аллокатор возвращает только блоки из уже размеченных слабов и не пытается увеличивать размер кэша. Запросы к памяти с этим приоритетом с довольно высокой вероятностью завершаются неудачей (хотя система стремится поддерживать определенное количество памяти, доступной для таких запросов), но завершаются за фиксированное время и, благодаря этому, могут вызываться из обработчиков прерываний.

Кэш представляет собой список слабов, каждый из которых разбит на элементарные блоки указанного размера. В зависимости от размера блока, используются две схемы разметки: on-slab (когда метаинформация о том, какие блоки в слабе свободны, а какие — нет, хранится в самом слабе) и off-slab, когда эта информация хранится в отдельных областях памяти (рис. 4.12).

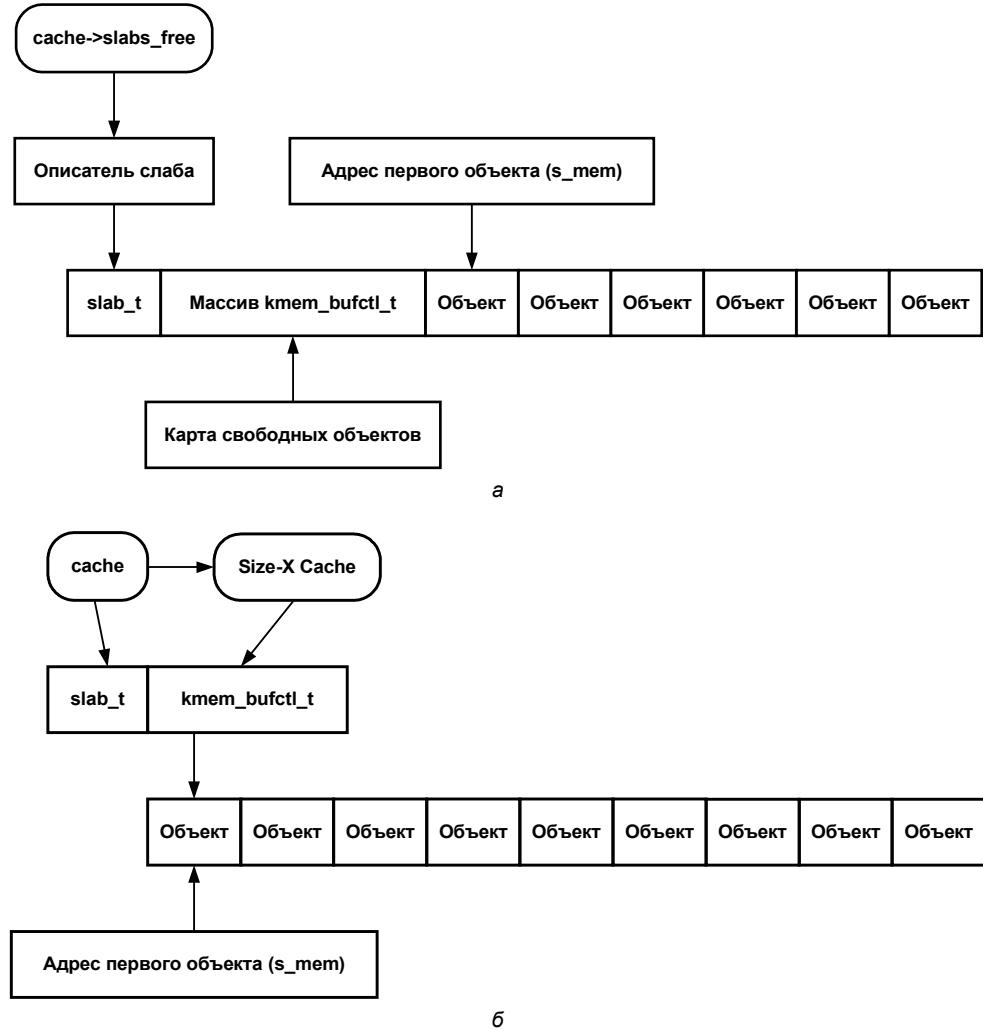


Рис. 4.12. On-slab (а) и off-slab (б) метаданные в ядре Linux 2.4

Аллокатор поддерживает три списка слабов: полностью занятые, частично свободные и полностью свободные, но размеченные. При запросе на создание объекта аллокатор всегда сначала проверяет доступность частично занятых слабов.

При нехватке свободных блоков в размеченных слабах менеджер кэша размещает новые слабы. Обычно это происходит при запросах с приоритетами GFP_BUFFER и GFP_KERNEL. При нехватке памяти ядро может потребовать у менеджера памяти сжатия кэша, при котором кэш возвращает системе полностью свободные слабы.

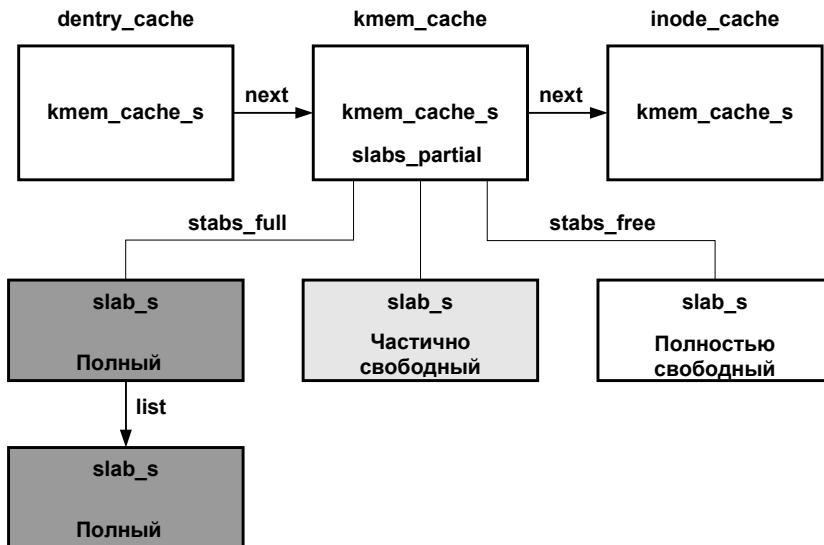


Рис. 4.13. Списки слабов в ядре Linux

4.3. Сборка мусора

— Ладно, — смирился Герера. — Эй, Стелмэн, лучше по-прежнему держись за мое плечо. Не стоит нам разделяться.
 — А я и так держусь, — отозвался Стелмэн.
 — Слушай, кажется, мне лучше знать, держится кто-нибудь за мое плечо или нет.
 — Это твое плечо, Пакстон?
 — Нет, — ответил Пакстон.
 — Плохо, — сказал Стелмэн очень медленно. — Это совсем плохо.
 — Почему?
 — Потому что я определенно держусь за чье-то плечо.

P. Шекли

Явное освобождение динамически выделенной памяти применяется во многих системах программирования и потому привычно для большинства программистов, но оно имеет серьезный недостаток: если программист по какой-то причине не освобождает выделенные блоки, память будет теряться. Эта ошибка, называемая *утечкой памяти* (*memory leak*), особенно неприятна в программах, которые длительное время работают без перезапуска, — подсистемах ядра ОС, постоянно запущенных сервисах или серверных приложениях. Дополнительная неприятность состоит в том, что медленные утечки могут привести к заметным потерям памяти лишь при многодневной или даже

многомесячной непрерывной эксплуатации системы, поэтому их сложно обнаружить при тестировании.

Напротив, чрезмерное увлечение программиста освобождением объектов может привести к еще более серьезной проблеме — *висячим ссылкам* (*dangling reference*). Висячие ссылки возникают, когда мы уничтожаем объект, на который где-то в другом месте сохранена ссылка или — из другой перспективы — когда мы сохраняем ссылки на объект, который уже уничтожен.

Поиск висячих ссылок существенно затрудняется тем фактом, что далеко не всегда память из-под объекта переиспользуется сразу или вскоре после его уничтожения. Поэтому некоторое время чтения по висячей ссылке могут возвращать осмысленные значения, а модификации — проходить без катастрофических последствий. Однако рано или поздно в освобожденной памяти будет создан другой объект. Если старый или новый объект сами содержали указатели на другие объекты, модификации данных по висячей ссылке могут привести к цепному разрушению значительной части или даже всех структур данных, размещенных в памяти нашего процесса. В системах с открытой памятью это обычно приводит к разрушению структур данных, кода других задач и самой ОС, а также к "зависанию" системы. Даже если используется защита памяти, как правило, эта защита срабатывает существенно позже момента, в который была совершена ошибка, поэтому по посмертному дампу программы не всегда можно определить, в какой момент все началось и что вообще произошло.

Главная трудность состоит в индетерминизме — то, насколько рано память будет переиспользована, решающим образом зависит от сценария выполнения программы. Поэтому такие ошибки очень сложно, зачастую невозможно найти при отладке и тестировании программы. Даже когда на такую ошибку натыкается заказчик программы при ее эксплуатации, разработчик далеко не всегда может воспроизвести условия ее возникновения.

Ряд систем программирования предоставляют средства для борьбы с висячими ссылками. Так, отладочная версия стандартной библиотеки Microsoft Visual C предоставляет специальные реализации функций `malloc/free`. При освобождении памяти функция `free` прописывает освобождаемый блок специальным битовым паттерном `0xDD`. В других ситуациях неиспользуемая память прописывается другими паттернами, которые перечислены в табл. 4.1

Таблица 4.1. Битовые паттерны, используемые отладочной библиотекой MS VC

Значение	Расшифровка	Описание
0xCD	Clean Memory	Память, выделенная <code>malloc/new</code> , еще не модифицированная приложением

0xDD	Dead Memory	Память, освобожденная с помощью delete/free. Это может использоваться для обнаружения записи по висячему указателю
------	-------------	--

Таблица 4.1 (окончание)

Значение	Расшифровка	Описание
0xFD	Fence Memory	Известно также как "ничейная земля" ("no mans land"). Используется для ограничения выделенных блоков памяти, как ограждение. Может использоваться для обнаружения выхода индекса за границы массива
0xAB	(Allocated Block?)	Память, выделенная LocalAlloc()
0xBAADF00D	Bad Food	Память, выделенная LocalAlloc() с флагом LMEM_FIXED
0xCC		В коде, компилированном с ключом /GZ, неинициализированные переменные прописываются на байтовом уровне этим шаблоном.

При выделении памяти `malloc` проверяет целостность паттерна во вновь выделяемом блоке. Разумеется, это сильно снижает производительность, но благодаря этому многие ошибки, связанные с висячими и неинициализированными указателями, можно обнаружить гораздо раньше. Впрочем, все равно обнаружение происходит не непосредственно в момент ошибки, поэтому рассматривать такие средства как панацею нельзя.

Некоторые системы программирования используют специальный метод освобождения динамической памяти, называемый *сборкой мусора* (garbage collection). Этот метод состоит в том, что ненужные блоки памяти не освобождаются явным образом. Вместо этого используется некоторый более или менее изощренный алгоритм, следящий за тем, какие блоки еще нужны, а какие — уже нет.

Это приводит к значительному снижению стоимости разработки — по моему опыту, время разработки приблизительно одинаковых по функциональности программ на C/C++ (языки программирования с явным освобождением памяти) и на Java (язык с неявным освобождением памяти) различается в 2-3 раза. Причем, поскольку Java использует синтаксис, во многом похожий на C/C++, можно не сомневаться, что дело здесь именно в способе управления памятью, а не в синтаксисе. В литературе и на форумах мне встречались оценки, из которых следует, что разница может достигать и 4-5 раз. Разумеется, это сильно зависит от разработчика и от решаемой задачи, но, так или иначе, разница

впечатляющая. К сожалению, как мы увидим далее в этом разделе, это преимущество достигается вовсе не бесплатно.

Все методы сборки мусора так или иначе сводятся к поддержанию базы данных о том, какие объекты на кого ссылается. Поэтому применять сборку мусора возможно практически только в интерпретируемых языках, таких, как Java, Smalltalk или Lisp, где с каждой операцией можно ассоциировать неограниченно большое количество действий. Впрочем, в некоторых компилируемых языках, таких, как C++, в которых можно переопределять операции присваивания и связывать определенный код с операциями создания и уничтожения объектов, при желании также можно реализовать сборку мусора для некоторых классов объектов.

4.3.1. Подсчет ссылок

Самый простой метод отличать используемые блоки от ненужных — считать, что блок, на который есть ссылка, нужен, а блок, на который ни одной ссылки не осталось, — не нужен. Для этого к каждому блоку присоединяют дескриптор, в котором подсчитывают количество ссылок на него. Каждая передача указателя на этот блок приводит к увеличению счетчика ссылок на 1, а каждое уничтожение объекта, содержащего указатель, — к уменьшению.

"Уходящий последним гасит свет" — когда при уничтожении ссылки счетчик оказывается равен нулю, блок памяти удаляется. В объектно-ориентированных языках перед этим может быть вызван деструктор соответствующего объекта. Нередко этот способ освобождения памяти не называют сборкой мусора; для него есть специальное название — *подсчет ссылок* (reference counting).

Необходимо отметить важное преимущество подсчета ссылок — при нем удаление объектов и, если это необходимо, вызов деструкторов происходит точно в момент удаления последней ссылки на такой объект, поэтому данный метод используется для управления не только памятью, но и другими дорогостоящими ресурсами. Например, если объект содержал описатель окна пользовательского интерфейса, то при его уничтожении окно можно закрыть — оно точно уже не понадобится программе.

Сборка мусора подсчетом ссылок используется вialectах языка Basic (в Visual Basic вплоть до версии 6; в VB 7 перешли к просмотру ссылок, который рассматривается далее в этом разделе), в языково-независимом объектном стандарте COM (Common Object Model — общая объектная модель) и в ряде других интерпретируемых языков программирования. Существуют также реализации подсчета ссылок на C++, так называемые смартпойнтеры (smart pointer, дословно "умный указатель"); есть ряд библиотек шаблонов,

которые реализуют подсчет ссылок для произвольных объектов. В частности, такой шаблон включен в ATL (Advanced Template Library — продвинутая библиотека шаблонов) компании Microsoft; эта библиотека, помимо прочего, используется для работы с объектами СОМ из языка C.

Далее в этой книге мы увидим несколько примеров использования подсчета ссылок для целей, отличающихся от задач управления оперативной памятью. Например, в файловых системах ОС семейства Unix файл может иметь несколько имен (так называемых жестких ссылок, hard link). Операции удаления файла как таковой не предоставляется, есть только операция удаления имени. При удалении последнего имени файл действительно удаляется.

Вообще, подсчет ссылок широко применяется в ядрах ОС для управления системными объектами, особенно такими, которые могут разделяться несколькими задачами — например, сегментами разделяемой памяти, особенно разделяемыми библиотеками, дескрипторами открытых файлов в Unix, средствами межпроцессного взаимодействия, объектами графического пользовательского интерфейса и т. д. Необходимость неявного удаления таких объектов продиктована тем, что задача может аварийно завершиться, не освободив некоторые из своих ресурсов; удалять же все разделяемые объекты при завершении одной из задач, очевидно, недопустимо. Часто такие системы указывают в описании системного API, что ресурсы, которые программа не освободит при своем завершении (как штатном, так и аварийном), будут освобождены неявно.

Впрочем, подсчет ссылок обладает одним практически важным недостатком — если у нас есть циклический список, на который нет ни одной ссылки извне, то все объекты в нем будут считаться используемыми, хотя они и являются мусором. Если мы по тем или иным причинам уверены, что кольца не возникают, метод подсчета ссылок вполне приемлем; если же мы используем графы произвольного вида, необходим более умный алгоритм.

4.3.2. Просмотр ссылок

Наиболее распространенной альтернативой подсчету ссылок является периодический просмотр всех ссылок, которые мы считаем "существующими" (что бы под этим ни подразумевалось) (рис. 4.14). В англоязычной литературе такой алгоритм называют mark and sweep (пометить и стереть).

Обычно просмотр начинается с именованных переменных и параметров функций. Если некоторые из указемых объектов сами по себе могут содержать ссылки, мы вынуждены осуществлять просмотр рекурсивно. Проведя эту рекурсию до конца, мы можем быть уверены, что то и только то, что мы просмотрели, является нужными данными, и с чистой совестью можем объявить все остальное мусором. Эта стратегия решает проблему кольцевых

списков, но требует остановки всей остальной деятельности, которая может сопровождаться созданием или уничтожением ссылок.

Сборка мусора просмотром ссылок (иногда ее называют просто сборкой мусора без каких-либо дополнительных уточнений) является очень дорогой операцией, т. к. требует остановки всех действий, связанных с передачей указателей (присваивания, передача параметров процедурам и т. д.). По той же причине эту операцию нельзя запускать отдельной низкоприоритетной нитью; если сборка мусора делается в многопоточной программе, на время этой операции необходимо остановить все нити.

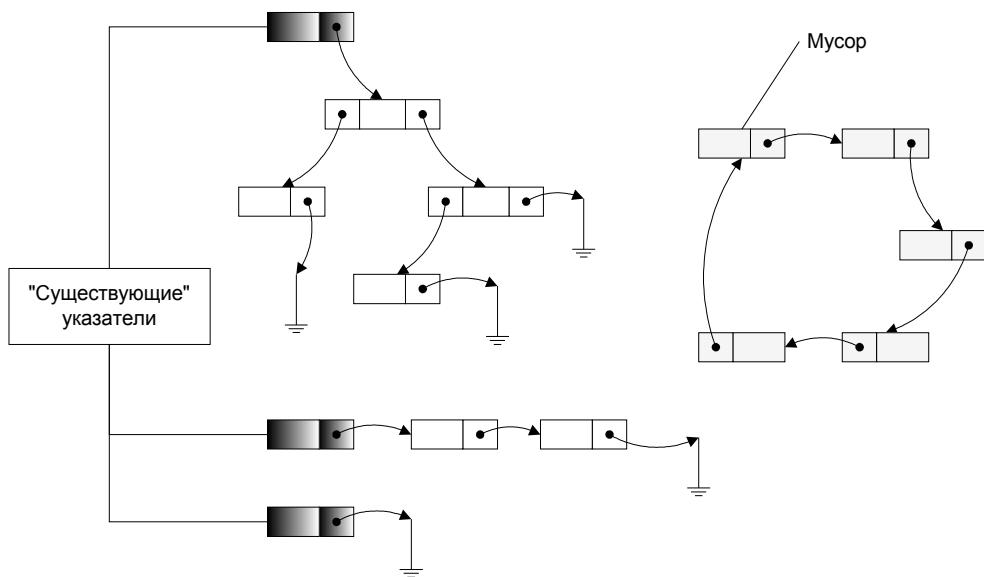


Рис. 4.14. Сборка мусора просмотром ссылок

Реальные системы, использующие сборку мусора, вызывают сборщик через определенные интервалы времени либо при достижении некоторого порога занятой памяти, что произойдет раньше. Ряд распространенных систем программирования, в том числе виртуальные машины Java, позволяют программисту вызвать сборщик мусора явно.

Но все эти методы, разумеется, не позволяют программисту гарантировать, что объект будет уничтожен к определенному моменту времени. В результате, потребности таких систем в оперативной памяти очень велики, и на большинстве задач в несколько раз превосходят требования эквивалентных программ, разработанных на основе явного удаления объектов или подсчета ссылок.

Кроме того, если объект занимает не только память, но и какие-то дорогостоящие внешние ресурсы — элементы графического пользовательского интерфейса, открытые файлы, сетевые соединения — эти ресурсы также будут освобождаться только по мере проходов сборщика мусора, т. е. в труднопредсказуемые моменты времени. На практике это приводит к чрезмерному использованию (или, скорее, к чрезмерному резервированию) таких ресурсов и опять-таки значительному повышению ресурсоемкости приложений.

Таким образом, хотя неявное удаление объектов резко снижает стоимость разработки программ, оно столь же резко повышает стоимость их исполнения. Для широкого класса задач — например, для систем поддержки бизнеса в небольших и средних предприятиях, которые разрабатываются на заказ и в которых переделка кода требуется довольно часто из-за меняющихся потребностей бизнеса, — оказывается выгодно минимизировать стоимость разработки и мириться с относительно высокой стоимостью исполнения.

В программах, распространяющихся большими тиражами, стоимость исполнения оказывается более важна — но тут вступают в игру более сложные факторы; например, необходимо учитывать возможность резкого изменения рыночной конъюнктуры (коммерческие риски) и тот, очевидный для экономистов, но не всегда понятный техническим специалистам факт, что деньги сегодня дороже денег завтра.

Говоря проще, если мы нарисовали бизнес-план, предполагающий миллионы запусков нашей программы, и, исходя из этого, оценили, что ее целесообразно писать на C++, мы сталкиваемся с риском, что кто-то напишет аналогичную программу раньше нас и наш проект закончится коммерческим провалом. При этом количество реальных запусков нашей программы будет изменяться единицами, а вовсе не запланированными миллионами.

Необходимо отметить, что большие трудозатраты при разработке на C++ приводят к увеличению сроков разработки и повышают этот риск. Несмотря на все эти соображения, тиражируемые продукты до сих пор преимущественно разрабатываются на языках с явным освобождением памяти.

Существует также ряд сегментов рынка, в которых стоимость исполнения однозначно важнее стоимости разработки. Нередко случается, что одной только неспособности выполнить программу в определенных ресурсных рамках достаточно для полного провала проекта. Важный пример — это встраиваемые приложения, особенно те, где требуется автономное питание — носимые и портативные устройства, бортовые компьютеры транспортных средств и космических аппаратов, многие военные приложения. В этих приложениях решающей часто оказывается не стоимость ресурсов самих по себе, а их энергопотребление. Добавление динамической оперативной памяти приводит к увеличению потребляемой мощности. У бортового компьютера это может вы-

звать перегрузку бортовой электросети, в портативном устройстве приведет к снижению срока автономной работы и/или увеличению габаритов и цены устройства за счет более мощной батареи и т. д. Для статического ОЗУ или ЭСППЗУ это не всегда так, но такая память намного дороже динамического ОЗУ и может быть просто не доступна в требуемых объемах.

4.3.3. Генерационная сборка мусора

Важным недостатком сборщиков мусора с просмотром ссылок является тот факт, что на время работы сборщика необходимо останавливать работу системы или, точнее, всю деятельность, которая может привести к созданию и уничтожению ссылок на объекты. Поэтому сборка мусора в том виде, в каком она описана в разд. 4.3.2, не может осуществляться фоновой нитью. Кроме того, время работы сборщика мусора растет в линейной зависимости от количества объектов. В больших приложениях оно может составлять десятки процентов от общего времени исполнения программы.

Особенно большую проблему эти недостатки представляют в таких языках, как Java/C#, в которых все объекты (кроме скалярных переменных) выделяются и уничтожаются сборщиком мусора. Действительно, если в языке C++ мы можем описать локальную переменную в функции или методе, память под нее будет выделена в момент вызова функции в стеке. При выходе из функции или, точнее, из блока, в котором был описан объект, эта память будет автоматически освобождена. Напротив, в Java в стеке создается только ссылка на объект; сам объект выделяется в куче. После выхода из блока уничтожается указатель, но не объект; для уничтожения объекта необходим проход сборщика мусора. Таким образом, нормальная работа программы на Java сопровождается созданием большого количества короткоживущих объектов, которые, однако, сами по себе не уничтожаются.

Генерационный сборщик требует перемещения объектов по памяти, поэтому он несовместим с языками, использующими указатели, такими как Pascal и C/C++. Для работы генерационного сборщика система должна реализовать "ручки" (handle) — промежуточные объекты, через которые проходят все обращения к указанному объекту. Язык высокого уровня делает использование "ручки" прозрачным для программиста и не требует явных преобразований "ручки" в указатель.

В простейшей форме генерационного сборщика мусора (generational garbage collection или generational scavenging) [Ungar 1984] объектный пул разбит на две части одинакового размера. Все вновь создаваемые объекты создаются в одной из частей, называемой "эдемом". Вторая часть пула в это время не используется (рис. 4.15).

При заполнении "эдема" система останавливается и начинается сборка мусора. Сборщик просматривает все объекты и — рекурсивно — ссылки из них на другие объекты, однако вместо простой отметки объекта он копирует каждый из обнаруженных объектов в неиспользуемую часть пула. Когда просмотр завершен, бывший "эдем" вместе со всем его содержимым объявляется мусором. Неиспользовавшаяся половина пула объявляется новым "эдемом" и работа системы продолжается. Такой сборщик мусора также называется *копирующим* (copying).

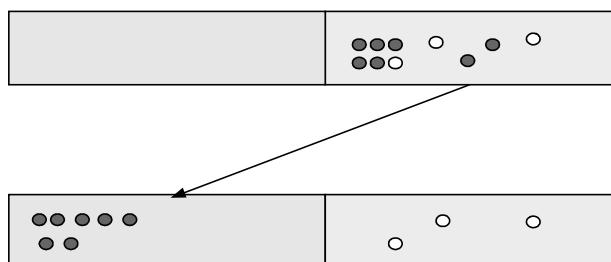


Рис. 4.15. Копирующий сборщик мусора

В системах, в которых объекты имеют финализационные методы, — как в Java/C# — разумеется, нельзя просто отбросить старый "эдем". Необходимо просмотреть его и выполнить финализационные методы для каждого из отброшенных объектов, но это может быть сделано фоновым процессом.

Важным преимуществом такого сборщика является тот факт, что сборка мусора оказывается совмещена с дефрагментацией пула. Поэтому можно значительно ускорить работу аллокатора — вместо просмотра списка свободных блоков памяти аллокатор может просто запоминать границу между свободной и занятой частями "эдема" и выделять память под новые объекты, сдвигая эту границу. Впрочем, основной недостаток сборщиков мусора — необходимость останавливать систему на время сборки — такой подход не устраняет.

Идея более сложных версий генерационных сборщиков основана на гипотезе, что срок жизни большинства объектов невелик и вероятность уничтожения вновь созданного объекта существенно выше, чем вероятность уничтожения давно существовавших объектов. Кроме того, с высокой вероятностью вновь создаваемые объекты будут ссылаться на старые объекты и друг на друга, но старые объекты вряд ли будут ссылаться на короткоживущие.

Вместо двух поколений объектов пул разбивается на несколько поколений разного размера с разными ожидаемыми сроками жизни. Сборка мусора состоит в переносе объектов из младших поколений в старшие. Сборка мусора,

которая охватывает только "эдем" и младшее поколение, называется "малой" (minor). Заполнение пула старшего поколения инициирует "большую" (major) сборку мусора, которая охватывает весь пул.

При таком подходе серьезную проблему представляют ссылки из объектов старших поколений на объекты младших поколений. Гипотеза, что такие ссылки редки, весьма убедительна и подтверждается практикой, но практика же показывает, что такие ссылки иногда все же возникают. Кроме того, для корректной работы системы такие ссылки необходимо обрабатывать, как бы ни была низка вероятность их появления.

Чтобы решить эту проблему, система отслеживает все передачи ссылок между объектами и запоминает все ссылки из объектов старшего поколения на младшие в специальном списке, который так и называется — *remembered set* (запомненное множество; общепринятого русскоязычного перевода этого названия нет). Таким образом, при малой сборке мусора необходимо просмотреть ссылки из именованных переменных, *remembered set* и — рекурсивно — из объектов младшего поколения, на которые эти ссылки указывают. Попадание по ссылке на объект старшего поколения означает прекращение рекурсии.

Этот подход обеспечивает приемлемую производительность только при небольшом объеме *remembered set*. Статистика показывает, что на практике этот объем действительно оказывается небольшим и, при удачном подборе параметров сборщика, иногда вообще нулевым.

Ориентация на среднюю производительность делает генерационную сборку мусора неприемлемой для задач реального времени; впрочем, все варианты сборки мусора с просмотром не обеспечивают гарантированного поведения и потому для задач реального времени непригодны.

Ключевым параметром, который определяет поведение системы под данной конкретной загрузкой, оказывается относительный размер пулов разных поколений. При правильном подборе этого параметра большинство объектов уничтожаются при малых сборках мусора. Большие сборки при этом происходят относительно редко. Поэтому, хотя работа сборщика по-прежнему предполагает остановку системы, среднее время такой остановки оказывается невелико и наблюдаемая производительность системы значительно возрастает.

Дополнительное преимущество генерационного сборщика состоит в том, что часто используемые объекты каждого из поколений накапливаются в начале пула своего поколения. Таким образом, все часто используемые объекты собираются в нескольких относительно небольших областях памяти — благодаря этому значительно повышается эффективность работы кэшей центральных процессоров и страничной виртуальной памяти.

Сборка мусора в Sun Java HotSpot VM

Виртуальная машина Java HotSpot была реализована в версии Java 1.2.2 и стала стандартной в Java 1.3. В этой VM используется генерационный сборщик мусора, предполагающий разбиение объектов на четыре поколения: перманентные, старые (old), молодые (new/young) и вновь создаваемые (eden) [Gotttry 2002]. Структура пулла Java Hotspot изображена на рис. 4.16.

Перманентные объекты — это системные объекты JVM, срок жизни которых всегда равен времени исполнения программы. При нормальной работе системы они не подвергаются сборке мусора.

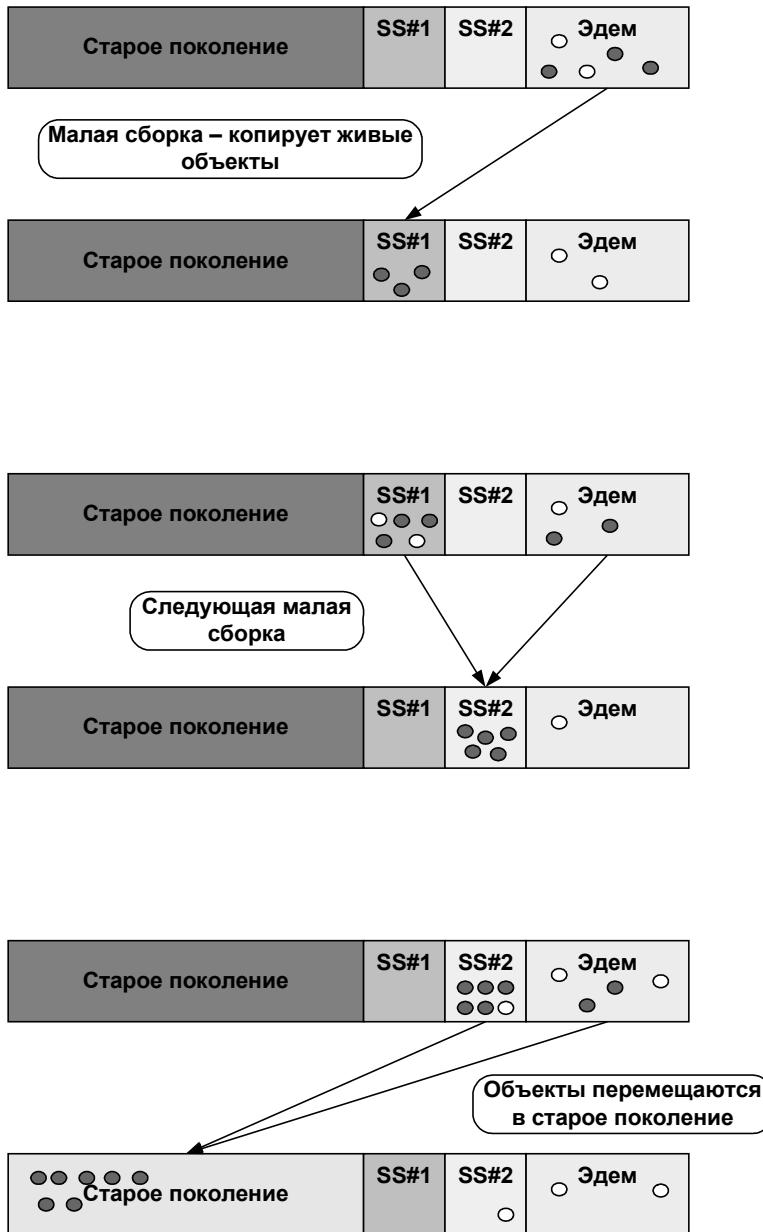


Рис. 4.16. Структура пула в JVM Hotspot

Пул молодых объектов состоит из двух частей, называемых Semispace#1 и Semispace#2 (полупространства 1 и 2, сокращенно SS#1 и SS#2). В некоторых статьях SS расшифровывают как Survivor Space (пространство выживших).

Все вновь создаваемые объекты создаются в "эдеме". При заполнении эдема инициируется малая сборка мусора.

При первой малой сборке мусора SS#1 и SS#2 пусты. Все используемые объекты из эдема перемещаются в SS#1; все, что осталось в эдеме, объявляется мусором, эдем очищается.

При второй малой сборке мусора SS#1 занят, но SS#2 пуст. Используемые объекты из SS#1 и эдема перемещаются в SS#2, оставшееся содержимое эдема и SS#1 очищается.

Объект, определенное число раз перемещавшийся между SS#1 и SS#2, признается долгоживущим (tenured) и перемещается в пул старшего поколения. Соответствующий параметр не задается явно; вместо этого система динамически подбирает порог "зрелости" объекта, пытаясь за счет этого обеспечить определенный уровень заполнения пространства выживших.

Вместе с каждым долгоживущим объектом перемещаются и все объекты, на которые он ссылался, даже если эти объекты находились в эдеме. Это делается, чтобы защитить соответствующие объекты от потери при последующих малых сборках мусора.

Данная стратегия обоснована только статистикой исполнения реальных программ и гипотезой, что долгоживущие объекты не ссылаются на короткоживущие, однако практика подтверждает, что обычно она работает довольно хорошо.

При заполнении пула старшего поколения инициируется большая сборка мусора. Эта сборка затрагивает только объекты старшего поколения и производится в два этапа. На первом этапе сборщик помечает используемые объекты, как при обычном mark and sweep. На втором этапе происходит дефрагментация — все используемые объекты копируются в начало пула. В зависимости от настроек JVM, после большой сборки мусора может быть запущена малая.

Таким образом, система имеет следующие параметры настройки:

- `ms` — начальный размер пула, который соответствует сумме размеров эдема и полупространств среднего поколения SS#1 и SS#2;
- `semispaces` или `SurvivorRatio` — размер полупространств среднего поколения; этот параметр может задаваться как абсолютным значением (`semispaces`), так и относительно начального размера пула (`SurvivorRatio`);
- `TargetSurvivorRatio` — доля (в процентах) заполнения пространства выживших;
- размер пространства старшего поколения. В параметрах командной строки вместо этого указывается общий максимальный размер пула `mx`; пространство старшего поколения соответствует разности между минимальным и максимальным размерами пула.

Существует также ряд других способов задавать эти параметры друг относительно друга; чаще всего используется параметр `NewRatio`, задающий отношение объемов пула старого и молодого поколения (суммы объемов эдема и полупространств). Так, при `NewRatio=2` под пул молодого поколения занимается 1/3 памяти, а пул старого, соответственно, 2/3.

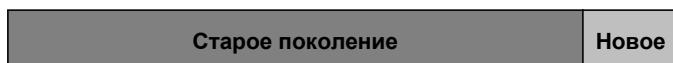
Пользователь может подбирать эти параметры самостоятельно, наблюдая за работой приложения, — для этого можно заставить JVM выдавать довольно подробную статистику работы сборщика мусора, в том числе:

- среднее и максимальное время работы сборщика (с разбивкой на малые и большие сборки);
- общий уровень заполнения пула и его разбивка по поколениям;
- порог " зрелости" объекта и т. д.

По умолчанию JVM реализует два режима работы (рис. 4.17): клиентский и серверный.

В клиентском режиме система выделяет 1/9 пула под новое поколение объектов и, соответственно, 8/9 под старое поколение. При этом малые сборки мусора происходят очень часто; клиентские программы обычно работают недолго, и потому до большой сборки мусора может вообще не дойти.

Для серверного режима под новое поколение выделяется 1/3 пула, а под старое — 2/3. Это приводит к значительному перераспределению времени между малыми и большими сборками мусора: малые сборки происходят реже, а большие — чаще, но зато каждая большая сборка занимает меньше времени.



New ratio = 8 для HotSpot Client JVM



New ratio = 2 для HotSpot Server JVM

Рис. 4.17. Относительные размеры пулов младшего и старшего поколения в разных версиях HotSpot JVM

Относительно успешная попытка решить проблему фоновой сборки мусора — это *репликационный* или *инкрементальный* сборщик мусора. В действительности, это вариант генерационного сборщика мусора. Основное отличие состоит в том, что при работе (как при просмотре, так и при дефрагментации) репликационный сборщик блокирует не систему в целом, а только нити, которые пытаются модифицировать уже просмотренные объекты. При этом нити, которые только читают значения полей объектов или модифицируют еще не просмотренные объекты, могут продолжать исполнение.

Поскольку объекты — особенно долгоживущие — читаются гораздо чаще, чем модифицируются, это может резко сократить время блокировок и значительно повысить наблюдаемую производительность, хотя, конечно же, полностью исключить блокировку не получается. Наибольший выигрыш при этом достигается при больших сборках, которые затрагивают преимуще-

ственno или исключительно долгоживущие объекты и вносят наибольший вклад в наблюдаемое время работы сборщика.

Особенно это значимо для многопоточных серверных программ.

В анонсе JVM 1.3 инкрементальный сборщик мусора даже был назван работающим без пауз (pauseless) [java.sun.com HotSpot]. Впрочем, надо отдать должное сотрудникам компании Sun, писавшим этот документ, — слово pauseless написано в кавычках и с разъяснением, что это означает на самом деле.

Все современные реализации Java, C# и других объектно-ориентированных языков со сборкой мусора (Visual Basic V7, Ocaml и т. д.) в той или иной форме поддерживают репликационную сборку мусора, хотя многие интерпретаторы позволяют ее отключить и вернуться к простому генерационному сборщику — это может быть выгодно для программ с малым числом потоков.

4.4. Открытая память (продолжение)

Описанные ранее алгоритмы распределения памяти используются не операционной системой, а библиотечными функциями, присоединенными к программе или, говоря шире, средой исполнения языка программирования. Однако ОС, которая реализует одновременную загрузку (но не обязательно одновременное выполнение: MS DOS — типичный пример такой системы) нескольких задач, также должна использовать тот или иной алгоритм размещения памяти. Отчасти такие алгоритмы могут быть похожи на работу функции `malloc`. Однако режим работы ОС может вносить существенные упрощения в алгоритм.

Перемещать образ загруженного процесса по памяти невозможно: даже если его код был позиционно-независим и не подвергался перенастройке, сегмент данных может содержать (и почти наверняка содержит) указатели, которые при перемещении необходимо перенастроить. Поэтому при выгрузке задач из памяти перед нами в полный рост встает проблема внешней фрагментации (рис. 4.18).

Управление памятью в OS/360

В этой связи нельзя не вспомнить поучительную историю, связанную с управлением памятью в системах линии IBM System 360. В этих машинах не было аппаратных средств управления памятью, и все программы разделяли общее виртуальное адресное пространство, совпадающее с физическим. Адресные ссылки в программе задавались 12-битовым смещением относительно базового регистра. В качестве базового регистра мог использоваться, в принципе, лю-

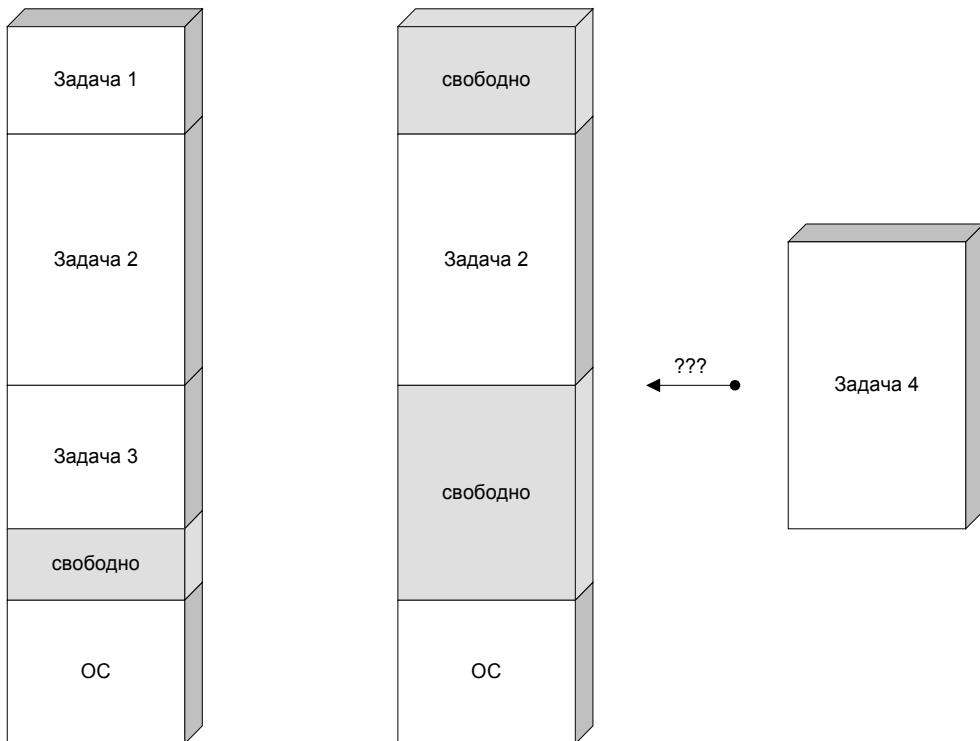


Рис. 4.18. Фрагментация при загрузке и выгрузке задач

бой из 16 32-битовых регистров общего назначения. Предполагалось, что пользовательские программы не модифицируют базовый регистр, поэтому можно загружать их с различных адресов, просто перенастраивая значение этого регистра. Таким образом, была реализована одновременная загрузка многих программ в многозадачной системе OS/360.

Однако после загрузки программу уже нельзя было перемещать по памяти: например, при вызове подпрограммы адрес возврата сохраняется в виде абсолютного 24-битового адреса (в System 360 под адрес отводилось 32-разрядное слово, но использовались только 24 младших бита адреса; в System 370 адрес стал 31-разрядным) и при возврате базовый регистр не применяется. Аналогично, базовый регистр не используется при ссылках на блоки параметров и сами параметры подпрограмм языка FORTRAN (в этом языке все параметры передаются по ссылке), при работе с указателями в PL/I и т. д. Перемещение программы, даже с перенастройкой базового регистра, нарушило бы все такие ссылки.

Разработчики фирмы IBM вскоре осознали пользу перемещения программ после их загрузки и попытались как-то решить эту проблему. Очень любопытный документ "Preparing to Rollin-Rollout Guideline" (Руководство по подготовке [программы] к вкатыванию и выкатыванию; к сожалению, мне не удалось найти полного текста этого документа) описывает действия, которые программа должна была бы предпринять после перемещения. Фактически программа должна была

найти в своем сегменте данных все абсолютные адреса и сама перенастроить их.

Естественно, никто из разработчиков компиляторов и прикладного программного обеспечения не собирался следовать этому руководству. В результате, проблема перемещения программ в OS/360 не была решена вплоть до появления машин System 370 со страничным или странично-сегментным диспетчером памяти и ОС MVS.

В данном случае проблема фрагментации особенно остра, т. к. типичный образ процесса занимает значительную часть всего доступного ОЗУ. Если при выделении небольших блоков мы еще можем рассчитывать на "закон больших чисел" и прочие статистические закономерности, то самый простой сценарий загрузки трех процессов различного размера может привести нас к неразрешимой ситуации (см. рис. 4.16).

Разделы памяти (*см. разд. 3.2*) отчасти позволяют решить проблему внешней фрагментации, устанавливая, что процесс должен либо использовать раздел целиком, либо не использовать его вовсе. Как и все ограничения на размер единицы выделения памяти, это решение загоняет проблему внутрь, переводя внешнюю фрагментацию во внутреннюю. Поэтому некоторые системы предлагают другие способы наложения ограничения на порядок загрузки и выгрузки задач.

Управление памятью в MS DOS

Так, например, процедура управления памятью MS DOS рассчитана на случай, когда программы выгружаются из памяти только в порядке, обратном тому, в каком они туда загружались (на практике они могут выгружаться и в другом порядке, но это явно запрещено в документации и часто приводит к проблемам). Это позволяет свести управление памятью к стековой дисциплине.

Каждой программе в MS DOS отводится блок памяти. С каждым таким блоком ассоциирован дескриптор, называемый *MCB* — *Memory Control Block* (рис. 4.19). Этот дескриптор содержит размер блока, идентификатор программы, которой принадлежит этот блок, и признак того, является ли данный блок последним в цепочке. Нужно отметить, что программе всегда принадлежит несколько блоков, но это уже несущественные детали. Другая малосущественная деталь та, что размер сегментов и их адреса отсчитываются в параграфах размером 16 байт. Знакомые с архитектурой процессора 8086 должны вспомнить, что адрес MCB в этом случае будет состоять только из сегментной части с нулевым смещением.

После запуска СОМ-файл получает сегмент размером 64 Кбайт, а EXE — всю доступную память. Обычно EXE-модули сразу после запуска освобождают ненужную им память и устанавливают *brklevel* на конец своего сегмента, а потом увеличивают *brklevel* и наращивают сегмент по мере необходимости.

Естественно, что наращивать сегмент можно только за счет следующего за ним в цепочке MCB, и MS DOS разрешит делать это только в случае, если этот сегмент не принадлежит никакой программе.

При запуске программы DOS берет последний сегмент в цепочке и загружает туда программу, если этот сегмент достаточно велик. Если он недостаточно велик, DOS говорит: "Недостаточно памяти" и отказывается загружать программу. При завершении программы DOS освобождает все блоки, принадлежавшие программе. При этом соседние блоки объединяются. Пока программы действительно завершаются в порядке, обратном тому, в котором они запускались, — все вполне нормально. Другое дело, что в реальной жизни возможны отклонения от этой схемы. Например, предполагается, что TSR-программы (Terminate, but Stay Resident — завершиться и остаться резидентно (в памяти)) никогда не пытаются по-настоящему завершиться и выгрузиться. Тем не менее любой уважающий себя хакер считает своим долгом сделать резидентную программу выгружаемой. У некоторых хакеров она в результате выбрасывается при выгрузке все резиденты, которые заняли память после нее. Другой пример — отладчики обычно загружают программу в обход обычной DOS-функции LOAD & EXECUTE, а при завершении отлаживаемой программы сами освобождают занимаемую ею память.

Я в свое время занимался прохождением некоторой программы под управлением отладчика. Честно говоря, речь шла о взломе некоторой игрушки... Эта программа производила какую-то инициализацию, а потом вызывала функцию DOS LOAD & EXECUTE. Я об этом не знал и, естественно, попал внутрь новой программы, которую и должен был взламывать.

После нескольких нажатий комбинаций клавиш `<Ctrl>+<Break>` я наконец-то вернулся в отладчик, но при каком-то очень странном состоянии программы. Покопавшись в программе с помощью отладчика в течение некоторого времени и убедившись, что она не хочет приходить в нормальное состояние, я вышел из отладчика и увидел следующую картину: системе доступно около 100 Кбайт в то время, как сумма длин свободных блоков памяти более 300 Кбайт, а размер наибольшего свободного блока около 200 Кбайт. Отладчик, выходя, освободил свою память и память отлаживаемой программы, но не освободил память, выделенную новому загруженному модулю. В результате посередине памяти остался никому не нужный блок изрядного размера, помеченный как используемый (рис. 4.20). Самым обидным было то, что DOS не пыталась загрузить ни одну программу в память под этим блоком, хотя там было гораздо больше места, чем над ним.

В системах с открытой памятью невозможны эффективные средства разделения доступа. Любая программная проверка прав доступа может быть легко обойдена прямым вызовом "защищаемых" модулей ядра. Даже криптографические средства не обеспечивают достаточно эффективной защиты, потому что можно "посадить" в память троянскую программу, которая будет анализировать код программы шифрования и считывать значение ключа.

В заключение можно привести основные проблемы многопроцессных систем без диспетчера памяти:

- проблема выделения дополнительной памяти задаче, которая загружалась не последней;
- проблема освобождения памяти после завершения задачи — точнее, возникающая при этом фрагментация свободной памяти;

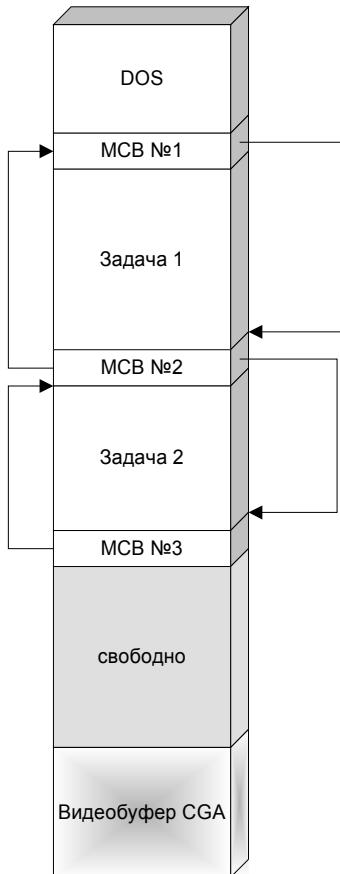


Рис. 4.19. Управление памятью в MS DOS

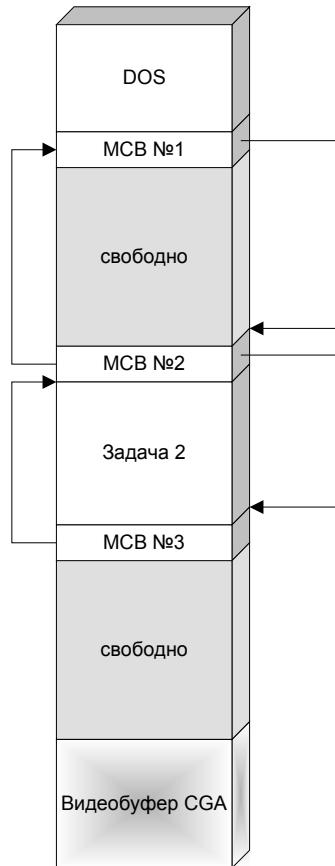


Рис. 4.20. Нарушения стекового порядка загрузки и выгрузки в MS DOS

- низкая надежность. Ошибка в одной из программ может привести к порче кода или данных других программ или самой системы;
- проблемы безопасности.

В системах с динамической сборкой первые две проблемы не так острой, потому что память выделяется и освобождается небольшими кусочками, по блоку на каждый объектный модуль, и код программы обычно не занимает непрерывного пространства. Соответственно, такие системы часто разрешают и данным программы занимать несмежные области памяти.

Такой подход используется многими системами с открытой памятью — AmigaDOS, Oberon, системами программирования для транспьютера и т. д. Проблема фрагментации при этом не снимается полностью, однако для создания катастрофической фрагментации не достаточно двух загрузок задач и од-

ной выгрузки, а требуется довольно длительная работа, сопровождающаяся интенсивным выделением и освобождением памяти.

В системе MacOS был предложен достаточно оригинальный метод борьбы с фрагментацией, заслуживающий отдельного обсуждения.

4.4.1. Управление памятью в MacOS и Win16

В этих системах предполагается, что пользовательские программы не сохраняют указателей на динамически выделенные блоки памяти. Вместо этого каждый такой блок идентифицируется целочисленным дескриптором или "ручкой" (handle) (рис. 4.21). Когда программа непосредственно обращается к данным в блоке, она выполняет системный вызов GlobalLock (запереть). Этот вызов возвращает текущий адрес блока. Пока программа не выполнит вызов GlobalUnlock (отпереть), система не пытается изменить адрес блока. Если же блок не заперт, система считает себя вправе передвигать его по памяти или даже сбрасывать на диск (рис. 4.22).

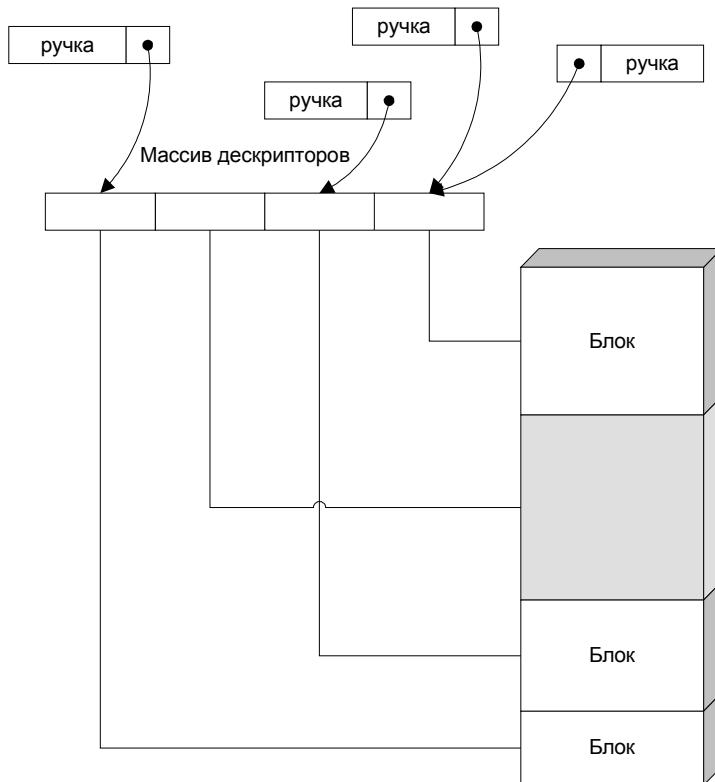


Рис. 4.21. Управление памятью с помощью "ручек"

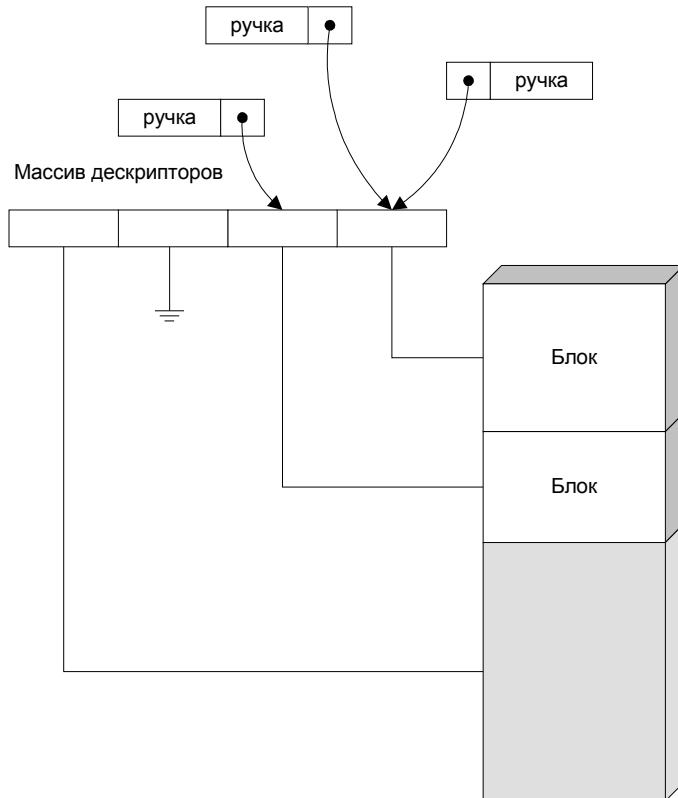


Рис. 4.22. Дефрагментация при управлении памятью с помощью "ручек"

"Ручки" представляют собой попытку создать программный аналог аппаратных диспетчеров памяти. Они позволяют решить проблему фрагментации и даже организовать некое подобие виртуальной памяти. Можно рассматривать их как средство организации оверлейных данных — поочередного отображения разных блоков данных на одни и те же адреса. Однако за это приходится платить очень дорогой ценой.

Нужно отметить, что эта цена оказывается дорогой только в системах программирования, в которых используются указатели в виде реальных адресов блоков памяти — т. е. в таких языках, как Pascal, Fortran, C/C++. В других языках, где понятие указателя или ссылки глубже спрятано от программиста — в Basic, Smalltalk, Java/C# — проблема не столь остры. В таких языках компилятор просто вставляет неявные вызовы GlobalLock/GlobalUnlock в каждое обращение к объекту. Разумеется, это снижает производительность, но преимущества, которые дает эта схема при управлении памятью и сборке мусора (см. разд. 4.3.3), могут оправдать этот недостаток.

В тех языках, где программисту требуется настоящий указатель, использование "ручек" сильно усложняет программирование вообще и в особенности перенос ПО из систем, использующих линейное адресное пространство. Все указатели на динамические структуры данных в программе нужно заменить на "ручки", а каждое обращение к таким структурам необходимо окружить вызовами `GlobalLock/GlobalUnlock`.

Вызовы `GlobalLock/GlobalUnlock`:

- сами по себе увеличивают объем кода и время исполнения;
- мешают компиляторам выполнять оптимизацию, прежде всего не позволяют оптимально использовать регистры процессора, потому что далеко не все регистры сохраняются при вызовах;
- требуют разрыва конвейера команд и перезагрузки командного кэша; в современных суперскалярных процессорах это может приводить к падению производительности во много раз. Любопытно, что в интерпретаторах Basic, Java/C# и т. д. этот недостаток в значительной мере компенсируется за счет того, что код функций `GlobalLock/GlobalUnlock` не оформлен в виде отдельных процедур, а просто вставлен в соответствующие точки кода интерпретатора. Легко понять, что сам этот код представляет собой просто установку и снятие флага в дескрипторе "ручки", т. е. в большинстве случаев может быть реализован одной командой центрального процессора.

Попытки уменьшить число блокировок требуют определенных интеллектуальных усилий. Фактически, к обычному циклу разработки ПО: проектирование, выбор алгоритма, написание кода и его отладка — добавляются еще две фазы: микрооптимизация использования "ручек" и отладка оптимизированного кода. Последняя фаза оказывается, пожалуй, самой сложной и ответственной.

Наиболее опасной ошибкой, возникающей на фазе микрооптимизации, является вынос указателя на динамическую структуру за пределы скобок `GlobalLock/GlobalUnlock`. Эту ошибку очень сложно обнаружить при тестировании, т. к. она проявляется, только если система пыталась передвигать блоки в промежутках между обращениями. Иными словами, ошибка может проявляться или не проявляться себя в зависимости от набора приложений, исполняющихся в системе, и от характера деятельности этих приложений. В результате мы получаем то, чего больше всего боятся эксплуатационники, — систему, которая работает *иногда*.

Не случайно фирма Microsoft полностью отказалась от управления памятью с помощью "ручек" в следующей версии MS Windows — Windows 95, в которой реализована почти полноценная виртуальная память. При переходе от

Windows 3.x к Windows 95 наработка на отказ — даже при исполнении той же самой смеси приложений — резко возросла, так что система из работающей *иногда* превратилась в работающую *как правило*. По-видимому, это означает, что большая часть фатальных ошибок в приложениях Win16 действительно относилась к ошибкам работы с "ручками".

Mac OS версии 10, построенная на ядре BSD Mach, также имеет страничную виртуальную память и никогда не перемещает блоки памяти, адресуемые "ручками".

4.5. Системы с базовой виртуальной адресацией

Как уже говорилось, в системах с открытой памятью возникают большие сложности при организации многозадачной работы. Самым простым способом разрешения этих проблем оказалось предоставление каждому процессу своего виртуального адресного пространства. Простейшим методом организации различных адресных пространств является так называемая *базовая адресация*. По-видимому, это наиболее старый из реализовавшихся на практике способов виртуальной адресации, первые его реализации относятся к началу 60-х годов XX века, т. е. к компьютерам второго поколения. Впрочем, этот способ находит применение и в современных процессорах — в устройствах, предназначенных для встраиваемых приложений или при разделении памяти между виртуальными машинами.

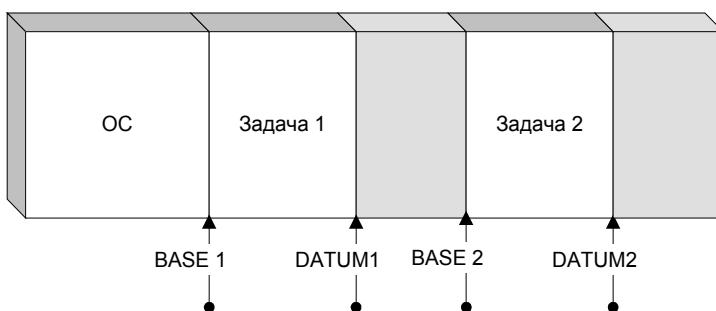


Рис. 4.23. Виртуальная память на основе базовой адресации

Вы можете заметить, что термин "базовая адресация" уже занят — мы называли таким образом адресацию по схеме `reg[offset]`. Метод, о котором сейчас идет речь, состоит в формировании адреса по той же схеме. Отличие состоит в том, что регистр, относительно которого происходит адресация, не доступен прикладной программе. Кроме того, его значение прибавляется ко

всем адресам, в том числе к "абсолютным" адресным ссылкам или переменным типа указатель.

Как правило, машины, использующие базовую адресацию, имеют два регистра (рис. 4.23). Один из регистров задает базу для адресов, второй устанавливает верхний предел. В системе ICL1900/Одренок эти регистры называются соответственно **BASE** и **DATUM**. Если адрес выходит за границу, установленную значением **DATUM**, возникает *исключительная ситуация (exception) ошибочной адресации*. Как правило, это приводит к тому, что система принудительно завершает работу программы.

Базовая адресация в процессорах PowerPC

В процессорах PowerPC предусмотрено два режима работы диспетчера памяти:

1. **Virtual mode** (виртуальный режим), при котором используется двухуровневая сегментно-страничная трансляция адресов.
2. **Real mode** (реальный режим), в котором обеспечивается прямой доступ к физической памяти.

Принципы работы диспетчера памяти в первом режиме будут рассматриваться в следующей главе.

При прямом доступе к памяти адрес все-таки может подвергаться модификации с помощью регистров RLMR (Real Mode Limit Register — регистр-ограничитель реального режима) и RMOR (Real Mode Offset Register — регистр смещения реального режима). Если эффективный адрес при обращении к памяти превышает значение RLMR, возникает исключение доступа к памяти, по которому вызывается процедура-обработчик.

Реальный режим главным образом используется во встраиваемых приложениях; процессоры линии Power находят применение во многих таких приложениях, начиная от управляющих процессоров лазерных принтеров и заканчивая бортовыми компьютерами марсианских зондов. Некоторые модели процессоров, например PPC 401CF, вообще не имеют сегментно-страничного диспетчера памяти и поддерживают только реальный режим адресации.

В процессорах с диспетчером памяти реальный режим и базовая адресация может использоваться для реализации гипервизора (*hypervisor*) или, что то же самое, диспетчера виртуальных машин. Каждая "задача", исполняющаяся в такой среде, представляет полноценную операционную систему, которая управляет своей памятью с помощью страничного диспетчера. Базовая же адресация позволяет защитить и скрыть эти "задачи" — образы виртуальных машин — друг от друга.

С помощью этих двух регистров мы сразу решаем две важные проблемы.

Во-первых, мы можем изолировать процессы друг от друга — ошибки в программе одного процесса не приводят к разрушению или повреждению образов других процессов или самой системы. Благодаря этому мы можем обеспечить защиту системы не только от ошибочных программ, но и от злонамеренных действий пользователей, направленных на разрушение системы или доступ к чужим данным.

Во-вторых, мы получаем возможность передвигать образы процессов по физической памяти так, что программа каждого из них не замечает перемещения. За счет этого мы решаем проблему фрагментации памяти и даем процессам возможность наращивать свое адресное пространство. Действительно, в системе с открытой памятью процесс может добавлять себе память только до тех пор, пока не доберется до начала образа следующего процесса. После этого мы должны либо говорить о том, что памяти нет, либо мириться с тем, что процесс может занимать несмежные области физического адресного пространства. Второе решение резко усложняет управление памятью как со стороны системы, так и со стороны процесса, и часто оказывается неприемлемым (подробнее связанные с этим проблемы обсуждаются в разд. 4.4). В случае же базовой адресации мы можем просто сдвинуть мешающий нам образ вверх по физическим адресам (рис. 4.24).

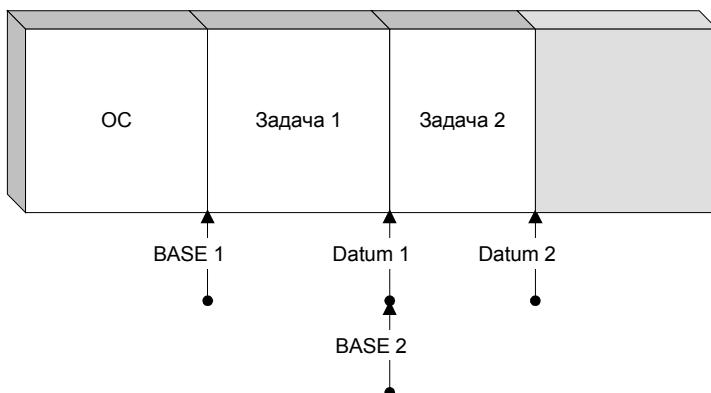


Рис. 4.24. Дефрагментация при использовании базовой адресации

Часто ОС, работающие на таких архитектурах, умеют сбрасывать на диск образы тех процессов, которые долго не получают управления. Это самая простая из форм *сплонкинга* (swapping — обмен) (русскоязычный термин "*страничный обмен*" довольно широко распространен, но в данном случае его использование было бы неверным, потому что обмену подвергаются не страницы, а целиком задачи).

Решив перечисленные ранее проблемы, мы создаем другие, довольно неожиданные. Мы оговорили, что базовый регистр недоступен прикладным задачам. Но какой-то задаче он должен быть доступен! Каким же образом процессор узнает, исполняет ли он системную или прикладную задачу, и не сможет ли злонамеренная прикладная программа его убедить в том, что является системной?

Другая проблема состоит в том, что, если мы хотим предоставить прикладным программам возможность вызывать систему и передавать ей параметры,

мы должны обеспечить процессы (как системные, так и прикладные) теми или иными механизмами доступа к адресным пространствам друг друга.

На самом деле эти две проблемы тесно взаимосвязаны — например, если мы предоставим прикладной программе свободный доступ к системному адресному пространству, нам придется рас прощаться с любыми надеждами на защиту от злонамеренных действий пользователей. Раз проблемы взаимосвязаны, то и решать их следует в комплексе.

Стандартное решение этого комплекса проблем состоит в следующем. Мы снабжаем процессор флагом, который указывает, исполняется системный или пользовательский процесс. Код пользовательского процесса не может манипулировать этим флагом, однако ему доступна специальная команда. В различных архитектурах данные специальные команды имеют разные мнемонические обозначения, далее мы будем называть эту команду `SYSCALL`. `SYSCALL` одновременно переключает флаг в положение "системный" и передает управление на определенный адрес в системном адресном пространстве. Процедура, находящаяся по этому адресу, называется *диспетчером системных вызовов* (рис. 4.25).

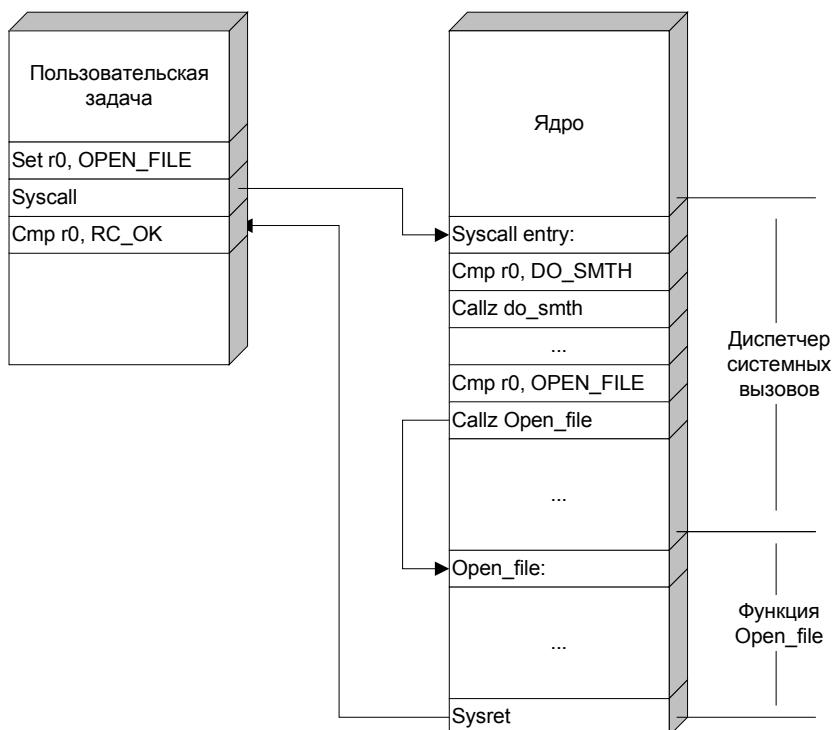


Рис. 4.25. Диспетчер системных вызовов

Возврат из системного вызова осуществляется другой специальной командой, назовем ее `SYSRET`. Эта команда передает управление на указанный адрес в указанном адресном пространстве и одновременно переводит флаг в состояние "пользователь". Необходимость выполнять эти две операции одной командой очевидна: если мы сначала сбросим флаг, мы потеряем возможность переключать адресные пространства, а если мы сначала передадим управление, никто не может нам гарантировать, что пользовательский код добровольно выйдет из системного режима.

Протокол общения прикладной программы с системой состоит в следующем: программа помещает параметры вызова в оговоренное место — обычно в регистры общего назначения или в стек — и исполняет `SYSCALL`. Одним из параметров передается и код системного вызова. Диспетчер вызовов анализирует допустимость параметров и передает управление соответствующей процедуре ядра, которая и выполняет требуемую операцию (или не выполняет, если у пользователя не хватает полномочий). Затем процедура помещает в оговоренное место (чаще всего опять-таки в регистры или в пользовательский стек) возвращаемые значения и передает управление диспетчеру, или вызывает `SYSRET` самостоятельно.

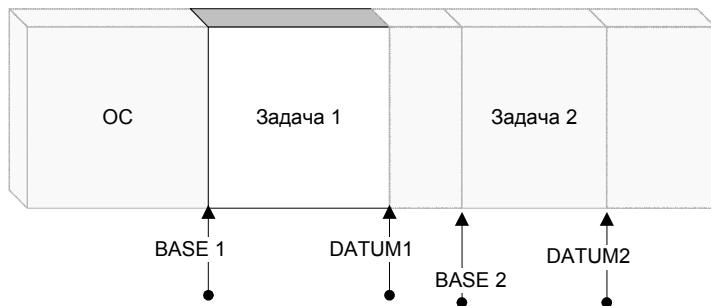
Сложность возникает, когда ядру при исполнении вызова требуется доступ к пользовательскому адресному пространству. В простейшем случае, когда все параметры (как входные, так и выходные) размещаются в регистрах и представляют собой скалярные значения, проблемы нет, но большинство системных вызовов, особенно запросы обмена данными с внешними устройствами, в эту схему не укладываются.

В системах с базовой адресацией эту проблему обычно решают просто: в "системном" режиме базовый и ограничительный регистры не используются вообще, и ядро имеет полный доступ ко всей физической памяти, в том числе и к адресным пространствам всех пользовательских задач (рис. 4.26). Это решение приводит к тому, что хотя система и защищена от ошибок в пользовательских программах, пользовательские процессы оказываются совершенно не защищены от системы, а ядро — не защищено от самого себя. Ошибка в любом из системных модулей приводит к полной остановке работы.

В архитектурах с более сложной адресациейнередко предоставляются специальные инструкции для передачи данных между пользовательским и системным адресными пространствами. Однако и в этом случае ядро ОС обычно имеет полный доступ к адресным пространствам пользовательских задач.

В современных системах базовая виртуальная адресация используется редко. Дело не в том, что она плоха, а в том, что более сложные методы, такие как сегментная и страничная трансляция адресов, оказались намного лучше. Часто под словами "виртуальная память" подразумевают именно сегментную или страничную адресацию.

Пользовательский режим



Системный режим

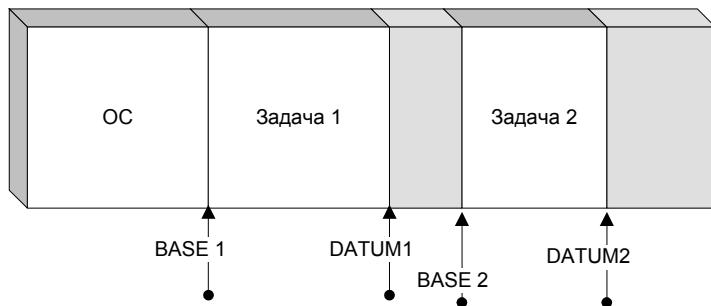


Рис. 4.26. Системный и пользовательский режимы

Вопросы для самопроверки

1. Что такое внешняя фрагментация?
2. Почему в большинстве случаев невозможно перемещать занятые блоки памяти?
3. Что такое внутренняя фрагментация? При каких условиях она возникает?
4. Может ли внутренняя фрагментация возникать при выделении памяти блоками переменного размера?
5. Почему стратегия first fit считается лучшей из известных стратегий поиска свободных блоков?

6. Почему алгоритм работы `malloc/free` из библиотеки GNU LibC назван аналогом алгоритма парных меток, несмотря на то, что никаких парных меток GNU LibC не использует?
7. Каково основное преимущество алгоритма близнецов?
8. Каким образом использование слабов позволяет перераспределять память между очередями блоков разного размера?
9. Назовите основные ограничения алгоритма подсчета ссылок.
10. Перечислите недостатки сборки мусора просмотром ссылок.
11. Как вы думаете, почему не применяются гибридные алгоритмы сборки мусора, сочетающие подсчет и просмотр ссылок?
12. Почему генерационные алгоритмы сборки мусора иногда называют "работающими без пауз"? Корректно ли такое название?
13. Что такое `remembered set` и почему без него невозможно реализовать "малую" сборку мусора?
14. Почему инкрементальную сборку мусора называют также репликационной?
15. Верно ли, что инкрементальная сборка мусора может выполняться фоновой нитью без блокировки рабочих нитей программы?
16. Почему невозможно осуществить сборку мусора с просмотром ссылок и какие бы то ни было ее варианты, в том числе генерационную и инкрементальную, без полной или хотя бы частичной остановки системы?
17. Какие проблемы может решить виртуальная память?
18. Почему даже простейшая реализация виртуальной памяти, такая как базовая адресация, требует специальных механизмов для исполнения системных вызовов?

ГЛАВА 5



Сегментная и страничная виртуальная память

— Так как же ты, Петька, дошел до такой жизни, что спрашиваешь меня, своего боевого командира, всегда ли то, что происходит у тебя в голове, — это то, что происходит у тебя в голове, или не всегда?

В. Пелевин

В предыдущей главе и во введении мы описали ряд причин, по которым прямой доступ к оперативной памяти оказывается нежелателен:

1. Внешняя фрагментация доступной памяти. Для борьбы с ней было бы желательно иметь возможность размещать логически непрерывные объекты (например, образы процессов или разделяемых библиотек) в несмежных областях физической памяти и/или передвигать объекты по памяти.
2. Защита программ друг от друга. Из-за ошибок программирования одна задача может испортить код и данные других задач или ядра системы. Для решения этой проблемы хотелось бы гарантировать, что задача сможет работать только с той памятью, которая была выделена ей системой.
3. Защита системы от злонамеренных программ. Если пользовательские задачи смогут бесконтрольно читать и изменять данные ядра системы, вызывать любые функции ядра или самостоятельно выполнять операции ввода/вывода, никакие средства защиты и разделения доступа, реализованные на уровне ядра ОС, не могут быть эффективны.
4. Дисковая подкачка (сплонг). Поскольку не все задачи активны одновременно, некоторые из задач можно остановить и сбросить их области на диск и, таким образом, увеличить объем реально доступной памяти. Более того, поскольку далеко не каждой задаче в каждый момент времени нужны весь ее код и данные, очень привлекательна была возможность сбрасывать на диск отдельные объекты, которые задача в данный момент не использует.

Далее в этой главе мы рассмотрим основной способ решения этой задачи в современных вычислительных системах — страничную и сегментную виртуальную память. Сначала мы разберем, как устроен диспетчер памяти в таких системах, а затем поймем, как он позволяет решать все перечисленные нами задачи.

В системах с *сегментной и страничной адресацией* виртуальный адрес имеет сложную структуру. Он разбит на два битовых поля: *селектор страницы (сегмента)* и *смещение* в нем. Соответственно, адресное пространство оказывается состоящим из дискретных блоков. Если все эти блоки имеют фиксированную длину и образуют вместе непрерывное пространство, они называются *страницами* (рис. 5.1). Если длина каждого блока может задаваться, а неиспользуемым частям блоков соответствуют "дыры" в виртуальном адресном пространстве, такие блоки называются *сегментами* (рис. 5.2). Как правило, один сегмент соответствует коду или данным одного модуля программы. Со страницей или сегментом могут быть ассоциированы права чтения, записи и исполнения.

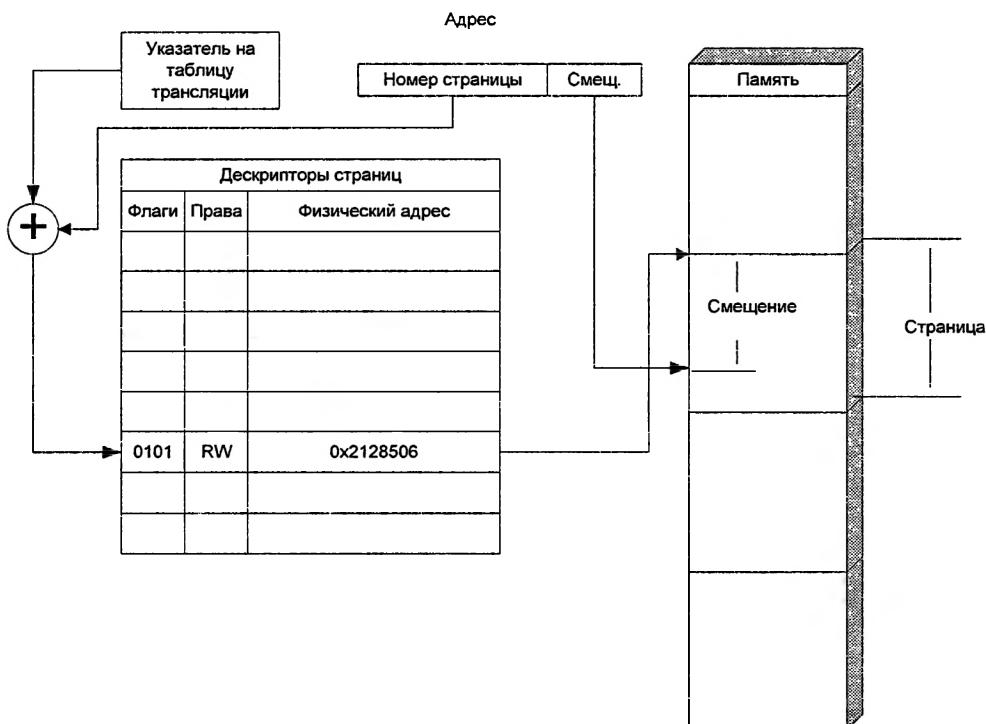


Рис. 5.1. Страницчная виртуальная память

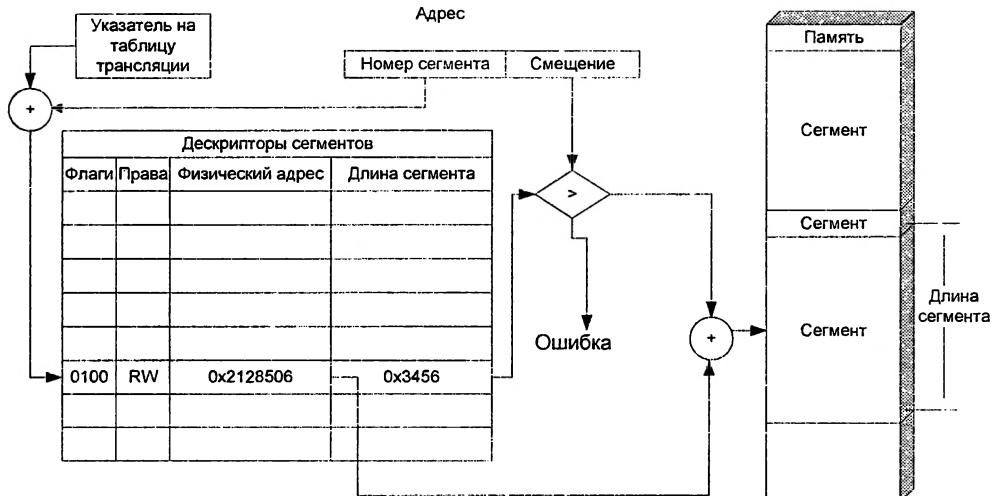


Рис. 5.2. Сегментная виртуальная память

Такая адресация реализуется аппаратно. Процессор имеет специальное устройство, называемое *диспетчером памяти* или, как его называли в старой русскоязычной литературе, УУП (Устройство Управления Памятью, ср. MMU — Memory Management Unit). В некоторых процессорах, например в MC68020, MC68030 или в некоторых старых RISC-системах, это устройство реализовано на отдельном кристалле; в других, таких как x86 или современные RISC-процессоры, диспетчер памяти интегрирован в процессор.

В 16-разрядной машине PDP-11, используемый в которой диспетчер памяти рассматривался в разделе 2.7, сегментов всего восемь, поэтому дескрипторы каждого из них размещаются в отдельном регистре (на самом деле, регистров не восемь, а шестнадцать — восемь для пользовательского адресного пространства и восемь для системного). На 32-разрядных машинах количество сегментов измеряется тысячами, а страниц — иногда и миллионами, разместить все их дескрипторы в регистрах процессора невозможно, поэтому приходится прибегать к более сложной схеме.

Диспетчер памяти содержит регистр — указатель на *таблицу трансляции*. Эта таблица размещается где-то в ОЗУ. Ее элементами являются дескрипторы каждой страницы/сегмента. Такой дескриптор содержит права доступа к странице, признак присутствия этой страницы в памяти и физический адрес страницы/сегмента. Для сегментов в дескрипторе также хранится его длина.

Большинство реальных программ используют далеко не все адресное пространство процессора, соответственно, таблица трансляции не обязательно содержит все допустимые дескрипторы. Поэтому практически все диспетче-

ры памяти имеют еще один регистр — ограничитель длины таблицы трансляции. Страницы или сегменты, селектор которых превосходит ограничитель, не входят в виртуальное адресное пространство процесса.

Как правило, диспетчер памяти имеет также *кэш* (cache) дескрипторов — быструю память с ассоциативным доступом. В этой памяти хранятся дескрипторы часто используемых страниц. Записи этого кэша называются TLB (Translation Lookaside Buffers — справочные буферы для трансляции [адресов]). TLB обычно несколько глупее, чем кэш-память данных: они автоматически не отслеживают изменений в хранящихся в памяти дескрипторах страниц, поэтому, если ОС поменяет отдельные записи в таблице трансляции или перезагрузит указатель на нее, обычно оказывается необходимо явным образом сбросить TLB или отдельные записи этого буфера.

Ассоциативная память

Важно понимать, чем ассоциативная память отличается от обычной. При обращении к обычной памяти необходимо указать адрес, т. е., попросту говоря, номер ячейки памяти. У ассоциативной памяти каждая ячейка имеет сложную структуру и состоит из двух полей — одно поле содержит собственно значение, второе поле — так называемый ключ (key) или тег (tag, ярлык). При обращении к ассоциативной памяти необходимо указать ключ, и в результате мы получим соответствующее этому ключу значение или сообщение, что такого ключа нет. В кэшах процессоров и в TLB в качестве ключа используется адрес, а в качестве значения — значение ячейки основной памяти, соответствующее этому адресу.

Способы реализации ассоциативной памяти отличаются большим разнообразием. В условиях, когда производительность не важна, ассоциативность может быть достигнута линейным поиском по всем ключам. Программные реализации ассоциативной памяти (например, ассоциативные массивы Perl, `stl::map` в C++ и др.) используют хэш-таблицы, отсортированные массивы или деревья поиска. Аппаратные реализации обычно производят параллельный поиск. В полностью ассоциативной памяти к каждой ячейке памяти или, точнее, к каждому тегу, подключен собственный компаратор, который и производит сравнение предоставленного ключа с хранящимся в ячейке. Реализованная таким образом ассоциативная память работает очень быстро (и результат поиска, и обоснованный ответ, что поиск завершился неудачей, могут быть получены за один такт), но, разумеется, гораздо дороже регистрового ОЗУ того же объема. Нередко встречаются реализации, в которых один компаратор установлен на каждые N ячеек. При обращении он осуществляет линейный поиск в своих ячейках памяти. Такие устройства значительно дешевле, чем полностью параллельная ассоциативная память, но и пропорционально медленнее.

TLB представляет собой ассоциативную память небольшого объема (обычно 32-64 записи), в которой тег соответствует селектору страницы виртуального адреса, а значение — дескриптору этой страницы.

В 64-разрядных процессорах объем таблиц трансляции мог бы измеряться терабайтами, поэтому УУП таких процессоров нередко не используют никаких таблиц в памяти. Такой диспетчер памяти имеют только TLB, в которых

записаны дескрипторы ранее использовавшихся страниц. Когда задача обращается к странице, для которой нет записи в TLB, возникает исключение, для обработки которого вызывается процедура-обработчик. Эта процедура должна найти (или сочинить) дескриптор соответствующей страницы; поскольку эта процедура предоставляемая ядром ОС, то способ, которым она будет формировать дескриптор, полностью под контролем разработчика ОС. Процедура-обработчик может использовать либо таблицы, аналогичные таблицам 32-разрядных процессоров, либо какие-то хэш-таблицы или другие разреженные структуры данных. В разд. 5.1 описывается организованное таким образом УУП процессоров UltraSPARC.

Алгоритм доступа к памяти по виртуальному адресу page:offset состоит из следующих шагов (рис. 5.3):

1. Проверить, существует ли страница page вообще. Если страницы не существует, возникает особая ситуация ошибки сегментации (segmentation

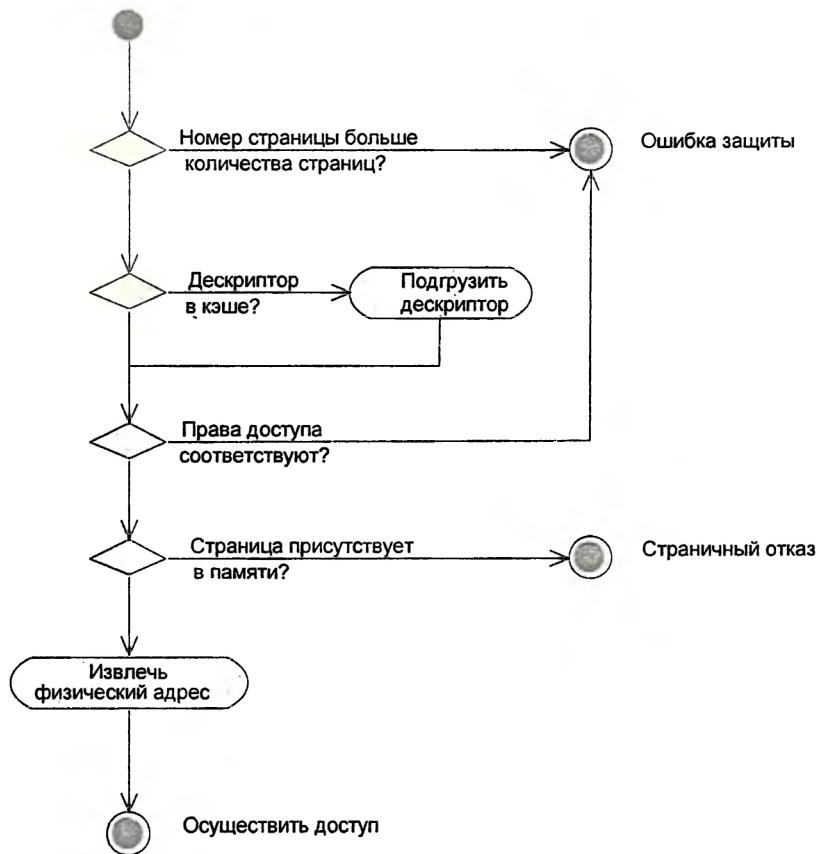


Рис. 5.3. Блок-схема алгоритма диспетчера памяти

violation) (понятие особой ситуации (исключения) подробнее разбирается в главе 6).

2. Попытаться найти дескриптор страницы в кэше.
3. Если его нет в кэше, загрузить дескриптор из таблицы в памяти.
4. Проверить, имеет ли процесс соответствующее право доступа к странице. Иначе также возникает ошибка сегментации.
5. Проверить, находится ли страница в оперативной памяти. Если ее там нет, возникает особая ситуация отсутствия страницы или страничный отказ (page fault). Как правило, реакция на нее состоит в том, что вызывается специальная программа-обработчик (trap — ловушка), которая загружает требуемую страницу с диска. В многопоточных системах во время такой загрузки может исполняться другой процесс.
6. Если страница есть в памяти, взять из ее дескриптора физический адрес `phys_addr`.
7. Если мы имеем дело с сегментной адресацией, сравнить смещение в сегменте с длиной этого сегмента. Если смещение оказалось больше, также возникает ошибка сегментации.
8. Произвести доступ к памяти по адресу `phys_addr[offset]`.

Видно, что такая схема адресации довольно сложна. Однако в современных процессорах все это реализовано аппаратно и, благодаря кэшу дескрипторов и другим ухищрениям, средняя скорость доступа к памяти получается почти такой же, как и при прямой адресации. Это справедливо даже для 64-разрядных процессоров, в которых используется программное управление TLB — обработчик исключения по отсутствию дескриптора в TLB обычно принудительно блокируют в кэш-памяти первого уровня, так что скорость его исполнения ненамного уступает скорости исполнения микропрограммы диспетчера памяти.

Страницчная и сегментная трансляция дает неоценимые преимущества при реализации многозадачных ОС.

Во-первых, мы можем связать с каждой задачей свою таблицу трансляции, а значит, и свое *виртуальное адресное пространство*. Благодаря этому даже в многозадачных ОС мы можем пользоваться абсолютным загрузчиком. Кроме того, программы оказываются изолированными друг от друга, и мы можем обеспечить их безопасность.

Во-вторых, мы можем сбрасывать на диск редко используемые области виртуальной памяти программ — не всю программу целиком, а только ее часть. В отличие от оверлейных загрузчиков, программа при этом вообще не обязана знать, какая ее часть будет сброшена.

В-третьих, программа не обязана занимать непрерывную область физической памяти. При этом она вполне может видеть непрерывное виртуальное адресное пространство. Это резко упрощает борьбу с фрагментацией памяти на уровне ОС (проблема фрагментации виртуального адресного пространства задачи остается), а в системах со страницочной адресацией проблема внешней фрагментации физической памяти вообще снимается.

В большинстве систем со страницочным диспетчером свободная память отслеживается с помощью битовой маски физических страниц. В этой маске свободной странице соответствует 1, а занятой — 0. Если кому-то нужна страница, система просто ищет в этой маске установленный бит. В результате виртуальное пространство программы может оказаться отображено на физические адреса очень причудливым образом, но это никого не волнует — в большинстве современных вычислительных систем скорость доступа ко всем страницам одинакова (рис. 5.4).

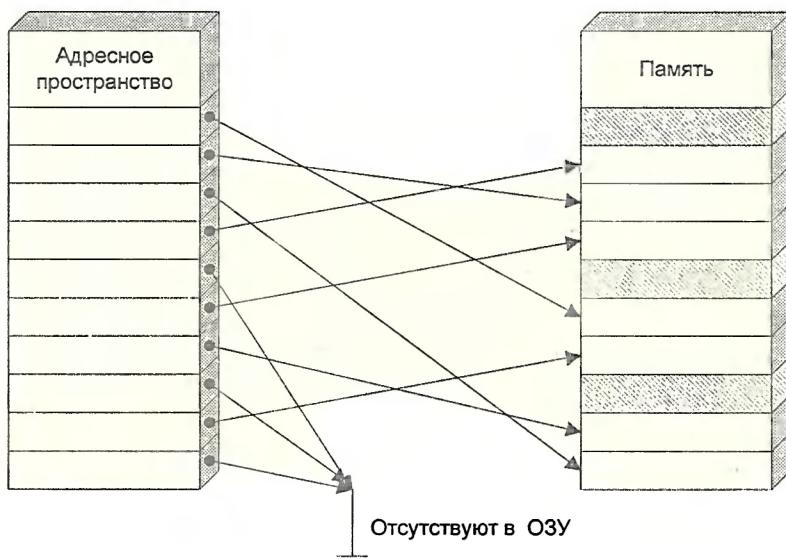


Рис. 5.4. Распределение адресного пространства по физической памяти

В некоторых многопроцессорных системах этого не происходит. Такие машины называются NUMA (NonUniform Memory Access — неоднородный доступ к памяти) и подробнее обсуждаются в разд. 6.5. ОС, предназначенные для работы на таких машинах, должны учитывать неоднородность памяти; очевидно, что возможность перемещать страницы по физической памяти без изменения их виртуального адреса облегчает решение и этой задачи тоже.

В-четвертых, система может обеспечивать не только защиту программ друг от друга, но в определенной мере и защиту программы от самой себя — на-

пример, от ошибочной записи данных на место кода или попытки исполнить данные.

В-пятых, разные задачи могут использовать общие области памяти для взаимодействия или, скажем, просто для того, чтобы работать с одной копией библиотеки подпрограмм.

Перечисленные преимущества настолько серьезны, что считается невозможным реализовать многозадачную систему общего назначения, такую, как UNIX или System/390 на машинах без диспетчера памяти.

Для систем реального времени, впрочем, виртуальная память оказывается скорее вредна, чем бесполезна: наличие диспетчера памяти увеличивает объем контекста процесса (*это понятие подробнее обсуждается в разд. 8.2*), воспользоваться же главным преимуществом — возможностью страничного обмена — задачи реального времени в полной мере не могут.

- Так, задаче РВ часто бывает нужно, чтобы определенные элементы ее образа никогда не сбрасывались на диск. Система реального времени обязана гарантировать время реакции, и это гарантированное время обычно намного меньше времени доступа к диску. Код, обрабатывающий событие, и используемые при этом данные должны быть всегда в памяти. Поэтому системы реального времени, использующие виртуальную память, обязательно предоставляют средства заблокировать определенные объекты в ОЗУ.
- Наличие TLB обеспечивает приемлемую среднюю производительность системы, но в худшем случае обращение к памяти приводит к промаху TLB (TLB miss) и к дополнительным обращениям к ОЗУ. Поэтому приложение РВ часто вынуждено блокировать не только свои страницы в памяти, но и дескрипторы этих страниц в TLB.

В результате, системы реального времени, даже работающие на процессорах со встроенным диспетчером памяти, часто этот диспетчер не используют.

Отдельной проблемой при разработке системы со страничной или сегментной адресацией является выбор размера страницы или максимального размера сегмента. Этот размер определяется шириной соответствующего битового поля адреса и поэтому должен быть степенью двойки.

С одной стороны, страницы не должны быть слишком большими, т. к. это может привести к внутренней фрагментации и перекачке слишком больших объемов данных при сбросе страниц на диск. С другой стороны, страницы не должны быть слишком маленькими, т. к. это приведет к чрезмерному увеличению таблиц трансляции, требуемого объема кэша дескрипторов и т. д.

В реальных системах размер страницы меняется от 512 байт у машин семейства VAX до нескольких килобайт. Например, x86 имеет страницу размером

4 Кбайт. Некоторые диспетчеры памяти, например у MC6801/2/30, имеют переменный размер страницы — система при запуске программирует диспетчер и устанавливает, помимо прочего, этот размер, и дальше работает со страницами выбранного размера. У процессора i860 и современных реализаций x86 (Pentium II, Athlon и более новых) размер страницы переключается между 4 Кбайт и 4 Мбайт.

С сегментными диспетчерами памяти ситуация сложнее. С одной стороны, хочется, чтобы один программный модуль помещался в сегмент, поэтому сегменты обычно делают большими, от 32 Кбайт и более. С другой стороны, хочется, чтобы в адресном пространстве можно было сделать много сегментов. Кроме того, может возникнуть проблема: как быть с большими неразделимыми объектами, например хэш-таблицами компиляторов, под которые часто выделяются сотни килобайт. Часто под такие структуры приходится выделять несколько смежных сегментов максимально возможной длины.

При использовании разделяемой памяти, особенно разделяемых библиотек, структура таблицы трансляции, которую я описал в начале этой главы, часто оказывается неудобна: с одной стороны, системе необходимо оставить некоторое свободное пространство между объектами, чтобы обеспечить их возможное увеличение (особенно это важно для стека и динамических данных); с другой стороны, это приведет к разреженным таблицам трансляции (часть дескрипторов в таблице не будет использоваться), т. е. к неэффективному использованию памяти таблицами.

Для решения этой проблемы ряд машин предоставляет двухступенчатую виртуальную память — сегментную адресацию, в которой каждый сегмент, в свою очередь, разбит на страницы, или даже ряд уровней сегментной и/или страничной трансляции. Это приносит ряд мелких преимуществ, например, позволяет давать права доступа сегментам, а подкачку с диска осуществлять постранично. Таким образом организована виртуальная память в IBM System 370 и ряде других больших компьютеров, а также в x86. Правда, в последнем виртуальная память используется несколько странным образом (подробное описание диспетчера памяти x86 приводится в *приложении 2*).

5.1. Сегменты, страницы и системные вызовы

О, порождение Земли и Тьмы, мы приказываем тебе отречься... — твердым, повелительным тоном начал Гальдер.

Смерть кивнул.

— Да, да, знаю я все это. Вызывали-то чего?

Т. Пратчетт

Реализовав страничную или сегментную виртуальную память, мы сталкиваемся с той же проблемой, о которой шла речь в разд. 4.5: пользовательские

программы не имеют доступа к адресным пространствам друг друга и к таблице дескрипторов, но ведь им надо иметь возможность вызывать системные сервисы и передавать им параметры!

Вторая проблема похожа на ситуацию, с которой мы сталкивались в разд. 2.6: если каждая задача имеет собственную таблицу трансляции, то что будет происходить с операционной системой в момент ее переключения? Ведь изменение адресного пространства будет приводить к неявной передаче управления в новое адресное пространство, поэтому при переключении таблицы трансляции (или при сбросе TLB) ядро ОС рискует потерять управление!

Существует два распространенных приема решения этих проблем, каждый из которых требует аппаратной поддержки.

Первый состоит в том, что УУП имеет две (или даже больше, чем две) таблицы трансляции. Одна таблица используется при работе в пользовательском режиме, другая — при переходе в режим супервизора. Дополнительные таблицы могут использоваться для ускорения переключения задач: вместо перезагрузки управляющих регистров УУП, система может просто переключать текущее адресное пространство.

Поскольку доступ к регистрам диспетчера памяти имеет только ядро, то оно может перезагружать управляющие регистры пользовательского режима без каких-либо ограничений. Определенную проблему при этом представляет передача параметров между пользовательской задачей и ядром при системных вызовах.

Виртуальная память в PDP-11

Уже упоминавшийся диспетчер памяти PDP-11 имеет два набора сегментных регистров: восемь для пользовательского режима и восемь для системного. В действительности, модели PDP-11 с диспетчером памяти имеют не только два набора регистров диспетчера памяти, но и два набора регистров общего назначения. Между системным и пользовательским режимами разделяется только слово состояния процессора. Для передачи параметров между пользовательским и системным адресным пространствами такие процессоры должны реализовать специальные команды. В PDP-11 это команды MFP1/MFPD (Move From Previous Instruction/Data — копировать [данные] из предыдущего [адресного пространства] кода/данных) и MTPI/MTPD (Move To Previous Instruction/Data — копировать [данные] в предыдущее [адресное пространство] кода/данных). Эти команды доступны только в режиме супервизора.

Адресные пространства и диспетчеры памяти процессоров SPARC

Процессоры SPARC используют для доступа к памяти данных команды LD и ST (LoAд и STore), которые, в действительности, имеют ряд специальных форм, позволяющих загружать операнды разных размеров и типов. Большинство этих форм имеет аналоги LDA (Load Alternate — загрузить из альтернативного [ад-

ресного пространства]) и STA (STore Alternate — сохранить в альтернативном [адресном пространстве]). Такие команды используют в качестве операнда восьмибитный идентификатор адресного пространства, который может быть либо восьмибитной константой, либо значением регистра ASI (Address Space Identifier — идентификатор адресного пространства).

ASI от 0 до 0x7F считаются привилегированными и доступны только в системном режиме процессора. ASI от 0x80 до 0xFF доступны в обоих режимах процессора, пользовательском и системном. Список идентификаторов адресных пространств, которые должны поддерживаться всеми реализациями SPARC, приведен в табл. 5.1.

Таблица 5.1. Идентификаторы адресных пространств SPARC

Значение	Name	Address space
00 ₁₆ .. 03 ₁₆	—	Определяется реализацией
04 ₁₆	ASI_NUCLEUS	Определяется реализацией
05 ₁₆ .. 0B ₁₆	—	Определяется реализацией
0C ₁₆	ASI_NUCLEUS_LITTLE	Определяется реализацией
0D ₁₆ .. 0F ₁₆	—	Определяется реализацией
10 ₁₆	ASI_AS_IF_USER_PRIMARY	Первичное адресное пространство, пользовательские полномочия
11 ₁₆	ASI_AS_IF_USER_SECONDARY	Вторичное адресное пространство, пользовательские полномочия
12 ₁₆ .. 17 ₁₆	—	Определяется реализацией
18 ₁₆	ASI_AS_IF_USER_PRIMARY_LITTLE	Первичное адресное пространство, пользовательские полномочия, little-endian
19 ₁₆	ASI_AS_IF_USER_SECONDARY_LITTLE	Вторичное адресное пространство, пользовательские полномочия, little-endian
1A ₁₆ .. 7F ₁₆	—	Определяется реализацией
80 ₁₆	ASI_PRIMARY	Первичное адресное пространство
81 ₁₆	ASI_SECONDARY	Вторичное адресное пространство
82 ₁₆	ASI_PRIMARY_NOFAULT	Первичное адресное пространство, не генерировать исключение

Таблица 5.1 (окончание)

Значение	Name	Address space
83_{16}	ASI_SECONDARY_NOFAULT	Вторичное адресное пространство, не генерировать исключение
$84_{16} \dots 87_{16}$	—	Зарезервировано
88_{16}	ASI_PRIMARY_LITTLE	Первичное адресное пространство, little-endian
89_{16}	ASI_SECONDARY_LITTLE	Вторичное адресное пространство, little-endian
$8A_{16}$	ASI_PRIMARY_NOFAULT_LITTLE	Первичное адресное пространство, не генерировать исключение, little-endian
$8B_{16}$	ASI_SECONDARY_NOFAULT_LITTLE	Вторичное адресное пространство, не генерировать исключение, little-endian
$8C_{16} \dots BF_{16}$	—	Зарезервировано
$C0_{16} \dots FF_{16}$	—	Определяется реализацией

Идентификаторы адресных пространств, имена которых содержат слово LITTLE, обозначают хранение многобайтовых значений в памяти в формате little-endian (младший байт находится по меньшему адресу), без этого слова — в формате big endian (старший байт по меньшему адресу). Таким образом, задачи, исполняющиеся под управлением одной ОС, могут использовать разные представления данных; при передаче параметров системным вызовам может происходить преобразование представления.

Адресные пространства AS_IF_USER позволяют осуществлять доступ из режима ядра с проверкой привилегий, так, будто доступ происходит от имени пользователя. При использовании раздельных адресных пространств пользовательской задачи и ядра это не требуется, но если система отображена в адресное пространство пользователя, он мог бы передать в качестве параметра системного вызова страницу, доступную только супервизору. Система при этом могла бы ошибочно перезаписать данные ядра.

Слово NOFAULT обозначает режимы, используемые для спекулятивной загрузки данных из памяти. При доступе в режиме NOFAULT доступ к несуществующим страницам или страницам, доступ к которым запрещен, не приводит к генерации исключения. Вместо этого диспетчер памяти имитирует чтение нулевого значения из ОЗУ. Этот режим доступа используется, главным образом, для оптимизации спекулятивного исполнения, при котором процессор пытается выполнять команды раньше, чем они были закодированы.

Так, при исполнении кода `if (a!=NULL) { b=a->field1; }`, попытка процессора выполнить тело условного оператора до проверки условия могла бы привести к ошибке защиты. Однако если компилятор предусмотрительно поро-

дил код, в котором загрузка данных по указателю осуществляется через `ASI_PRIMARY_NOFAULT`, этого не произойдет. Прочитанный ошибочный результат будет отброшен, когда условие все-таки будет проверено.

При трансляции адреса процессор передает диспетчеру памяти собственно адрес, тип доступа (чтение, запись или атомарная операция чтения-записи), идентификатор текущего адресного пространства, режим работы процессора (пользовательский или привилегированный), а также то, является ли адрес адресом команды или данных.

Архитектура диспетчера памяти не определена спецификациями SPARC; однако машины, поставлявшиеся компанией Sun в разное время, использовали один из трех типов диспетчеров памяти:

1. Sun 4 — 32-разрядный виртуальный адрес, 4-килобайтные страницы и программный синтез записей TLB. Такие устройства использовались как с процессорами SPARC, так и с процессорами MC68010/68020.
2. Sun 4c — 32-разрядный виртуальный адрес и трехуровневая аппаратная трансляция адреса, но с возможностью отключить эту трансляцию и вернуться к программному синтезу TLB. Отключая уровни трансляции, можно получать 8-килобайтные, 512-килобайтные и 4-мегабайтные страницы. Это устройство также известно как SRMMU v8 (SPARC v8 Reference MMU — референтная [реализация] УУП для SPARC v8). Описание этого устройства может быть найдено в документе [[solutions.sun.com 802-7100-01](http://solutions.sun.com/802-7100-01)].
3. Sun 4u — 64-разрядный виртуальный адрес, программный синтез записей в TLB, размеры страниц 8, 64 или 512 Кбайт или 4 Мбайта. Данное устройство известно также как SRMMU v9. В действительности, это несколько разных устройств с разными объемами адресных пространств: диспетчер памяти UltraSparc I/II (Cheetah) имел 42-разрядный виртуальный адрес, который расширялся со знаком до 64 бит, и относительно небольшие объемы TLB; диспетчер памяти UltraSparc III (Spitfire) использует полный 64-битный виртуальный адрес и значительно увеличенные TLB. Детальное описание этого устройства может быть найдено в документе [[www.sun.com USIIIv2.pdf](http://www.sun.com/USIIIv2.pdf)].

Рассмотрим архитектуру диспетчера памяти Sun 4u Spitfire. Как уже говорилось, это устройство поддерживает 64-битные виртуальные адреса и транслирует их в 42-битные физические адреса. Размер страницы может составлять 8 Кбайт (51 бит — селектор страницы, 13 бит — смещение), 64 Кбайт, 512 Кбайт или 4 Мбайта (размеры соответствующих битовых полей читатель может вычислить самостоятельно). Трансляция адресов данных и команд осуществляется двумя независимыми подсистемами УУП.

Страница идентифицируется селектором, который имеет длину 51 бит (если размер страницы больше 8 Кбайт, младшие биты селектора равны 0), и 13-битным идентификатором контекста. УУП содержит два управляющих регистра, в которых записаны идентификаторы контекстов, соответствующие `ASI_PRIMARY` и `ASI_SECONDARY`. Кроме того, контекст 0 зарезервирован для супервизора (`nucleus`). Для всех трех контекстов в управляющих регистрах УУП также задан размер страницы.

Отображение передаваемого процессором ASI на идентификатор контекста происходит по довольно простым правилам. Все ASI, перечисленные в табл. 5.1, в имени которых содержится `NUCLEUS`, используют контекст 0. Все ASI, в имени которых содержится `PRIMARY`, транслируются через контекст

PRIMARY; все ASI, в имени которых содержится SECONDARY, транслируются через контекст SECONDARY.

Кроме того, УУП поддерживает еще более десятка привилегированных ASI, адресация с использованием которых не приводит к страницной трансляции (полный их список слишком обширен, чтобы приводить его в этой книге). Несколько из этих ASI используются для доступа к регистрам самого УУП. Большая часть регистров доступна через ASI 0x58; ASI 0x5C-0x5F используются для доступа к TLB и его модификации. Определена также довольно большая группа ASI, используемых для прямого доступа к физическому адресному пространству в обход УУП и для доступа к адресному пространству ввода-вывода.

Определив на основании ASI, что трансляция необходима, УУП определяет соответствующий номер контекста (первичный, вторичный или супервизор) и длину страницы, и формирует из номера контекста и селектора страницы 64-битный тег. По этому тегу осуществляется поиск в ассоциативной памяти TLB.

Для трансляции адресов данных используются два больших TLB по 512 записей и один малый с 16 записями. Для трансляции адресов кода используются один TLB емкостью 128 записей и один — с емкостью 16 записей. Записи малых TLB могут блокироваться (заблокированные записи сохраняются при общем сбросе TLB и не могут быть объявлены жертвой при поиске свободной записи), записи больших TLB — нет. Кроме того, все записи каждого из больших TLB должны соответствовать страницам одинакового размера, а малые TLB могут содержать дескрипторы страниц произвольного размера.

Как правило, малые TLB используются для системных страниц, прежде всего для кода и данных обработчика исключений УУП.

Поиск во всех трех TLB данных и в обоих TLB команд происходит параллельно. Если эти TLB содержат дублирующиеся теги или теги, соответствующие перекрывающимся адресам с разными размерами страниц, поведение УУП непредсказуемо; в интерфейсе УУП не предусмотрено механизмов, которые можно было бы использовать для оповещения системы о такой ошибке. В принципе, у программного обеспечения даже есть возможность разместить две или более записей с одинаковыми тегами в одной TLB — поведение УУП в этой ситуации также непредсказуемо, гарантируется только, что таким образом невозможно физически повредить УУП. Предотвращение таких ситуаций возлагается на операционную систему.

Поскольку трансляция обращений к коду и данным осуществляется через разные TLB, совпадающие виртуальные адреса для кода и данных (даже принадлежащих к одному контексту) могут отображаться на разные страницы физической памяти, т. е. ОС может (но не обязана) реализовать раздельные адресные пространства для кода и данных.

Если тег был найден в TLB, то из буфера извлекается соответствующее этому тегу значение — дескриптор страницы, который в документации называется TTE (Translation Table Entry — запись таблицы трансляции).

- Бит 63 (старший бит дескриптора) обозначается *v* (Valid). Если этот бит равен нулю, то обращение к этой странице приведет к генерации исключения TLB miss (промах TLB).
- Биты 61 и 62 кодируют один из четырех допустимых размеров страницы: значение 00 соответствует размеру страницы 8 Кбайт, 01 — 64 Кбайт, 10 — 512 Кбайт, 11 — 4 Мбайт.

- Бит 60 обозначается NFO (No Fault Only) — страницы с этим битом доступны только в ASI NO_FAULT, доступ к ним из других ASI приводит к исключению по ошибке трансляции.
- Бит 59 обозначается IE (Invert Endiannes) — при доступе к странице с этим битом происходит обращение порядка байт по отношению к тому, который был задан в ASI. Такие страницы могут использоваться для сегментов ОЗУ, разделяемых между процессами, использующими разный порядок байт.
- Биты 50-58 могут использоваться ОС для хранения собственных данных, например бита отсутствия, clock-бита, битов защиты от исполнения и т. д. ОС может хранить эти данные в образе дескриптора в ОЗУ, но при записи в TLB соответствующие биты дескриптора игнорируются и могут бытьброшены при последующем чтении дескриптора из TLB.
- Биты 43-49 зарезервированы, скорее всего, для расширения физического адреса страницы.
- Биты 13-42 содержат физический адрес страницы. Для страниц, размер которых больше 8 Кбайт, младшие биты адреса игнорируются при трансляции, но при чтении TLB значения этих битов сохраняются.
- Биты 7-12, аналогично битам 50-58, доступны для программного использования.
- Бит 6 обозначается L (Lock) — блокировка. Соответствующие записи малого TLB не будут рассматриваться в качестве жертвы при поиске свободного TLB. В больших TLB этот бит игнорируется.
- Биты 3, 4 и 5 управляют кэшированием страницы. Бит 3 обозначает, что запись в эту страницу приводит к побочным эффектам (т. е., скорее всего, что эта страница отображается не на ОЗУ, а на регистры внешнего устройства).
- Бит 2 обозначается P (Privileged) и обозначает, что страница доступна только в привилегированном режиме процессора.
- Бит 1 обозначается W (Writable) и обозначает, что страница доступна для модификации.
- Бит 0 (младший бит дескриптора) обозначается G (Global). Страницы, обозначаемые этим битом, доступны во всех контекстах УУП.

Если запись в TLB не найдена или найдена, но имеетброшенный бит V, происходит исключение промаха TLB (TLB miss). При этом вызывается определяемая операционной системой программа-обработчик. При вызове обработчика УУП размещает в одном из своих регистров виртуальный адрес, который вызвал промах, и его контекст, так что обработчику не надо тратить время на поиск этой информации.

Страницы, в которых размещены код и данные этого обработчика, должны всегда находиться в TLB. Процессоры SPARC допускают рекурсивный вызов обработчиков исключений, в отличие, например, от процессоров x86, у которых на этот случай предусмотрено специальное исключение двойного отказа (Double Fault). Однако глубина рекурсии ограничена глубиной аппаратного стека исключений, которая не очень велика; переполнение этого стека приводит к переводу процессора в специальное состояние RED (Reset, Error and Debug —брос, ошибка и отладка), в котором обычно происходит останов системы.

Как правило, дескрипторы страниц обработчика промаха TLB блокируются в малых TLB; рекомендуется также блокировать эти страницы в кэше первого уровня — во всяком случае, страницы, которые содержат наиболее часто использующиеся ветви кода.

Обработчик промаха TLB должен найти или сочинить дескриптор страницы и поместить его в TLB, или сгенерировать ошибку доступа к странице. Как правило, дескрипторы страниц все-таки не сочиняются на ходу, а берутся из многоуровневых разреженных таблиц трансляции или хэш-таблиц, размещенных в ОЗУ. При этом обработчик может связать с дескриптором дополнительные флаги, которые другие УУП реализуют аппаратно; такие как бит отсутствия, бит защиты от исполнения (его наличие игнорируется при обращении к данным и приводит к ошибке доступа к странице при обращении к коду) и т. д.

Описание структуры, используемой ОС Solaris, лишь недавно появилось в открытых источниках [[blogs.sun.com paulsan 14 June 2005](http://blogs.sun.com/paulsan/14_June_2005)], и оно довольно фрагментарно, хотя исследователь с большим запасом свободного времени может изучить исходные тексты OpenSolaris, доступные на сайте opensolaris.org. Вместо многоуровневой таблицы, Solaris использует глобальную хэш-таблицу, индексом в которой является сочетание селектора 64-килобайтного сегмента (если используются 8-килобайтные страницы) или "длинной" страницы (если используются страницы длиной 64 Кбайт или больше) и идентификатора процесса. Дескриптор сегмента называется `hme_blk` и описывает блок переменной длины, состоящий из нескольких последовательных страниц. При нормальной работе Unix-системы многие сегменты разделяются между процессами (это касается не только сегментов кода, но во многих случаях и сегментов данных), поэтому собственно хэш содержит не сами структуры `hme_blk`, а указатели на них, а структура `hme_blk`, в свою очередь, содержит счетчик ссылок. Когда процесс завершается или освобождает сегмент, счетчик ссылок в соответствующем `hme_blk` уменьшается. Когда его значение становится нулевым, `hme_blk` удаляется.

Для ускорения поиска свободного TLB УУП имеет аппаратно реализованный алгоритм поиска жертвы с помощью довольно-таки грубого генератора псевдо-случайных чисел. Этот генератор представляет собой двоичный счетчик, который увеличивается на каждом такте ЦПУ. Запись TLB, номер которой равен значению счетчика, объявляется жертвой. Если разработчики ОС желают использовать более сложный алгоритм поиска жертвы, никто не запрещает это сделать, но осуществить это придется полностью программными средствами.

Также УУП предоставляет область памяти, называемую TSB (Translation Storage Buffer — буфер памяти [для] трансляции). Теоретически эта память может размещаться где угодно в ОЗУ и иметь практически произвольный размер, но на практике рекомендуется заблокировать эту область в кэше данных второго уровня, а при выборе ее размера учитывать общий размер этого кэша. УУП предоставляет аппаратные средства, позволяющие интерпретировать TSS как хэш-таблицу, в которой в качестве ключа используются теги TLB, а в качестве значений — TTE. Таким образом, УУП позволяет создать своеобразный программно управляемый кэш второго уровня для дескрипторов страниц с ассоциативным доступом. Впрочем, никто не может заставить ОС пользоваться именно этим механизмом для поиска дескрипторов.

Операционная система Sun Solaris для процессоров SPARC использует раздельные адресные пространства. Пользовательская задача видит полное

32- или 64-разрядное адресное пространство. Несмотря на наличие ASI для двух пользовательских адресных пространств (первичного и вторичного), оба эти адресных пространства в Solaris указывают на один и тот же контекст УУП.

Разные идентификаторы контекстов соответствуют разным процессам. Во многих случаях планировщику достаточно перегрузить регистры контекста УУП, чтобы сменить пользовательское адресное пространство — разумеется, если TTE данного контекста отсутствуют в TLB, это приведет к промахам TLB. Если количество активных процессов в системе превосходит количество доступных контекстов (8195), планировщик оказывается вынужден поддерживать "рабочее множество" процессов, у которых есть собственный контекст. Простой подсчет показывает, что 1024 записей больших TLB не хватает на все допустимые контексты, т. е. большинство процессов, принадлежащих к этому множеству, не могут иметь активных записей в TLB.

При планировании процесса, не принадлежащего рабочему множеству, ОС вынуждена назначить один из активных процессов "жертвой", отобрать у него идентификатор контекста (в исходных текстах ядра это называется *context stealing* — кража контекста) и сбросить соответствующие записи в TLB, если они там существуют. Для этого УУП предоставляет режим частичного сброса TLB с указанием идентификатора контекста.

Системный вызов осуществляется с помощью команды Tcc (Trap conditional — условное прерывание). Эта команда вызывает программное прерывание с указанным номером; фактически, она переводит процессор в системный режим и вызывает процедуру, адрес которой находится в таблице векторов прерываний. Номер системного вызова определяется операндом команды Tcc и при ее исполнении сохраняется в специальном регистровом стеке ТТ. Скалярные параметры и указатели передаются в регистрах; разрешение указателей осуществляется командами Load Alternative/Store Alternative с использованием вторичного адресного пространства.

Использование нескольких адресных пространств усложняет диспетчер памяти. Кроме самого дублирования управляющих регистров и/или появления новых сущностей (идентификатора контекста и ASI), значительно усложняется структура TLB: каждая запись в TLB должна хранить не только виртуальный адрес, к которому она относится, но и идентификатор контекста, к которому относится этот адрес.

Данный недостаток имеет и оборотную сторону, которая, в свою очередь, является преимуществом: переключив задачу, планировщик не обязан сбрасывать TLB; диспетчер памяти сам "сообразит", что часть записей кэша относится к предыдущему адресному пространству, и будет расценивать их как инвалидные записи, при попадании на которые дескриптор страницы необходимо подкачать из памяти. Более того, если старая задача вскоре вновь получит управление, то какие-то из ее записей в TLB могут сохраниться, что, разумеется, могло бы повысить среднюю производительность системы; при таких объемах TLB, как у современных реализаций Sun 4u, это довольно часто случается на практике.

На первый взгляд, такое решение обладает еще одним недостатком: ядро, обращаясь к параметрам системного вызова, должно использовать какие-то специальные команды. Как мы увидим далее, на практике этот недостаток не играет большой роли: наиболее известное из альтернативных решений также приводит к необходимости использовать специальные команды для передачи параметров системных вызовов по ссылке.

Альтернативное решение, применяемое в ряде других процессоров, — в том числе в таких распространенных процессорах, как x86, — состоит в том, что таблица трансляции в каждый момент времени одна. Это означает, что ядро и пользовательская задача разделяют общее адресное пространство. При этом страницы и/или сегменты ядра защищены от пользовательских задач на уровне прав доступа в дескрипторах страниц. Обычно дескриптор страниц имеет бит супервизора, наличие которого означает, что страница доступна только из системного режима. Благодаря этому, пользователь может сформировать указатель на объекты ядра, но не может обратиться к такому объекту (внимательный читатель, наверное, заметил, что диспетчер памяти Sun 4i также имеет механизмы, которые могли бы пригодиться при таком использовании и не нужны в раздельных адресных пространствах, — бит супервизора в TTE и ASI ..._AS_IF_USER_).

Диспетчер памяти в таких системах обычно более прост, чем в системах, реализующих раздельные пространства; в частности, простота выражается в том, что записи TLB не имеют никаких идентификаторов адресного пространства, поэтому либо переключение таблицы дескрипторов неявно приводит к сбросу всех TLB, либо после каждого переключения все TLB надо сбрасывать вручную. Некоторые диспетчеры памяти допускают более тонкое управление TLB, при котором сбрасываются записи кэша, относящиеся к пользовательским адресам, но сохраняются записи, относящиеся к ядру.

Ядро в таких системах отображено в адресные пространства всех задач. Поскольку только ядро имеет доступ к управляющим регистрам УУП, то только оно и может переключать адресные пространства. Таким образом, ядро оказывается аналогом менеджера банков памяти в системах с банковой адресацией: оно присутствует во всех "банках" и поэтому переключение адресного пространства не приводит к неявной передаче управления.

Наиболее очевидный недостаток этого решения состоит в том, что адресное пространство задачи оказывается меньше теоретически возможного адресного пространства, допускаемого архитектурой процессора.

Так, в Linux/x86 три гигабайта адресного пространства доступны пользовательской задаче, а четвертый гигабайт занят ядром. В Solaris/x86 под ядро отведено всего 256 Мбайт. На практике, этот недостаток трудно назвать очень уж важным: действительно, задачи, потребности которых в памяти

приближаются к четырем гигабайтам, лучше перекомпилировать для исполнения в 64-битном режиме.

Адресное пространство в Win32

В Windows NT 32-битное адресное пространство разбито на две части. Первые два гигабайта выделены под пользовательские код и данные. В верхней части этого адресного пространства находится относительно небольшая область, в которой размещены разделяемые системные DLL. Вторые два гигабайта зарезервированы для ядра. Разумеется, ядро на практике занимает лишь небольшую часть этого пространства, поэтому в Windows 2000 была предпринята попытка перераспределить ресурсы.

Если ядро Windows 2000 и более поздних версий NT (XP и 2003) загрузить с ключом /3GB, то пользовательской программе будет выделено три гигабайта адресного пространства, а под ядро оставлен всего один гигабайт. К сожалению, тестирование этого режима привело к обнаружению некоторых неприятных фактов. А именно, оказалось, что многие пользовательские программы, написанные на языке С, неправильно обрабатывают переполнение при арифметических операциях над указателями, и поэтому могут некорректно работать с объектами, адреса которых превосходят 0x7FFFFFFF. С другой стороны, многие драйверы устройств и другие модули ядра, разработанные третьими фирмами, не были хорошо протестированы и потому некорректно работали в "урезанном" ядре. Поэтому режим /3GB по умолчанию выключен и включается только если администратор системы явно задаст его в файле C:\BOOT.INI. Во вторых, даже в системе, загруженной с ключом /3GB, новая структура адресного пространства используется только для задач, собранных с ключом /LARGEADDRESSAWARE (этот ключ линкера приводит к установке соответствующего флага в заголовке EXE-файла). Все остальные задачи загружаются с традиционной структурой адресного пространства.

Тем не менее все драйверы ядра теперь в обязательном порядке тестируются на совместимость с ключом /3GB. По сообщению Рэймонда Чена [blogs.msdn.com theoldnewthing], это уже приводит к проблемам при разработке видеодрайверов: так, к моменту выхода второго издания книги на рынке уже есть видеокарты с гигабайтом видеопамяти и вскоре можно ожидать появления устройств с несколькими гигабайтами ОЗУ. Разумеется, такой видеобуфер нельзя целиком отобразить в адресное пространство "урезанного" ядра и приходится возвращаться к архаичному механизму отображения видеопамяти по "банкам", подобно тому, как это делалось в старых VGA-адаптерах.

Для сравнения, ни в Linux/x86, ни в Solaris/x86 этой проблемы не существует: в этих ОС реальная работа с видеобуфером осуществляется не модулем ядра, а отдельным процессом, так называемым сервером X Window. Прикладные программы, желая нарисовать что-то на экране, обращаются не к ядру, а к этому процессу, используя специальный протокол X 11. Аналогичная архитектура графической подсистемы (GDI) использовалась и в старых версиях Windows NT, но в Windows NT 4.0, якобы по соображениям производительности, графическая подсистема была включена в ядро. Трудно сказать, насколько эти соображения были верными: производительность Windows NT 3.51, конечно, оставляла желать много лучшего, но дело тут, по-видимому, не только в том, что запросы к GDI передавались через средства межзадачного взаимодействия: наблюдаемая производительность графической подсистемы ОС семейства Unix на том же оборудовании обычно была более или менее приемлемой.

На первый взгляд, единое адресное пространство ядра и пользовательской задачи сильно упрощает не только диспетчер памяти, но и передачу параметров системным вызовам. Код системного вызова мог бы просто обращаться к данным, указатель на которые был передан пользователем без всяких специальных команд обращения к альтернативному адресному пространству. Специальная команда системного вызова нужна только для контролируемого повышения уровня привилегий, но не для переключения адресного пространства.

В действительности, не все так просто. Как мы видели, пользователь может сформировать указатель в адресное пространство ядра. Да, сам он не может воспользоваться этим указателем, но он может передать его в качестве параметра ядру, которое работает в привилегированном режиме и имеет доступ к своим собственным коду и данным.

Например, пользователь мог бы попросить систему прочитать данные из файла в буфер в памяти и передать в качестве указателя на этот буфер указатель на какую-то из системных структур данных или даже на сегмент кода ядра. Исполнение этого запроса могло бы привести к разрушению ядра или даже к исполнению троянского кода с системным уровнем привилегий.

Передав ядру указатель "в никуда" — на недопустимую страницу — пользователь мог бы спровоцировать ошибку защиты при исполнении системного вызова.

Поэтому, прежде чем разыменовывать переданный пользователем указатель, любая системная процедура должна убедиться, что он указывает на страницу, которая доступна пользователю от его собственного имени. Поэтому все процессоры, использующие такой подход к защите памяти, имеют специальные средства (как правило, специальные команды), с помощью которых код ядра может исполнять отдельные операции с памятью с пониженным уровнем привилегий или проверять, каковы права доступа к памяти с данным адресом. Модули ядра всегда должны пользоваться этими средствами, обращаясь к пользовательским данным, указатели на которые были получены в параметрах системного вызова.

Так, именно для этой цели в процессорах SPARC реализована группа привилегированных ASI, имена которых начинаются с `ASI_AS_IF_USER`.

На практике, в обоих типах ОС — как в системах, использующих раздельные адресные пространства ядра и пользователя, так и в системах с отображением ядра в адреса пользователя — разработчикам модулей ядра настоятельно не рекомендуется самостоятельно работать с пользовательским адресным пространством. Вместо этого разработчикам предоставляются специальные процедуры или макроопределения, с помощью которых можно копировать данные между пользовательским и системным сегментами данных. В теле этих

процедур используются правильные команды и делаются все необходимые проверки.

Используя такие процедуры, можно разрабатывать модули ядра, которые могут переноситься между разными версиями ОС простой перекомпиляцией. Так, мы видели, что Solaris/SPARC и Solaris/x86 применяют разные подходы к защите памяти, но модули ядра (в том числе и драйверы устройств PCI), написанные на языке C или C++ с использованием правильных процедур для передачи параметров, могут быть скомпилированы для работы под каждой из версий ОС без изменений исходных текстов. Разумеется, модули ядра, написанные на ассемблере, таким образом переносить нельзя, но большая часть системных программ в наше время пишется на языке C.

Виртуальная память и режимы процессора VAX

Миникомпьютеры VAX имеют четыре режима работы процессора (в порядке возрастания прав доступа):

- режим пользователя (User);
- режим супервизора (Supervisor);
- режим исполнителя (Executive);
- режим ядра (Kernel).

Текущий режим определяется битами 22 и 23 в слове состояния процессора (рис. 5.5). Переключение режима работы сопровождается переключением указателя стека. То есть в действительности у VAX четыре регистра указателя стека, но остальные регистры общего назначения общие для всех режимов.

Операционная система VAX/VMS использует три из доступных режимов (пользовательский, исполнительный и ядра), а BSD Unix — только два.

Каждая страница адресного пространства может иметь различные права доступа для разных режимов. При этом соблюдаются следующие правила:

1. Допустимы только права чтения и записи.
2. Наличие права записи предполагает наличие права чтения.
3. Каждый более привилегированный режим всегда имеет те же права, которые имеют менее привилегированные режимы, плюс, возможно, какие-то еще. Такая организация доступа называется *кольцами защиты*.

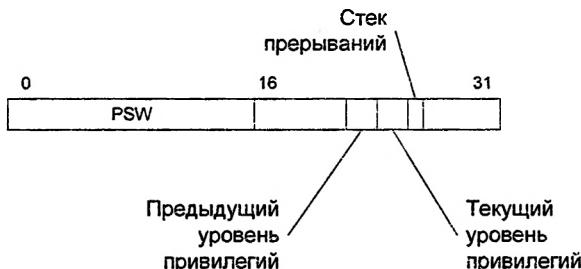


Рис. 5.5. Слово состояния процессора VAX

Таблица 5.2. Коды защиты для различных режимов доступа процессора VAX
Цитируется по [Прохоров, 1990]

Код	Kernel	Executive	Supervisor	User
0000	—	—	—	—
0001	Не предсказуем			
0010	RW	—	—	—
0011	R	—	—	—
0100	RW	RW	RW	RW
0101	RW	RW	—	—
0110	RW	R	—	—
0111	R	R	—	—
1000	RW	RW	RW	—
1001	RW	RW	R	—
1010	RW	R	R	—
1011	R	R	R	—
1100	RW	RW	RW	R
1101	RW	RW	R	R
1110	RW	R	R	R
1111	R	R	R	R

R — право чтения, W — право записи, — — отсутствие прав.

Все допускаемые этими правилами комбинации прав могут быть закодированы с помощью четырех битов в дескрипторе страницы (табл. 5.2). Переключение режимов осуществляется командами СНМК, СНМЕ, СНМС и СНМУ. Эти команды помещают слово состояния процессора и счетчик команд в стек, соответствующий новому режиму, сохраняют предыдущий режим в специальном поле слова состояния (зачем нужно сохранять режим в двух местах, мы поймем чуть позже), устанавливают новый режим и, наконец, передают управление по фиксированному адресу, аналогично команде SYSCALL в системах с двумя уровнями доступа. Передача управления по фиксированному адресу позволяет нам защититься от бесконтрольного повышения уровня доступа, а все предыдущие операции дают возможность вернуться на предыдущий уровень доступа с одновременной передачей управления, используя специальную команду REI, которая восстанавливает и счетчик команд, и слово состояния.

Переключение стека в момент переключения уровня привилегий необходимо для того, чтобы избежать манипуляций содержимым стека со стороны пользовательского кода. Действительно, если бы старое слово состояния процессора

сохранялось бы в пользовательском стеке, в многопоточной системе можно было бы реализовать следующий сценарий:

1. Пользовательская задача вызывает ядро. Слово состояния сохраняется в ее стеке.
2. Планировщик переключается на другую нить той же задачи. Эта нить изменяет биты в сохраненном слове состояния и/или адрес возврата.
3. Ядро возвращает управление и восстанавливает измененное слово состояния.
4. Первая из нитей пользовательской задачи получает управление с повышенным уровнем привилегий.

Команды СНМ(Х) обычно используются для повышения уровня доступа, а REI может использоваться только для его понижения или возврата на тот же уровень (рис. 5.6).

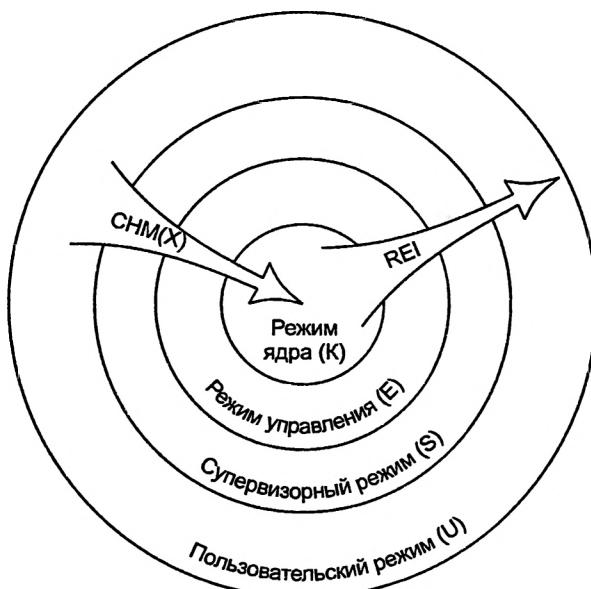


Рис. 5.6. Переключение режимов процессора VAX

32-битное адресное пространство процессора VAX разбито на четыре части, каждая объемом по гигабайту. Первый гигабайт адресов предназначен для кода и данных пользовательской программы, второй — для пользовательского стека, третий — для системы, четвертый не используется (рис. 5.7). Каждая из частей имеет собственный указатель на таблицу дескрипторов страниц. Важно отметить, что деление адресного пространства на таблицы не обязательно связано с правами доступа на отдельные страницы — в системной таблице могут быть страницы, доступные для записи из пользовательского режима (на практике этого никогда не бывает, но на уровне диспетчера памяти контроль за этим не реализовано), а в пользовательской — страницы, доступные только ядру.

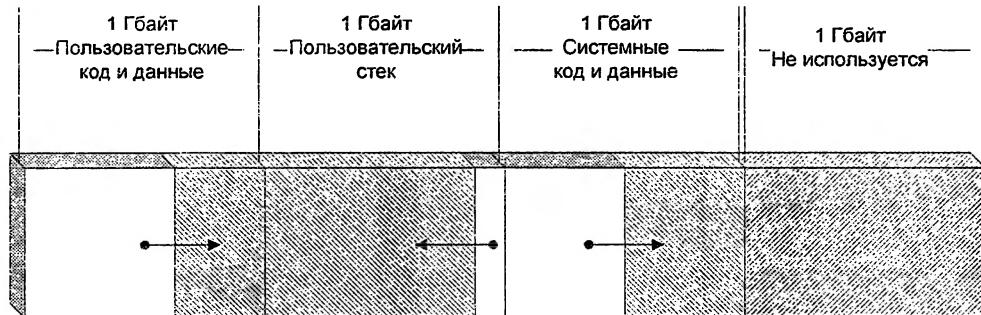


Рис. 5.7. Адресное пространство VAX

Системная таблица страниц одна во всей системе и содержится в адресных пространствах всех задач. Напротив, пользовательские таблицы у каждой задачи свои (внимательный читатель может отметить определенную параллель между этой структурой и описанным в разд. 2.6 переключателем банков памяти, который присутствует во всех банках). Для того чтобы упростить системе управление пользовательскими таблицами дескрипторов, эти таблицы хранятся не в физическом, а в виртуальном системном адресном пространстве, и при доступе к ним происходит двойная трансляция адреса.

Ядро системы, таким образом, присутствует в адресных пространствах всех задач. Многие системные модули (например, функция для получения текущего реального времени) доступны для чтения из пользовательского режима и могут вызываться непосредственно, как обычные процедуры. Адреса точек входа этих процедур размещены в специальной таблице в начале системного адресного пространства (рис. 5.8). Другие системные модули (например, подсистема работы с файлами, RMS — *Record Management Service* (Служба управления записями)) требуют повышения уровня доступа: действительно, если одна из задач работает с файлами с ограниченным доступом, было бы неразумно позволять всем остальным задачам видеть используемые при этом системные буферы. Точки входа этих процедур размещаются в той же таблице, что и прямо вызываемые системные подпрограммы, но тела этих процедур состоят только из двух команд: переключения режима процессора и возврата.

Процедура, работающая в "повышенном" (более привилегированном) режиме процессора, имеет полный доступ ко всем данным режимов с более низким уровнем доступа. Благодаря этому мы можем передать привилегированной процедуре указатель, и она доберется до наших данных простым разрешением этого указателя, без каких бы то ни было специальных команд.

Впрочем, при таком подходе возникает определенная проблема. Поскольку система и пользователь находятся в одном адресном пространстве, пользователь может "подсунуть" системе указатель на страницу, к которой сам не имеет доступа — например, попросить считать нечто из файла в системный сегмент данных. Для исключения таких ситуаций VAX предоставляет команды PROBER и PROBEW, которые проверяют, существует ли доступ к указанной странице в предыдущем режиме работы процессора. Как мы помним, предыдущий режим сохраняется не только в стеке, но и в слове состояния процесса, и нужно это именно для таких проверок.

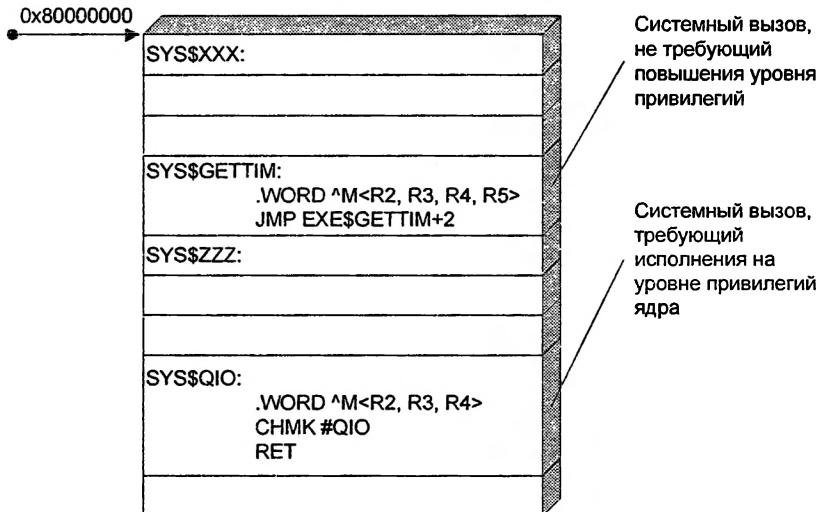


Рис. 5.8. Точки входа системных подпрограмм VAX/VMS

Видно, что обойтись без специальных команд все-таки не удалось. К тому же платой за принятое в VAX техническое решение оказалось сокращение полезного адресного пространства задачи в два, а на самом деле даже в четыре (ко-му нужен стек размером 1 Гбайт?) раза. В 70-е годы, когда разрабатывался VAX, это еще не казалось проблемой.

Уровни доступа 80286

Чуть дальше в близком направлении продвинулись разработчики 80286: у этого процессора уровень доступа определяется старшими двумя битами селектора сегмента (рис. 5.9). Код, исполняющийся в сегменте с уровнем доступа 2, имеет доступ ко всем сегментам своего и более низких уровней. Межсегментный переход с повышением уровня доступа возможен лишь через сегменты со специальным дескриптором, так называемые *шлюзы* (gate).

В этой архитектуре для проверки прав доступа к сегменту в предыдущем режиме работы не нужны специальные команды, достаточно проверки селектора сегмента.

Адрес

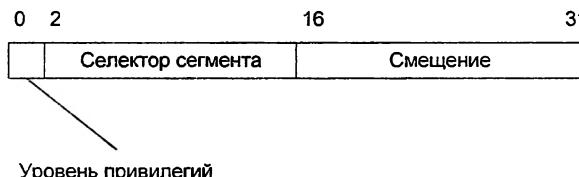


Рис. 5.9. Структура адреса процессора i80286

Процессор имеет две таблицы дескрипторов сегментов — GDT (Global Descriptor Table — глобальная таблица) и LDT (Local Descriptor Table — локальная таблица). Третий бит селектора сегмента определяет, в какой из таблиц искать дескриптор соответствующего сегмента. Предполагалось, что в таблице GDT будут размещены дескрипторы разделяемых сегментов, а в таблице LDT — дескрипторы сегментов, локальных для задачи. Таким образом, при переключении задачи достаточно перегрузить указатель на LDT. Разумеется, код диспетчера задач, осуществляющего эту операцию, должен размещаться в сегменте из GDT.

80286, хотя и предоставлял почти полноценную сегментную адресацию, не имел сегментных отказов, поэтому использовать все преимущества виртуальной памяти на этом процессоре было невозможно. Вторым недостатком было отсутствие режима совместимости с 8086 в защищенном режиме — не существовало возможности создать такую таблицу трансляции, которая бы воспроизвела специфическую структуру адресного пространства этого процессора. Отчасти это было обусловлено и использованием битов в селекторе сегмента для задания прав доступа. Обе ОС, которые разрабатывались для этого процессора, — Win16 и OS/2 1.x — большого успеха не имели.

OS/2 использует три режима доступа: пользовательский, системные DLL и собственно ядро (рис. 5.10). Windows NT (которая начинала свою карьеру как OS/2 New Technology) первоначально проектировалась как переносимая ОС. Требование переносимости на RISC-архитектуры с двумя уровнями привилегий заставило разработчиков отказаться от уровня системных DLL. Позже фирма

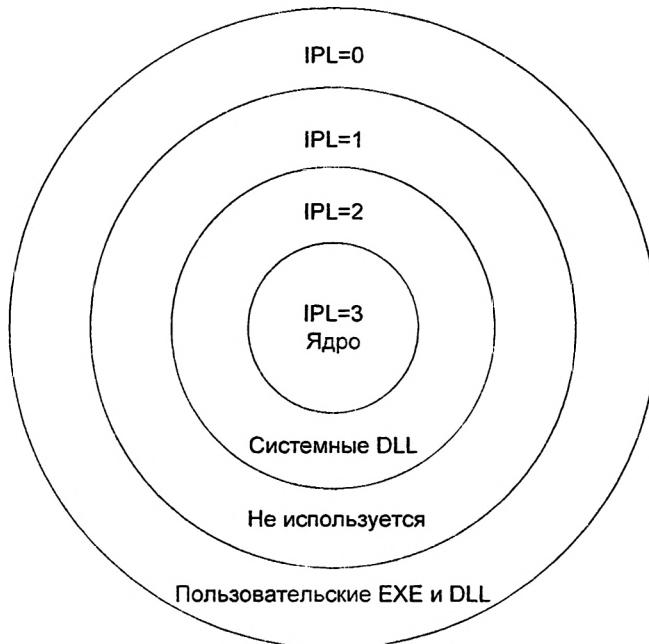


Рис. 5.10. Уровни доступа в OS/2

Microsoft постепенно отказалась от поддержки всех аппаратных архитектур, кроме x86 (дольше всех они держались за DEC Alpha), но двухуровневая структура доступа так и осталась в новых версиях этой системы — Windows 2000/XP/2003.

Системы семейства Unix используют x86 как нормальную 32-разрядную машину с двухуровневым доступом: пользовательской задаче выделяется один сегмент, ядру — другой. 4-гигабайтowego сегмента x86, разбитого на страницы размером по 4 Кбайт, достаточно для большинства практических целей. Например, в Linux системный вызов исполняется командой `Int 80h`. Селектор пользователяского сегмента помещается в регистр `FS`. Для доступа к этому сегменту из модулей ядра используются процедуры `memcpy_from_fs` и `memcpy_to_fs`.

Многоуровневый доступ, основанный на концепции колец, не имеет принципиальных преимуществ по сравнению с двумя уровнями привилегий. Как и в двухуровневой системе, пользовательские модули вынуждены полностью доверять привилегированным, привилегированные же модули не могут защищаться даже от ошибок в собственном коде. Самое лучшее, что может сделать Windows XP, обнаружив попытку обращения к недопустимому адресу в режиме ядра, — это нарисовать на синем экране дамп регистров процессора. В OS/2 фатальная ошибка в привилегированных модулях, исполняемых во втором кольце защиты, не обязательно приводит к остановке ядра, но подсистема, в которой произошла ошибка, оказывается неработоспособна. Если испорчен пользовательский интерфейс или сетевая подсистема, система в целом становится бесполезной и нуждается в перезагрузке.

Кроме того, разделение адресных пространств создает сложности при разделении кода и данных между процессами: разделяемые объекты могут оказаться отображены в разных процессах на разные адреса, поэтому в таких объектах нельзя хранить указатели (*подробнее см. разд. 5.4*). Стремление обойти эти трудности и создать систему, в которой сочетались бы преимущества как единого (легкость и естественность межпроцессного взаимодействия), так и раздельных (защита процессов друг от друга) адресных пространств многие годы занимало умы разработчиков аппаратных архитектур.

Например, машины фирмы Burroughs предоставляли пользователю и системе единое адресное пространство, разбитое на сегменты (единую таблицу дескрипторов сегментов). В отличие от ранее рассматривавшихся архитектур, в этих машинах в адресное пространство каждой задачи было отображено не только ядро, но и сегменты кода и данных всех задач. Таким образом, при переключении задач нам вообще не надо переключать никакие таблицы, таблица дескрипторов сегментов одна на всю систему. При этом возникает вопрос: если все задачи используют одну таблицу, как может получиться, что разные задачи имеют разные права на один и тот же сегмент?

Решение этого вопроса состоит в том, что права доступа кодируются не дескриптором, а селектором сегмента. Таким образом, разные права доступа на один сегмент — это, строго говоря, разные адреса. Понятно, что такое разделение работает лишь постольку, поскольку пользовательская программа не может формировать произвольные селекторы. В компьютерах Burroughs это достигалось теговой архитектурой: каждое слово памяти снабжалось дополнительными битами, *тегом* (tag), который определял тип данных, хранимых в этом слове, и допустимые над ним операции. Битовые операции над указателями не допускались. Благодаря этому задача не могла сформировать не только произвольные права доступа, но и вообще произвольный указатель.

Аналогичным образом реализована защита памяти в AS/400 [redbooks.ibm.com/sg242222]. Пользовательские программы имеют общее адресное пространство. Первоначально это общее пространство имен, в процессе же преобразования имени в адрес бинарное представление селектора сегмента снабжается и битами доступа, которые затем обрабатываются точно так же, как в машинах Burroughs, и точно так же недоступны пользовательскому коду для модификации.

Реализации языка С для этой архитектуры допускают использование указателей только в пределах одного сегмента кода или данных, но не позволяют формировать средствами С произвольные селекторы сегментов, остальные же языки, используемые на этой платформе (Cobol, RPG, языки хранимых процедур СУБД) вообще не имеют указателя как понятия.

Еще дальше в том же направлении продвинулись разработчики фирмы Intel, создавая экспериментальный микропроцессор iAPX432, описанный в следующем разделе.

5.2. Взаимно недоверяющие подсистемы

— Вы куда?
— У меня там портфель!
— Я вам его принесу!
— Я вам не доверяю. У меня там ценный веник.
(«Ирония судьбы, или С легким паром!»)
Г. Горин

С точки зрения безопасности, основной проблемой систем с кольцами защиты является неспособность таких систем защитить себя от ошибок в модулях, исполняющихся в высшем кольце защиты. В свете этого очень привлекательной концепцией представляется идея *взаимно недоверяющих подсистем*.

Согласно этой концепции, пользовательская задача не должна предоставлять системе доступа ко всем своим данным. Вместо этого задача должна выдавать *мандат* на доступ к буферу или нескольким буферам, предназначенным

для обмена данными. Все акты обмена данными как между пользовательской задачей и системой, так и между двумя пользовательскими задачами или двумя модулями системы, также осуществляются при помощи передачи мандатов.

Например, при исполнении системного вызова `int read(int file, void *buf, size_t size)` программа должна передать системе мандат на право записи в буфер `buf` размером `size` байт (рис. 5.11). При этом буфер будет отображен в адресное пространство подсистемы ввода/вывода, но эта подсистема не получит права записи в остальное адресное пространство нашей задачи. Впрочем, данный подход имеет две очевидные проблемы:

- при использовании страницочных и двухслойных сегментно-страницочных диспетчеров памяти мы можем отображать в чужие адресные пространства только объекты, выровненные на границу страницы и имеющие размер, кратный размеру страницы;
- мы не можем избавиться от супервизора. В данном случае это должен быть модуль, ответственный за формирование мандата и размещение отображеного модуля в адресном пространстве задачи, получающей мандат.

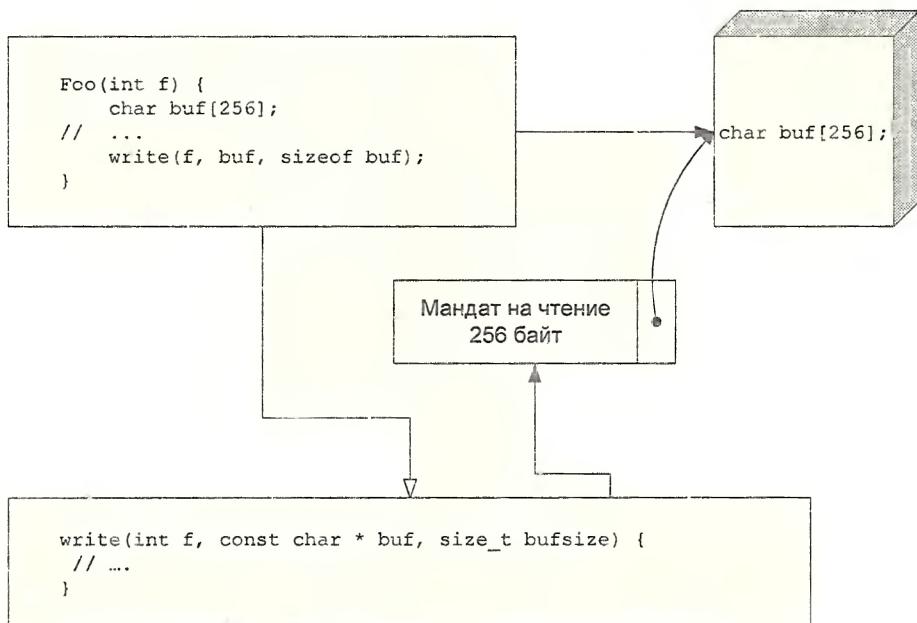


Рис. 5.11. Передача мандатов

Первая проблема является чисто технической — она приводит лишь к тому, что мы можем построить такую систему только на основе сегментного УУП,

в котором размер сегмента задается с точностью до байта или хотя бы до слова. Сегментированная память страдает от внешней фрагментации, но иногда такую цену стоит заплатить за повышение надежности.

Функции модуля, управляющего выдачей мандатов, оказываются слишком сложны для аппаратной и даже микропрограммной реализации. Этот модуль должен выполнять следующие функции:

- убедиться в том, что передаваемый объект целиком входит в один сегмент исходного адресного пространства;
- если объект состоит из нескольких сегментов, разумным образом обработать такую ситуацию. Для программной реализации может оказаться желательным умение объединить все элементы в один сегмент. Для аппаратной или микропрограммной реализации достаточно хотя бы уметь генерировать соответствующее исключение;
- сформировать содержимое дескриптора сегмента для объекта и записать в него соответствующие права. Эта операция требует заполнения 4—5 битовых полей, и запись ее алгоритма на псевдокоде займет около десятка операторов. По сложности алгоритма одна только эта операция сравнима с наиболее сложными командами современных коммерческих CISC-процессоров, таких как VAX или транспьютер;
- отметить в общесистемной базе данных, что на соответствующую область физической памяти существует две ссылки. Это нужно для того, чтобы процессы дефрагментации и управления виртуальной памятью правильно обрабатывали перемещения сегмента по памяти и его перенос на диск, отмечая изменения физического адреса или признака присутствия во всех дескрипторах, ссылающихся на данный сегмент. Далее эта проблема будет обсуждаться подробнее;
- найти свободную запись в таблице дескрипторов сегментов задачи-получателя. Эта операция аналогична строковым командам CISC-процессоров, которые считаются сложными командами;
- разумным образом обработать ситуацию отсутствия такой записи;
- записать сформированный дескриптор в таблицу.

Но еще больше проблем создает операция уничтожения мандата. Легко показать необходимость уничтожения мандата той же задачей, которая его создала.

Например, представим себе, что пользовательская программа передала какому-либо модулю мандат на запись в динамически выделенный буфер. Допустим также, что этот модуль вместо уничтожения мандата после передачи данных сохранил его — ведь наша система предполагает, что ни одному из модулей нельзя полностью доверять! Если мы вынуждены принимать в расчет такую

возможность, то не можем ни повторно использовать такой буфер для других целей, ни даже возвратить системе память, им занятую, потому что не пользующийся нашим доверием модуль по-прежнему может сохранять право записи в него.

В свете этого возможность для задачи, выдавшей мандат, в одностороннем порядке прекращать его действие представляется жизненно необходимой. Отказ от этой возможности или ее ограничение приводит к тому, что задача, выдающая мандат, вынуждена доверять тем задачам, которым мандат был передан, т. е. можно не городить огород и вернуться к обычной двухуровневой или многоуровневой системе колец защиты.

Для прекращения действия мандата мы должны отыскать в нашей общесистемной базе данных все ссылки на интересующий нас объект и объявить их недействительными. При этом мы должны принять во внимание возможности многократной передачи мандата и повторной выдачи мандатов на отдельные части объекта. Аналогичную задачу нужно решать при перемещении объектов в памяти во время дефрагментации, сбросе на диск и поиске жертвы (см. разд. 5.5.1) для такого сброса.

Структура этой базы данных представляется нам совершенно непонятной, потому что она должна отвечать следующим требованиям:

- обеспечивать быстрый поиск всех селекторов сегментов, ссылающихся на заданный байт или слово памяти;
- обеспечивать быстрое и простое добавление новых селекторов, ссылающихся на произвольные части существующих сегментов;
- занимать не больше места, чем память, отведенная под сегменты.

Последнее требование непосредственно не следует из концепции взаимно недоверяющих подсистем, но диктуется простым здравым смыслом. Если мы и смиримся с двух- или более кратным увеличением потребностей в памяти, оправдывая его повышением надежности, нельзя забывать, что даже простой просмотр многомегабайтовых таблиц не может быть быстрым.

В системах, использующих страницочные диспетчеры памяти, решить эти проблемы намного проще, потому что минимальным квантом разделения памяти является страница размером несколько килобайтов. Благодаря этому мы смело можем связать с каждым разделяемым квантом несколько десятков байтов данных, обеспечивающих выполнение первых двух требований. Если же минимальным квантом является байт или слово памяти, наши структуры данных окажутся во много раз больше распределяемой памяти.

Я оставляю читателю возможность попробовать самостоятельно разработать соответствующую структуру данных. В качестве более сложного упражнения можно рекомендовать записать алгоритм работы с такой базой данных на

псевдокоде, учитывая, что мы хотели бы иметь возможность реализовать этот алгоритм аппаратно или, по крайней мере, микропрограммно.

Дополнительным стимулом для читателя, решившего взяться за эту задачу, может служить тот факт, что разработчики фирмы Intel не смогли найти удовлетворительного решения.

Архитектура i432

Автору известен только один процессор, пригодный для полноценной реализации взаимно недоверяющих подсистем: iAPX432 фирмы Intel. Вместо создания системной базы данных о множественных ссылках на объекты, специалисты фирмы Intel усложнили диспетчер памяти и соответственно алгоритм разрешения виртуальных адресов.

Виртуальный адрес в i432 состоит из селектора объекта и смещения в этом объекте. Селектор объекта ссылается на таблицу доступа текущего домена. Домен представляет собой программный модуль вместе со всеми его статическими и динамическими данными (рис. 5.12). По идее разработчиков, домен соответствует замкнутому модулю языков высокого уровня, например пакету (package) языка Ada.

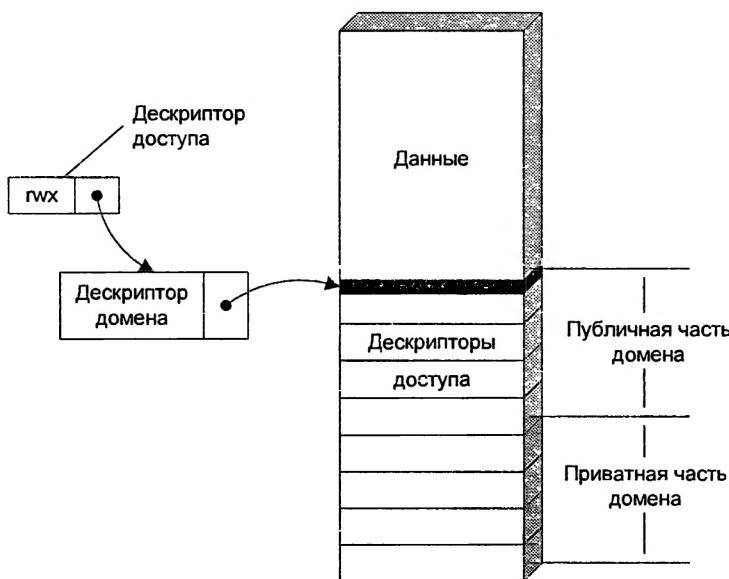


Рис. 5.12. Домен i432

Элемент таблицы доступа состоит из прав доступа к объекту и указателя на таблицу объектов процесса. Элемент таблицы объектов может содержать непосредственную ссылку на область данных объекта — такой объект аналогичен сегменту в обычных диспетчерах памяти, однако обращение к нему происходит через два уровня косвенности.

Иными словами, вместо

```
segment_table[seg].phys_address + offset
```

мы имеем

```
object_table[access_table[selector]].phys_address + offset
```

Дополнительный уровень косвенности позволяет нам упростить отслеживание множественных ссылок на объект: вместо использования структуры данных, которую мы так и не сумели изобрести, мы можем следить только за дескриптором исходного объекта, поскольку все ссылки на объект через дескрипторы доступа вынуждены в итоге проходить через дескриптор объекта.

Более сложными сущностями являются *уточнения* — объекты, ссылающиеся на отдельные элементы других объектов (рис. 5.13). Вместо физического адреса и длины дескриптор такого объекта содержит селектор целевого сегмента, смещение уточнения в целевом объекте и его длину. При ссылке на уточнение диспетчер памяти сначала проверяет допустимость смещения, а затем повторяет полную процедуру разрешения адреса и проверки прав доступа для целевого объекта.

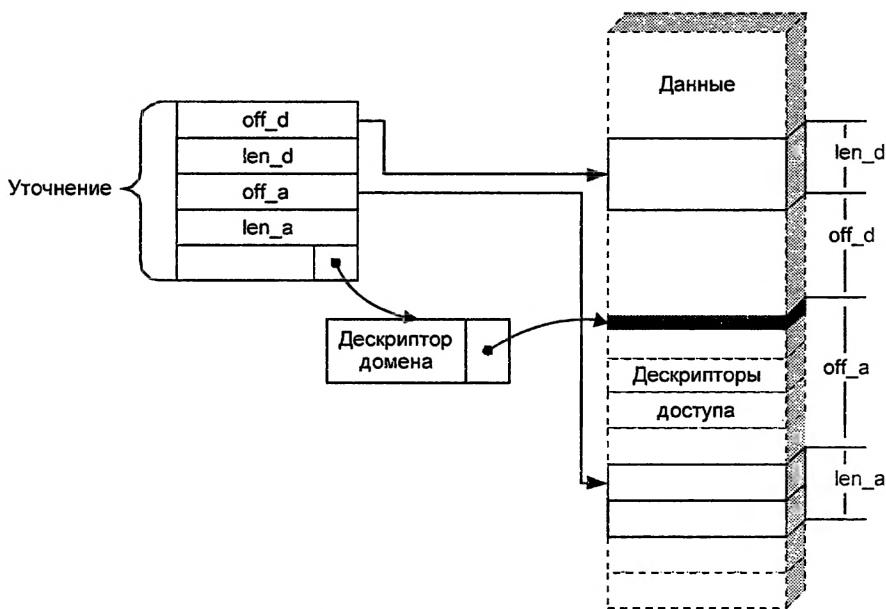


Рис. 5.13. Уточнение

Целевой объект также может оказаться уточнением, и диспетчер памяти будет повторять процедуру до тех пор, пока не дойдет до объекта, ссылающегося на физическую память. Более подробно этот механизм описан в работе [Органик 1987].

Уточнения позволяют решить проблему отзыва мандатов — мандат реализуется как уточнение исходного объекта (например, сегмента данных модуля, в ко-

тором выделен буфер), а программе-получателю передается даже не само уточнение, а лишь дескриптор доступа к нему. Для прекращения действия ман-дата достаточно удалить дескриптор уточнения. После этого все переданные другим модулям дескрипторы доступа будут указывать в пустоту, т. е. станут недействительными. Задачи же перемещения в памяти и сброса объектов на диск решаются путем изменения физического адреса или признака присутствия целевого объекта; для этого не требуется прослеживать цепочку уточнений.

Однако накладные расходы, связанные с реализацией этого сложного механизма, оказались очень большими. Процессор i432 остался экспериментальным и не получил практического применения. Трудно сказать, какой из факторов оказался важнее: катастрофически низкая производительность системы (автору не удалось найти официальных заявлений фирмы Intel по этому поводу, но легенды гласят, что экспериментальные образцы i432 с тактовой частотой 20 МГц исполняли около 20 000 операций в секунду) или просто неспособность фирмы Intel довести чрезмерно сложный кристалл до массового промышленного производства.

По непроверенным сведениям, проект i960 начинался как попытка реанимировать архитектуру i432, используя вместо экстравагантного CISC-процессора RISC-ядро, но сохранив архитектуру диспетчера памяти. Трудно проверить по открытым источникам, так ли это на самом деле, потому что выпускавшийся небольшой серией процессор i960MP, рассчитанный на работу с внешним УУП, не имел документированного интерфейса: все контакты микросхемы процессора, предположительно предназначенные для соединения с УУП, были описаны в документации как Not Connected (не присоединены). Впоследствии i960 был лишен всех остатков интерфейса с УУП и имел большой успех во встраиваемых приложениях и периферийных контроллерах.

Так или иначе, едва ли не единственной практической пользой, извлеченной из этого амбициозного проекта, оказалось тестирование системы автоматизированного проектирования, использованной впоследствии при разработке процессора 80386 и сопутствующих микросхем.

После завершения проекта i432 других попыток полностью реализовать взаимно недоверяющие подсистемы не делалось — изготовители коммерческих систем, по-видимому, сочли эту идею непродуктивной, а исследовательским организациям проекты такого масштаба просто не по плечу.

5.3. Сегменты, страницы и системные вызовы (продолжение)

Аппаратные схемы тонкого разделения доступа к адресному пространству не имели большого успеха не только из-за высоких накладных расходов, но и из-за того, что решали не совсем ту проблему, которая реально важна. Для повышения надежности системы в целом важно не обнаружение ошибок и даже не их локализация с точностью до модуля сама по себе, а возможность восстановления после их возникновения.

Самые распространенные фатальные ошибки в программах — это ошибки работы с указателями и выход индекса за границы массива (в наш сетевой век

ошибки второго типа более известны как "срыв буфера"). Эти ошибки не только часто встречаются, но и очень опасны, потому что восстановление после них практически невозможно.

Ошибки работы с указателями еще можно попытаться устраниТЬ, искоренив само понятие указателя. Примерно этой логикой продиктован запрет на формирование указателей в машинах Burroughs, именно из этих соображений в Java и некоторых других интерпретируемых языках указателей вообще нет.

Однако искоренить понятие индексируемого массива уже не так легко. А ошибки индексации присущи всем компилируемым языкам, начиная с Fortran и Algol 60. Вставка проверок на границы индекса перед каждой выборкой элемента массива создает накладные расходы, но тоже не решает проблемы: ошибка все равно обнаруживается не в момент совершения (в тот момент, когда мы вычислили неверный индекс), а позже, когда мы попытались его использовать. В момент индексации обычно уже невозможно понять, какой же элемент массива имелся в виду. Программе остается только нарисовать на экране какое-нибудь прощальное сообщение вроде "Unhandled Java Exception" и мирно завершить свой земной путь.

Понятно, что реакция пользователя на подобное сообщение будет ничуть не более адекватной, чем на хрестоматийное "Ваша программа выполнила недопустимую операцию — General Protection Fault" (впрочем, кто знает, может быть, такая реакция и является самой адекватной?).

Прогресс в решении этой проблемы лежит уже в сфере совершенствования технологий программирования, и вряд ли может быть обеспечен усложнением диспетчеров памяти. Уровень же надежности, обеспечиваемый современными ОС общего назначения с разделением памяти, по-видимому, оптимален в том смысле, что улучшения в сфере защиты памяти могут быть достигнуты лишь ценой значительных накладных расходов без принципиального повышения наработки на отказ.

5.4. Разделяемые библиотеки

Ранее мы упоминали разделяемые библиотеки как одно из преимуществ страницных и сегментных диспетчеров памяти перед базовыми и банковыми. При базовой адресации образ каждого процесса должен занимать непрерывные области как в физическом, так и в логическом адресном пространстве. В этих условиях реализовать разделяемую библиотеку невозможно. Но и при использовании страницной адресации не все так просто.

Использование разделяемых библиотек и/или DLL (в данном случае разница между ними не принципиальна) предполагает ту или иную форму сборки в момент загрузки: исполняемый модуль имеет неразрешенные адресные ссыл-

ки и имена библиотек, которые ему нужны. При загрузке эти библиотеки подгружаются и ссылки разрешаются. Проблема здесь в том, что при подгрузке библиотеки ее нужно переместить, перенастроив абсолютные адресные ссылки в ее коде и данных (см. главу 3). Если в разных процессах библиотека будет настроена на разные адреса, она уже не будет разделяемой (рис. 5.14)! Если разделяемые библиотеки могут иметь неразрешенные ссылки на другие библиотеки, проблема только усугубляется — к перемещаемым ссылкам добавляются еще и внешние.

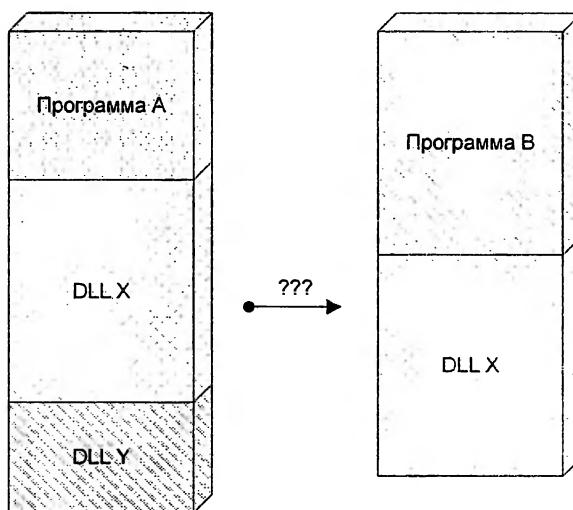


Рис. 5.14. Конфликтующие адреса отображения DLL

В старых системах семейства Unix, использовавших абсолютные загружаемые модули формата a.out, разделяемые библиотеки также поставлялись в формате абсолютных модулей, настроенных на фиксированные адреса. Каждая библиотека была настроена на свой адрес. Поставщик новых библиотек должен был согласовать этот адрес с разработчиками системы. Это было весьма непрактично, поэтому разделяемых библиотек было очень мало (особенно если не считать те, которые входили в поставку ОС).

Более приемлемое решение этой проблемы реализовано в OS/2 2.x и Windows 95 (обе эти архитектуры являются развитием систем с единым адресным пространством). Идея состоит в том, чтобы выделить область адресов под загрузку DLL и отображать эту область в адресные пространства всех процессов. Таким образом, все DLL, загруженные в системе, видны всем (рис. 5.15). Аналогичное решение использовалось в ряде других систем — в VAX/VMS, Unix-системах с загружаемыми модулями формата COFF (Xenix, SCO Unix) [Робачевский 1999] и др.

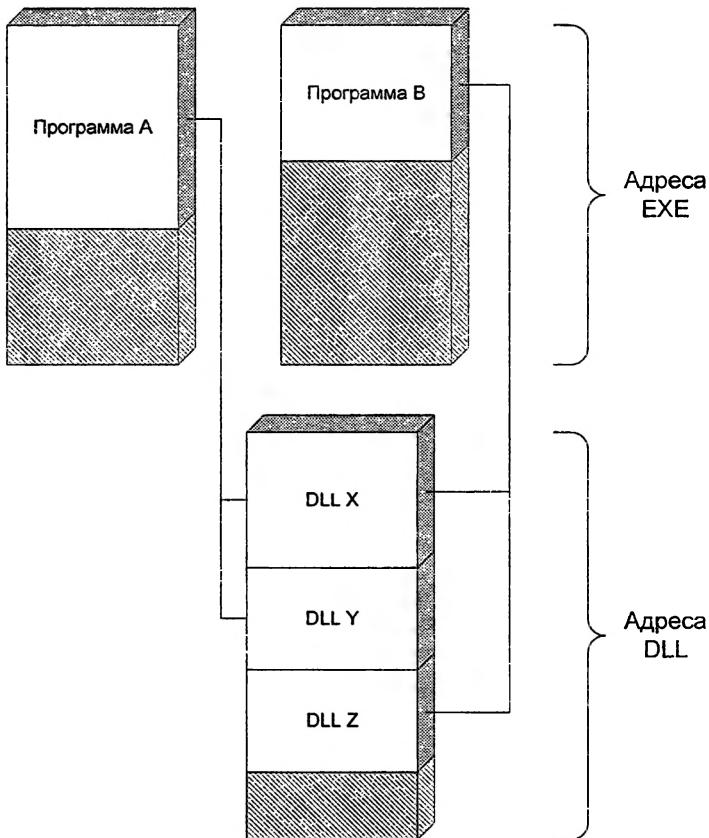


Рис. 5.15. Загрузка DLL в OS/2 и Win32

Загрузка DLL в OS/2 и Windows 95

В IBM OS/2 2.x процессу выделяется 512 Мбайт адресного пространства (это странное ограничение обусловлено требованиями совместимости с 16-разрядным кодом). EXE-модули и приватные DLL (бывают и такие) загружаются с младших адресов с увеличением адреса загрузки. Приватная область, как следует из ее названия, своя у каждого процесса. Разделяемые DLL загружаются, начиная с верхней границы этой области с уменьшением адресов. Соответствующая область памяти отображается в адресные пространства всех процессов. Заранее установленной границы между приватным и разделяемым пространствами памяти нет, разделяемая область растет по мере того, как увеличивается количество загруженных библиотек.

В Windows 95 все 32-битные DLL загружаются в третий гигабайт 32-битного адресного пространства. Четвертый гигабайт используется для образа ядра ОС. По сообщениям ряда источников (я должен признать, что не проверял это самостоятельно), Windows 95 в принудительном порядке разделяет сегменты кода и данных всех DLL, даже тех, в заголовке которых не указан флаг Shared.

Поэтому каждый процесс имеет доступ к сегментам кода и данных всех DLL в системе, в том числе и к тем сегментам, доступа к которым он не просил.

Очевидным недостатком такого решения (как, впрочем, и предыдущего) является неэффективное использование адресного пространства: при сколь-нибудь сложной смеси загруженных программ большей части процессов большинство библиотек будут просто не нужны. В конце семидесятых, когда разрабатывалась VAX/VMS, это еще не казалось серьезной трудностью, но сейчас, когда многим приложениям становится тесно в 4 Гбайт и серверы с таким объемом оперативной памяти уже не редкость, это действительно может стать проблемой.

Менее очевидный, но более серьезный недостаток состоит в том, что эта архитектура не позволяет двум приложениям одновременно использовать две разные, но одноименные DLL — например, две разные версии стандартной библиотеки языка С. Поэтому либо мы вынуждены требовать от всех разделяемых библиотек абсолютной (*bug-for-bug*) совместимости версий, либо честно признать, что далеко не каждая смесь прикладных программ будет работоспособна. Первый вариант нереалистичен, второй же создает значительные неудобства при эксплуатации, особенно если система интерактивная и многопользовательская.

Загрузка одноименных DLL в Windows 2000 и OS/2

В современных реализациях Win32 и OS/2 допускается загрузка одноименных DLL; при этом, для определения того, какая DLL будет использоваться, загрузчик руководствуется полным путевым именем этой библиотеки. В Windows 2000 это поведение загрузчика включено по умолчанию, в OS/2 оно включается для отдельных процессов недокументированной переменной среды LIBPATHSTRICT.

Это позволяет запускать в одной системе несколько разных версий одной и той же программы и вообще несколько повышает возможности сборки работоспособной смеси приложений, но опять-таки приводит к загромождению адресных пространств всех процессов в системе ненужным большинству процессов комом.

Разделяемые библиотеки Windows NT

В Windows NT используется технология разделения библиотек, которая парадоксальным образом сочетает недостатки, характерные как для архаичных Unix-систем, так и для OS/2 и Windows 95, но имеет также и ряд важных собственных недостатков. При этом я затрудняюсь назвать хотя бы одно преимущество, характерное для этой технологии.

Windows NT, как и все остальные ОС семейства Win32, использует загрузочные модули формата PE (Portable Executable). Формат PE представляет собой нестандартное расширение формата COFF (Common Object File Format), используемого в ряде систем семейства Unix (Microsoft/SCO Xenix, SCO Unix, IBM AIX и др.). Самое главное из расширений состоит в том, что оригинальный COFF не допускает исполняемых модулей и разделяемых библиотек, имеющих таб-

лицу перемещений, относительные сегменты могут содержаться лишь в объектных модулях. Формат PE это допускает, хотя форматы таблиц перемещений в объектных и загружаемых модулях различны, а сами таблицы даже имеют разные названия: в загружаемом модуле они называются fixup table.

Каждая DLL обязана иметь определенный загрузочный адрес и поставляется настроенной на этот адрес. Загрузочный адрес задается при сборке DLL; MS Visual C по умолчанию настраивает DLL на адрес 0x10000000, т. е. на начало второго гигабайта адресного пространства. Разумеется, если вы собираете проект, включающий несколько DLL, то хотя бы у одной из этих DLL необходимо изменить этот адрес.

В поставке MS Visual C и в Resource Kit (набор неподдерживаемых утилит, включаемых в дистрибутив ОС) всех версий Windows NT содержится утилита REBASE, которая позволяет перенастроить DLL на другой загрузочный адрес без полной ее пересборки.

В районе верхней границы второго гигабайта адресного пространства размещается относительно небольшая область, в которую загружаются системные DLL — KERNEL32.DLL USER32.DLL и т. д.

В отличие от Unix-систем, в Win32 не существует никакого центрального реестра, в котором можно было бы регистрировать базовые адреса для несистемных разделяемых DLL. Поэтому конфликты между адресами DLL различных поставщиков — распространенное и даже штатное явление. Нередко конфликтуют даже адреса DLL, поставляемых одним разработчиком в составе разных линеек продуктов. Какого-то согласования удается добиться только в пределах одного программного продукта. Так, по сообщению [blogs.msdn.com/larrygosterman 174516], при сборке каждой новой версии Microsoft Exchange запускается специальный скрипт, который находит подходящий базовый адрес для Exchange.DLL путем перебора всех допустимых диапазонов адресов.

Если DLL отображается в память по указанному загрузочному адресу, то она может исполняться без какой-либо дополнительной перенастройки. При этом "загрузка" DLL осуществляется простым отображением ее сегментов кода и данных в виртуальную память процесса. Подкачка сегментов из файла происходит при первом обращении к ним; поскольку сегменты кода не модифицируются, то их никогда не сбрасывают обратно на диск. Если страница из такого сегмента будет объявлена жертвой при страницочной подкачке, а потом программа вновь востребует ее, то эту страницу снова подкачивают из файла. Поэтому Windows NT блокирует файлы EXE и DLL во время их исполнения и не допускает их удаления, переименования и какой-либо модификации.

Если отобразить DLL по ее загрузочному адресу невозможно, в действие вступает механизм перемещения. Система находит свободное место и загружает DLL туда. При этом создается приватная копия сегмента кода. Загрузчик не может просто отобразить сегмент кода DLL в память и подкачивать его при первом обращении; он должен полностью считывать его в память и перенастроить. Соответствующие страницы не разделяются между процессами ни при каких обстоятельствах, т. е. даже если в двух процессах можно будет отобразить DLL на одни и те же адреса, система не попытается это сделать и честно создаст две копии сегмента кода. Подсистема страницной подкачки считает соответствующие страницы модифицированными, сохраняет их в swap-файле и подкачивает их оттуда, а не из файла DLL.

Таким образом, если пользователь установит и попытается исполнить смесь приложений, использующих разделяемые DLL с конфликтующими адресами, то эта смесь приложений запустится без каких-либо видимых возражений. Пользователь может заподозрить конфликт адресов только по резкому увеличению времени запуска приложений и по необъяснимому росту потребностей в памяти и swap-пространстве. Средства, позволяющие анализировать распределение адресов между DLL, существуют в природе, но не входят в основную поставку ОС; к тому же использование таких средств — сложная и кропотливая работа, доступная далеко не каждому квалифицированному системному администратору.

Поэтому (а также из-за проблем обеспечения совместимости разных версий библиотек) Microsoft настоятельно не рекомендует злоупотреблять разделяемыми DLL. Однако многие серверные приложения и так называемые "интегрированные офисные пакеты" (например, MS Office) состоят из довольно большого количества процессов, разделяющих значительные объемы кода — подобные приложения, разумеется, не могут обойтись без разделяемых DLL.

Поставщик компонентного приложения обычно все же пытается распределить адреса между DLL своего пакета, но установка нескольких серверных приложений может привести к конфликтам. В значительной мере это компенсируется тем фактом, что Windows NT, в отличие от Windows 95, не отображает разделяемые DLL в адресные пространства тех процессов, которые этого не просили. Несмотря на это, во многих документах и даже в учебных курсах Microsoft настоятельно рекомендует руководствоваться принципом "одно приложение или сервис — одна машина". В том числе не рекомендуется совмещать на одной машине сервисы, предоставляемые приложениями Microsoft, например Microsoft SQL Server и контроллер домена или SQL Server и MS Exchange.

По ряду неофициальных и полуофициальных сообщений, в Windows Vista сделана попытка частично решить эту проблему. Загрузчик DLL этой системы содержит список так называемых "известных DLL" (well-known DLL) — разделяемых DLL, относящихся к популярным прикладным пакетам. Для каждой из таких DLL выделен собственный загрузочный адрес, обычно не совпадающий с ее оригинальным загрузочным адресом. Обнаружив попытку загрузки одной из таких DLL, Vista перемещает ее на указанный адрес и затем организует разделение перемещенного образа.

Лишнее всех перечисленных недостатков решение предлагают современные системы семейства Unix, использующие загружаемые модули формата ELF. Впрочем, для реализации этого решения пришлось, ни много, ни мало, переделать компилятор и научить его генерировать позиционно-независимый код (см. разд. 3.5).

Разделяемые библиотеки формата ELF

Исполняемые модули формата ELF бывают двух типов: статические — полностью самодостаточные, не использующие разделяемых объектов, и динамические — содержащие ссылки на разделяемые объекты и неразрешенные символы. И статические, и динамические модули являются абсолютными. При создании образа процесса система начинает с того, что отображает статически собранный исполняемый объект в адресное пространство. Статический модуль

не нуждается ни в какой дополнительной настройке и может начать исполнение сразу после этого.

Для динамического же загрузочного модуля система загружает так называемый интерпретатор, или редактор связей времени исполнения (run-time linker), по умолчанию `ld.so.1`. Он исполняется в контексте процесса и осуществляет подгрузку разделяемых объектов и связывание их с кодом основного модуля и друг с другом.

При подгрузке разделяемый объект также отображается в адресное пространство формируемого процесса. Отображается он не на какой-либо фиксированный адрес, а как получится, с одним лишь ограничением: сегменты объекта будут выровнены на границу страницы. Не гарантируется даже, что адреса сегментов будут одинаковы при последовательных запусках одной и той же программы.

Документ [HOWTO Library] без обиняков утверждает, что в разделяемых объектах можно использовать только код, компилированный с ключом `-f PIC`. Документ [docs.sun.com 816-0559-10] менее категоричен:

"Если разделяемый объект строится из кода, который не является позиционно-независимым, текстовый сегмент, скорее всего, потребует большое количество перемещений во время исполнения. Хотя редактор связей и способен их обработать, возникающие вследствие этого накладные расходы могут вести к серьезному снижению производительности"¹.

Как уже говорилось в разд. 3.5, используемый в разделяемых объектах код не является истинно позиционно-независимым: он содержит перемещаемые и даже настраиваемые адресные ссылки, такие как статически инициализированные указатели и ссылки на процедуры других модулей. Но все эти ссылки размещены в сегменте данных. Используемые непосредственно в коде ссылки собраны в две таблицы, *GOT* (Global Offset Table, Глобальная таблица смещений) и *PLT* (Procedure Linkage Table, Таблица процедурного связывания) (рис. 5.16). Каждый разделяемый модуль имеет свои собственные таблицы. Порожденный компилятором код определяет адреса этих таблиц, зная их смещение в разделяемом объекте относительно точки входа функции (см. примеры 3.7 и 5.1).

Пример 5.1. Типичный пролог функции, предназначенный для использования в разделяемом объекте

```
.text
.align 2,0x90
.globl _strerror
_strerror:
    pushl %ebp ; Стандартный пролог функции
    movl %esp,%ebp
    pushl %ebx
    call L4
```

¹ Перевод автора.

L4:

```
popl %ebx ; Загрузка текущего адреса в регистр EBX
addl $_GLOBAL_OFFSET_TABLE_+[.-L4],%ebx
```

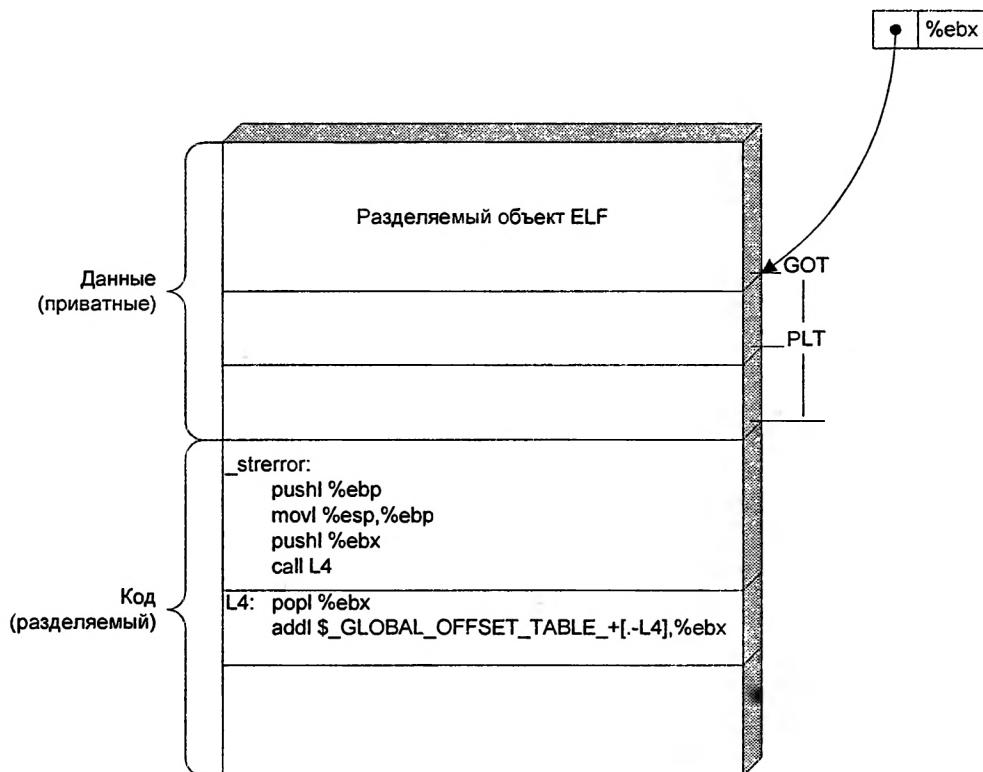


Рис. 5.16. Global Offset Table (Глобальная таблица смещений) и Procedure Linkage Table (Таблица процедурного связывания)

Сегмент кода отображается с разделением его между всеми процессами, использующими объект (конечно, при условии, что он был компилирован с правильными ключами и не содержит перемещаемых адресов).

Напротив, сегмент данных и таблицы GOT и PLT создаются в каждом образе заново и, если это необходимо, адресные ссылки в них подвергаются перемещению. По мере разрешения внешних ссылок, интерпретатор заполняет PLT объекта ссылками на символы, определенные в других объектах (подобный стиль работы с внешними ссылками широко распространен в байт-кодах интерпретируемых языков — см. разд. 3.9).

Сегмент данных разделяемого объекта, таким образом, соответствует тому, что в OS/2 и Win32 называется приватным сегментом данных DLL: каждая задача, использующая объект, имеет свою копию этого сегмента. Аналога глобальному сегменту данных разделяемые библиотеки ELF не имеют — если это необхо-

димо, код библиотеки может создать собственный сегмент разделяемой памяти, но в нем невозможно иметь статически инициализированные данные, и для него никто не гарантирует отображения на одни и те же адреса разных процессов, поэтому в нем невозможно хранить указатели.

По умолчанию, интерпретатор осуществляет отложенное редактирование связей: если сегмент данных он полностью настраивает до передачи управления пользовательскому коду, то записи в PLT изначально указывают на специальную процедуру редактора связей. Будучи вызвана, эта процедура по стеку вызова или другими средствами определяет, какую же процедуру пытались вызвать на самом деле, и настраивает ее запись в PLT (рис. 5.17). В случае, когда большинство программ не вызывает большую часть функций, как это часто и бывает при использовании разделяемых библиотек, это дает определенный выигрыш в производительности.

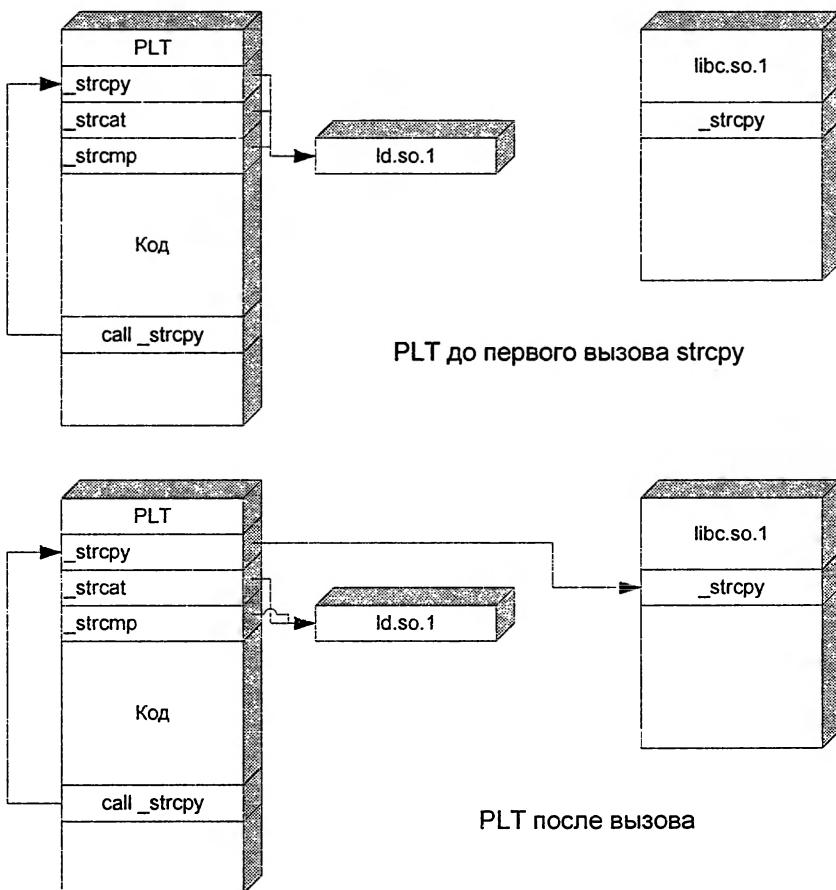


Рис. 5.17. Редактор связей времени исполнения

Пример 5.2. Структура PLT для процессора SPARC
(цитируется по [docs.sun.com 816-0559-10])

Первые две (специальные) записи PLT до загрузки программы:

.PLT0:

```
unimp  
unimp  
unimp
```

.PLT1:

```
unimp  
unimp  
unimp
```

...

Обычные записи PLT до загрузки программы:

.PLT101:

```
sethi (.-.PLT0),%g1  
ba,a .PLT0  
nop
```

.PLT102:

```
sethi (.-.PLT0),%g1  
ba,a .PLT0  
nop
```

...

Специальные записи PLT после загрузки программы:

.PLT0:

```
save %sp,-64,%sp  
call runtime-linker  
nop
```

.PLT1:

```
.word identification  
unimp  
unimp
```

...

Обычные записи PLT после настройки:

.PLT101:

```
sethi (.-.PLT0),%g1  
sethi %hi(name1),%g1  
jmp1 %g1+%lo(name1),%g0
```

.PLT102:

```
sethi (.-.PLT0),%g1  
sethi %hi(name2),%g1  
jmp1 %g1+%lo(name2),%g0
```

Таким образом, каждая пользовательская задача, загруженная в Unix, имеет собственное адресное пространство с собственной структурой (рис. 5.18). Некоторые участки памяти у разных задач могут совпадать, и это позволяет сэкономить ресурсы за счет их разделения. Это существенно менее глубокое разделение, чем то, что достигается в Windows 95 и OS/2, но, как мы видели в разд. 3.10, более глубокое и принудительное разделение кода чревато серьезными проблемами.

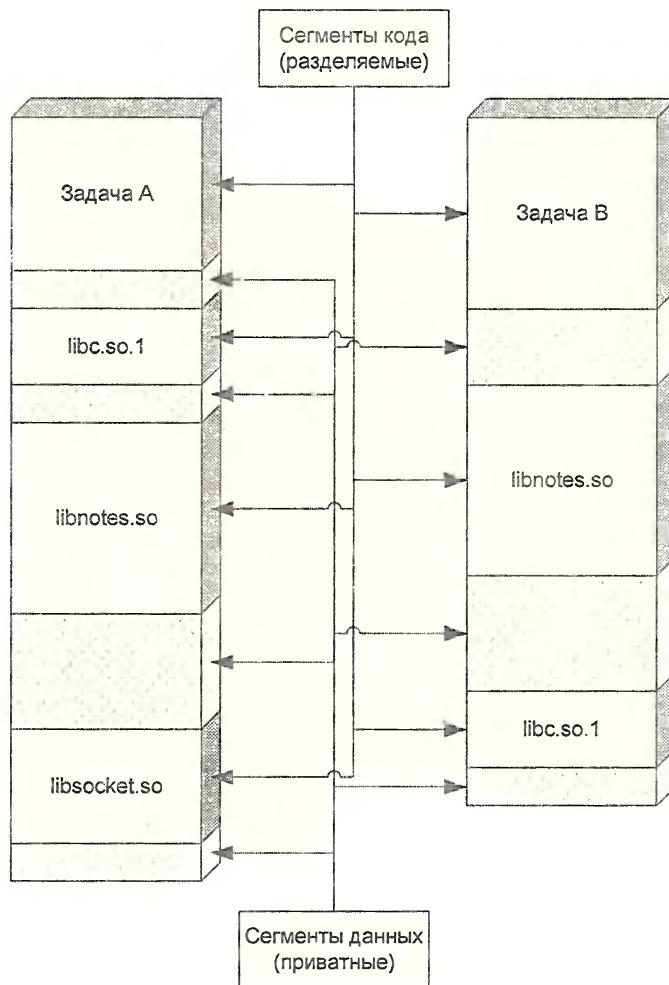


Рис. 5.18. Разделяемые библиотеки ELF

Разделяемые объекты ELF идентифицируются по имени файла. Исполняемый модуль может ссылаться на файл как по простому имени (например, `libc.so.1`), так и с указанием пути (`/usr/lib/libc.so.1`). При поиске файла по простому имени

редактор связей ищет его в каталогах, указанных в переменной среды `LD_LIBRARY_PATH`, в записи `RPATH` заголовка модуля и, наконец, в каталогах по умолчанию, перечисленных в конфигурационном файле `/var/ld/ld.config` (именно в таком порядке, [docs.sun.com 816-0559-10]). При формировании имен каталогов могут использоваться макроподстановки с использованием следующих переменных:

`$ISALIST` — список систем команд — полезно на процессорах, поддерживающих несколько систем команд, например `x86` и `8086`, `SPARC 32` и `SPARC 64`.

`$ORIGIN` — каталог, из которого загружен модуль. Полезно для загрузки приложений, которые имеют собственные разделяемые объекты.

`$OSNAME`, `$OSREL` — название и версия операционной системы.

`$PLATFORM` — тип процессора. Полезно для приложений, которые содержат в поставке бинарные модули сразу для нескольких процессоров, сетевых установок таких приложений, или сетевой загрузки в гетерогенной среде.

Понятно, что задание простого имени предпочтительнее, т. к. дает администратору системы значительную свободу в размещении разделяемых библиотек. Впрочем, системный редактор связей `ldd` позволяет изменять имена внешних ссылок и `RPATH` в уже построенном модуле, в частности заменяя одни файловые пути на другие, путевые имена на простые и наоборот. Благодаря этому, поставщик приложений для ОС, основанных на формате `ELF`, имеет гораздо меньше возможностей испортить жизнь системному администратору, чем поставщик приложений для `Windows`.

По стандартному соглашению, имя библиотеки обязательно содержит и номер версии (в обоих примерах это 1). В соответствии с требованиями фирмы Sun номер версии меняется, только когда интерфейс библиотеки меняется на несовместимый — убираются функции, изменяется их семантика и т. д. Из менее очевидных соображений [docs.sun.com 816-0559-10] требуется менять номер версии и при добавлении функции или переменной: ведь вновь добавленный символ может конфликтовать по имени с символом какой-то другой библиотеки.

Исправление ошибок, т. е. нарушение "bug-for-bug compatibility", основанием для изменения номера версии фирма Sun не считает. Напротив, в Linux принято снабжать разделяемые библиотеки минимум двумя, а иногда и более номенклатурой версий — старшая (*major*) версия изменяется по правилам, приблизительно соответствующим требованиям Sun, а младшие (*minor*) — после исправления отдельных ошибок и других мелких изменений.

Благодаря этому соглашению, в системе одновременно может быть установлено несколько версий одного и того же модуля, а пользовательские программы могут ссылаться именно на ту версию, с которой разрабатывались и на совместимость с которой тестировались. Администратор может управлять выбором именно той библиотеки, на которую ссылаются конкретные модули, либо изменяя ссылки в этих модулях с помощью `ldd`, либо используя символические связи.

Как правило, администратору не требуется самостоятельно разрешать конфликты версий с помощью `ldd`. Современные Unix-системы имеют развитые средства управления пакетами, позволяющие для каждого пакета указать, от каких других пакетов он зависит и какие версии этих других пакетов ему нужны

или хотя бы подходят. При установке приложения инсталляционный скрипт проверяет все зависимости и доустанавливает все необходимые библиотеки, сохраняя старые версии, если это необходимо.

В наиболее выигрышном положении, конечно же, оказываются пользователи свободно распространяемых ОС, таких как Linux и FreeBSD, которые могут в любой момент скачать нужную им версию любой разделяемой библиотеки с ближайшего зеркала дистрибутивного сервера.

5.5. Страницочный обмен

Подкачка, или *сплонг* (от англ. swapping — обмен), — это процесс выгрузки редко используемых областей виртуального адресного пространства программы на диск или другое устройство массовой памяти. Такая массовая память всегда намного дешевле оперативной, хотя и намного медленнее.

Во многих учебниках по ОС приводятся таблицы, аналогичные табл. 5.3.

Таблица 5.3. Сравнительные характеристики и стоимость различных типов памяти

Тип памяти	Время доступа	Цена 1 Мбайт (цены 1995 г.)	Способ использования
Статическая память	15 нс	\$200	Регистры, кэш-память
Динамическая память	70 нс	\$30 (4 Мбайт SIMM)	Основная память
Жесткие магнитные диски	1—10 мс	\$3 (1.2 Гбайт EIDE)	Файловые системы, устройства свопинга
Магнитные ленты	Секунды	\$0.025 (8 мм Exabyte)	Устройства резервного копирования

При разработке системы всегда есть желание сделать память как можно более быстрой. С другой стороны, потребности в памяти очень велики и постоянно растут.

Примечание

Существует эмпирическое наблюдение, что любой объем дисковой памяти будет полностью занят за две недели.

Очевидно, что система с десятками гигабайтов статического ОЗУ будет иметь стоимость, скажем так, совершенно не характерную для персональных систем, не говоря уже о габаритах, потребляемой мощности и прочем. К счастью, далеко не все, что хранится в памяти системы, используется одновременно. В каждый заданный момент исполняется только часть установленного в системе программного обеспечения, и работающие программы используют только часть данных.

Эмпирическое правило "80-20", часто наблюдаемое в коммерческих системах обработки транзакций, гласит, что 80% операций совершаются над 20% файла [Heising 1963]. В ряде работ, посвященных построению оптимизирующих компиляторов, ссылаются на правило "90-10" (90% времени исполняется 10% кода) — впрочем, есть серьезные основания сомневаться в том, что в данном случае соотношение именно таково [Кнут 2000, т. 3].

В действительности, удивительно большое количество функций распределения реальных дискретных величин (начиная от количества транзакций на строку таблицы и заканчивая распределением богатства людей или капитализации акционерных обществ) подчиняются закону Парето [Pareto 1897]: $p_k = ck^{\theta-1}$, где θ — число в диапазоне от 0 до 1, k — значение величины (в нашем случае — количество обращений к данной записи), p — количество записей, к которым происходит k обращений, c — нормализующий коэффициент (правило "80-20" соответствует $\theta = \log 80 / \log 20 = 0,1386$) или его частному случаю, распределению Зипфа [Zipf 1949]: $p = c/k$.

Детальное обсуждение этого явления, к сожалению, не доходящее до глубинных его причин, приводится в [Кнут 2000, т. 3].

Это различие приводит нас к идеи многослойной или *многоуровневой памяти*, когда в быстрой памяти хранятся часто используемые код или данные, а редко используемые постепенно мигрируют на более медленные устройства. Действительно, если исполняемый нашей системой код подчиняется правилу "10-90", то мы можем перенести 90% кода в медленную память — и это приведет к падению наблюдаемой производительности лишь на несколько процентов!

Впрочем, если программа этому правилу не подчиняется, то падение производительности будет гораздо более значительным. Если нарушение правила оказывается локальным (например, при завершении работы одного модуля программы и передаче управления другому), то падение производительности может оказаться небольшим, но все-таки заметным. Это явление даже имеет специальное название — большое количество промахов кэш-памяти, приводящее к полной смене содержимого кэша, называется *промыванием кэша* (cache washing). Аналогичное явление при большом количестве страничных отказов не имеет столь же общепринятого названия, в разных книгах встречаются такие описательные термины, как трэшиング (thrashing), пробуксовка и т. д. Понятно, что система, объем быстрой памяти которой недостаточен для исполняющейся на ней смеси задачий, будет постоянно находиться в состоянии трэшинга или близком к нему.

Иногда оказывается даже целесообразно специально переделывать программу, чтобы локализовать исполнение кода и обращения к данным так, чтобы они помещались в памяти.

Способы миграции данных и определения того, какие именно данные следует мигрировать, отличаются довольно-таки большим разнообразием.

В случае дисковой памяти такая миграция осуществляется вручную: администратор системы сбрасывает на ленты или, в современных условиях, CD-RW или DVD, редко используемые данные и программы и заполняет освободившееся место чем-то нужным. Для больших и сильно загруженных систем существуют специальные программы, которые определяют, что является малоценным, а что — нет, обычно используя статистику обращения к данным. Управление миграцией из ОЗУ на диск иногда осуществляется пользователем, но часто это оказывается слишком утомительно. В случае миграции между кэш-памятью и ОЗУ делать что-то вручную просто физически невозможно.

5.5.1. Поиск жертвы

...и вот мы обрадовались вашему приходу. — может, вы согласитесь принести себя в жертву.

А. Тумуола

Естественно, для того чтобы автоматизировать процесс удаления "барахла" — редко используемых данных и программ — мы должны иметь какой-то легко формализуемый критерий, по которому определяется, какие данные считаются редко используемыми.

Один критерий выбора очевиден — при прочих равных условиях, в первую очередь мы должны выбирать в качестве "жертвы" (*victim*) для удаления тот объект, который не был изменен за время жизни в быстрой памяти. Действительно, вы скорее удалите с винчестера саму игрушку (если у вас есть ее копия на дискетах), чем файлы сохранения!

Для ручного переноса данных очевиден и другой критерий: нужно удалять то, что дальше всего не будет использоваться в будущем.

Замечание

Для блоков данных, которые не подверглись изменению за время пребывания в быстрой памяти, удаление будет состоять в уничтожении копии. Для модифицированных же блоков новое значение данных должно быть скопировано обратно в медленную память, и только после этого можно произвести собственно удаление.

Конечно, любые предположения о будущем имеют условный характер, все может неожиданно измениться. Мы делаем предположения о том, что будет использоваться, только на основании накопленной статистики обращений к страницам. Такая экстраполяция не совсем корректна логически, поэтому

может оказаться целесообразным смириться с некоторой неточностью или неполнотой этой статистики.

Самая простая стратегия — выбрасывать случайно выбранный объект. При этом не надо собирать никакой статистики о частоте использования и т. д., важно лишь обеспечить равномерность распределения генерируемых псевдо-случайных чисел. Очевидно, что при этом удаленный объект совершенно не-обязательно будет ненужным...

Можно также удалять то, что дольше всего находится в памяти. Это называется алгоритмом *FIFO* (First In, First Out, первый вошел, первый вышел). Видно, что это уже чуть сложнее случайного удаления — нужно запоминать, когда мы что загружали.

Поиск жертвы в VAX/VMS и Windows NT/2000/XP

В VAX/VMS и Windows NT/2000/XP применяется любопытный вариант данного алгоритма. В этих системах страница, объявленная жертвой, исключается из адресного пространства задачи (у соответствующего дескриптора выставляется бит отсутствия), но не отдается немедленно под другие нужды. Вместо этого страницы, помеченные как отсутствующие, помещаются в пул свободных страниц, который, по совместительству, используется и для дискового кэша и может занимать более половины оперативной памяти. У VAX/VMS этот пул состоит из трех очередей (рис. 5.19):

1. Очередь модифицированных страниц, ждущих записи на диск (по мере записи эти страницы переходят во вторую очередь).
2. Очередь немодифицированных страниц.
3. Очередь свободных страниц (освобожденных прикладной программой или освободившихся после ее завершения).

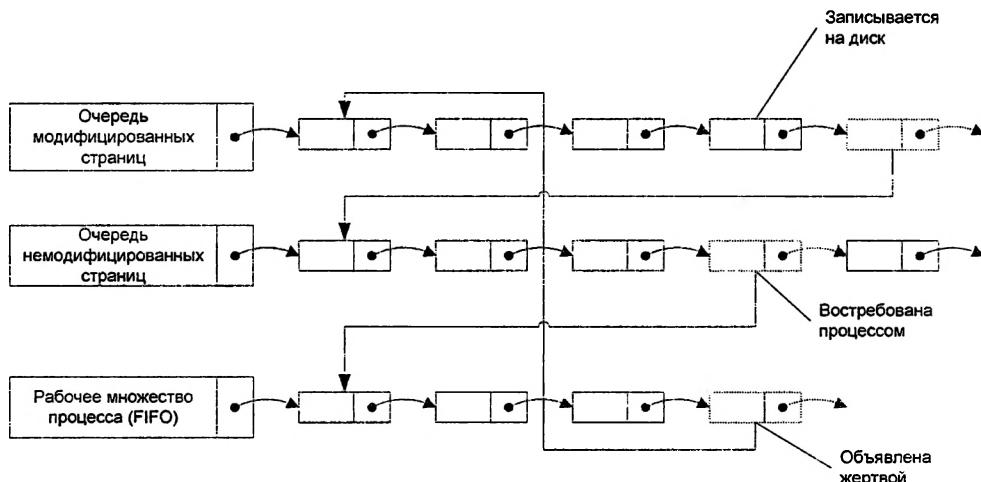


Рис. 5.19. Виртуальная память VAX/VMS

Жертвой с равной вероятностью может быть объявлена как модифицированная, так и немодифицированная страница, однако для запросов прикладных программ и буферов дискового кэша используются только страницы из второй и третьей очередей.

Обрабатывая страницочный отказ, система не обращается к диску за содержимым требуемой страницы, а сначала пытается найти ее в одной из очередей пулла. Если страница там, ее без дальнейших вопросов включают в адресное пространство задачи (рис. 5.20).

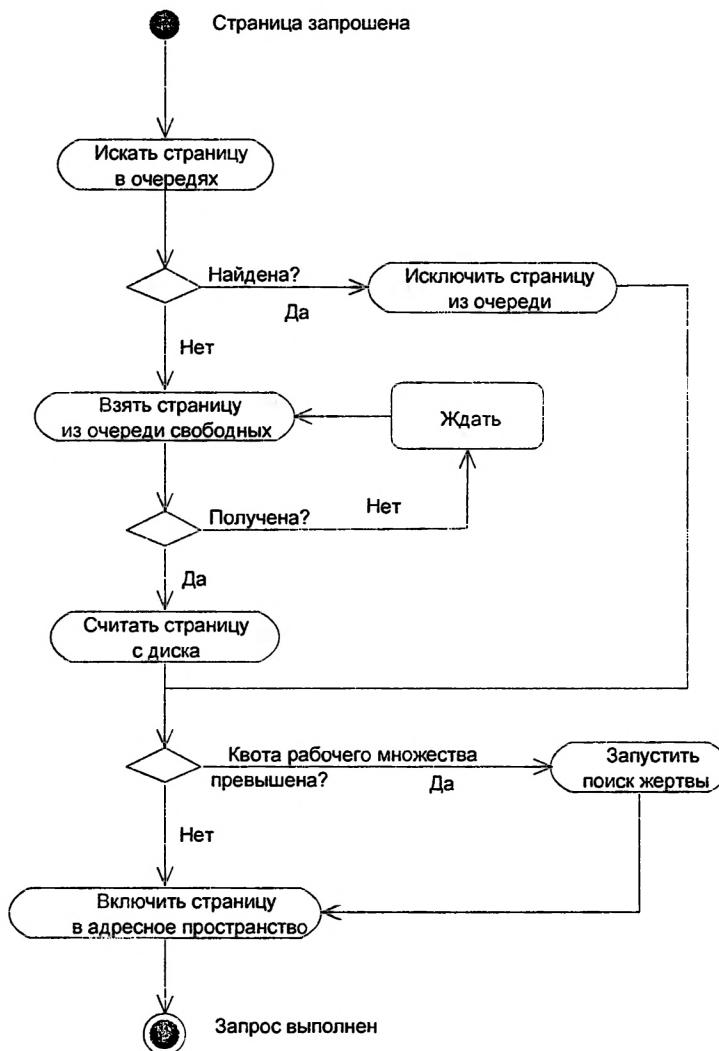


Рис. 5.20. Обработка страницочного отказа (блок-схема)

Такой алгоритм порождает много лишних страничных отказов, но обеспечивает большую экономию обращений к диску. У VAX/VMS эта стратегия сочетается с управлением объемом пула страниц: у каждого пользователя есть квота на объем рабочего множества запущенных им программ. При превышении квоты ОС и осуществляет поиск жертвы среди адресных пространств задач этого пользователя. При разумном управлении этими квотами система обеспечивает весьма хорошие показатели даже при небольших объемах оперативной памяти.

Windows NT (и более поздние версии этой системы, 2000/XP/2003) пытается управлять пулом свободных страниц динамически, не предоставляя администратору никаких средств для прямой настройки. Поэтому на нехватку ОЗУ или нетипичный режим использования памяти эта система реагирует резким падением производительности. По-видимому, это падение обусловлено развитием автоколебаний в динамической настройке пула свободных страниц.

Так, автору удавалось привести в неработоспособное состояние Windows 2000 Workstation с 256 Мбайт ОЗУ, всего лишь открыв по ошибке для редактирования в far файл объемом 64 Мбайт. Файл не превосходил 1/4 доступной оперативной памяти. Тем не менее машина "впала в кому", не реагируя даже на клавиши $<\text{Ctrl}>+<\text{Alt}>+<\text{Del}>$, и находилась в этом состоянии достаточно долго для того, чтобы собрать вокруг нее консилиум системных администраторов и, в конце концов, решить, что проще нажать на кнопку системного сброса.

Фирма Microsoft вполне сознает ущербность принятой стратегии управления памятью и официально не рекомендует запускать на серверах под управлением Windows NT/2000/XP более одного приложения или ресурсоемкого сервиса.

Пул свободных страниц, в который входят как действительно свободные, так и отобранные у задач в качестве "жертв" страницы, в той или иной форме поддерживают все ОС, использующие страничный обмен, однако обычно этот пул отделяют от дискового кэша и при полной загрузке он не превосходит нескольких процентов ОЗУ. При запросах ядра и страничных отказах система выделяет страницы из этого пула, и только при падении его объема ниже некоторого предела начинает поиск жертв в адресных пространствах активных процессов.

Наиболее справедливым будет удалять тот объект, к которому дольше всего не было обращений в прошлом — *LRU* (Least Recently Used). Такой подход требует набора сведений обо всех обращениях. Например, диспетчер памяти должен поддерживать в дескрипторе каждой страницы счетчик обращений, и при каждой операции чтения или записи над этой страницей увеличивать этот счетчик на единицу. Это требует довольно больших накладных расходов — в ряде работ, например в [Краковяк 1988], утверждается, что они будут недопустимо большими.

Остроумным приближением к алгоритму LRU является так называемый *clock-алгоритм*, применяемый во многих современных ОС, в том числе в системах семейства Unix. Он состоит в следующем (рис. 5.21):

1. Дескриптор каждой страницы содержит бит, указывающий, что к данной странице было обращение. Этот бит называют *clock-битом*.

2. При первом обращении к странице, в которой clock-бит был сброшен, диспетчер памяти устанавливает этот бит.
3. Программа, занимающаяся поиском жертвы, циклически просматривает все дескрипторы страниц. Если clock-бит сброшен, данная страница объявляется жертвой, и просмотр заканчивается до появления потребности в новой странице. Если clock-бит установлен, то программа сбрасывает его и продолжает поиск.

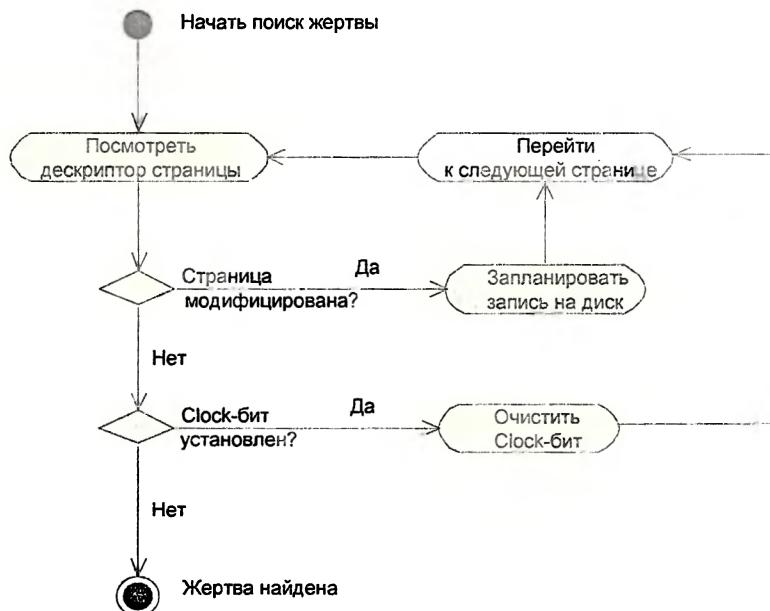


Рис. 5.21. Clock-алгоритм (блок-схема)

Название *clock*, по-видимому, происходит от внешнего сходства процесса циклического просмотра с движением стрелки часов (рис. 5.22). Очевидно, что вероятность оказаться жертвой для страницы, к которой часто происходят обращения, существенно ниже. Накладные расходы этого алгоритма гораздо меньше, чем у LRU: вместо счетчика мы храним только один бит и изменяем его не при каждом обращении к странице, а только при первом обращении после прохода "стрелки".

Практически все известные автору современные диспетчеры памяти предполагают использование *clock*-алгоритма. Такие диспетчеры хранят в дескрипторе страницы или сегмента два бита — *clock*-бит и признак модификации. Признак модификации устанавливается при первой записи в страницу/сегмент, в дескрипторе которой этот признак был сброшен.

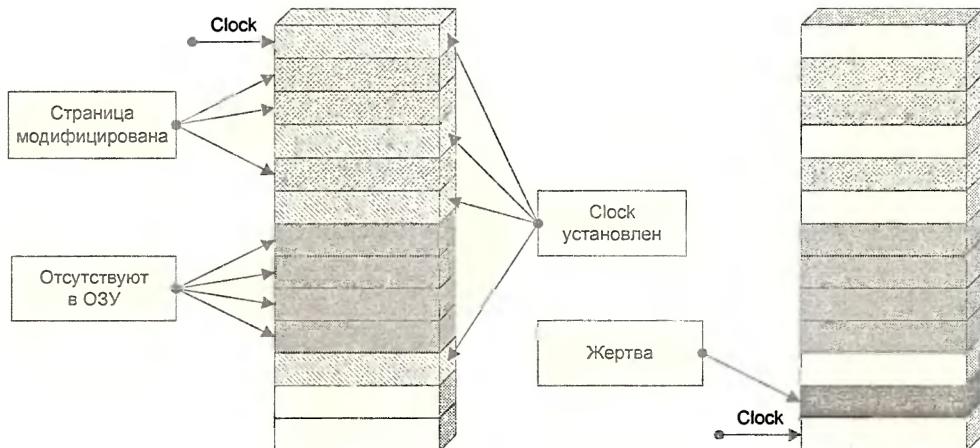


Рис. 5.22. Работа clock-алгоритма

Имитация clock-алгоритма

Ранние версии процессора VAX не имели аппаратно реализованного clock-бита. BSD Unix на этих процессорах реализовал clock-алгоритм, используя для этого бит отсутствия страницы.

Читателю предлагается детально представить себе алгоритм такого использования и найти отличия между этой стратегией и FIFO-алгоритмом в исполнении VAX/VMS.

Экспериментальные исследования показывают, что реальная производительность системы довольно слабо зависит от применяемого алгоритма поиска жертвы. Статистика исполнения реальных программ говорит о том, что каждая программа имеет некоторый набор страниц, называемый *рабочим множеством*, который ей в данный момент действительно нужен. Размер такого набора сильно зависит от алгоритма программы, он изменяется на различных этапах исполнения и т. д., но в большинстве моментов мы можем довольно точно указать его.

Если рабочий набор запущенных программ превосходит оперативную память, частота страницных отказов резко возрастает. При нехватке памяти программе почти на каждой команде требуется новая страница, и производительность системы катастрофически падает.

В системах коллективного пользования размер памяти часто выбирают так, чтобы система балансировала где-то между состоянием, когда все программы держат свое рабочее множество в ОЗУ, и *трэшингом*.

Точное положение точки балансировки определяется в зависимости от соотношения между скоростью процессора и скоростью обмена с диском, а также от потребностей прикладных программ. Во многих старых учебниках реко-

мендуется подбирать объем памяти так, чтобы канал дискового обмена был загружен на 50% [Краковяк 1988].

Еще одно эмпирическое правило приписывается архитектору линии компьютеров IBM 360 Джину Эмделу (Gene Amdahl): сбалансированная система должна иметь по мегабайту памяти на каждый MIPS (Million of Instructions Per Second — миллион операций в секунду) производительности центрального процессора. Если система не использует память, определенную по этой формуле, есть основания считать, что процессор также работает с недогрузкой. Иными словами, это означает, что вы купили слишком мощный для ваших целей процессор и заплатили лишние деньги.

Это правило было выработано на основе опыта эксплуатации больших компьютеров четвертого поколения, в основном на задачах управления базами данных. Скорость дисковой подсистемы в этих машинах была примерно сравнима с дисковыми контроллерами современных персональных компьютеров, поэтому в первом приближении этот критерий применим и к ПК, особенно работающим под управлением систем с виртуальной памятью — OS/2, Windows NT и системами семейства Unix.

В любом случае, для выдачи рекомендаций требуется анализ смеси приложений, которая будет исполняться в системе, и других требований к ней.

В современных персональных системах пользователь, как правило, работает в каждый момент только с одной-двумя программами, и задержки в выполнении этих программ существенно снижают наблюдаемую скорость системы. Поэтому в таких системах память обычно ставят с большим запасом, так, чтобы при обычной деятельности рабочие множества программ даже близко не подходили к размеру ОЗУ. Отчасти это обусловлено тем, что в наше время динамическая память становится все дешевле и дешевле.

5.6. Управление своп-файлом

Для хранения образов модифицированных страниц система должна выделить какое-то пространство на диске. Для этого может использоваться как раздел диска, так и файл, место для которого выделяется наравне с файлами данных. Большинство современных систем может использовать как тот, так и другой методы, и поддерживает динамическое подключение и отключение своп-файлов и своп-разделов.

При наличии в системе нескольких дисков рекомендуется разделить своп-пространство между всеми (или хотя бы между менее используемыми) дисками. Дело в том, что операции чтения и записи требуют времени на позиционированиечитывающей головки. Пока один накопитель передвигает головку, второй вполне может передавать данные. Понятно, что линейного рос-

та производительности с увеличением числа дисков таким способом не получить: всегда есть вероятность, что все требуемые данные окажутся на одном диске, да еще и находятся в разных местах, но все равно выигрыш в большинстве случаев оказывается значительным.

В те времена, когда компьютеры были большими, для своих часто использовались не диски, а специальные устройства, магнитные барабаны. В отличие от диска, который имеет одну или, реже, несколько головок чтения-записи, барабан имел по одной головке на каждую дорожку. Конечно, это повышало стоимость устройства, но полностью исключало потери времени на позиционирование (рис. 5.23).

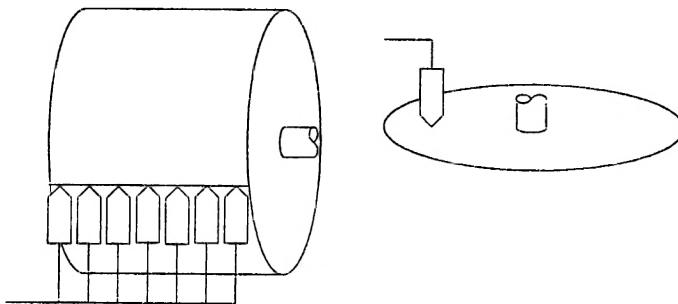


Рис. 5.23. Магнитный диск и магнитный барабан

В swap-файл попадают только страницы, которые изменились с момента загрузки процесса. Если ОС использует абсолютную загрузку или позиционно-независимый код, исполняющийся код не отличается от своего образа в загрузочном файле, поэтому страницы кода вполне можно подкачивать оттуда, и нет никакой необходимости копировать их в swap. Часто при загрузке программы система помещает в память только страницу, на которую указывает стартовый адрес, а весь остальной используемый код и данные подгружаются механизмом страничного обмена.

При загрузке статически инициализированных данных обычно используется стратегия *copy-on-write* (копирование при модификации): первоначально страница подкачивается из файла. Если она не будет модифицирована и ее объявит жертвой, то при повторном обращении ее снова подгрузят из того же файла (рис. 5.24). Только если страница будет изменена, ей выделят место в swap-файле.

Если же используется относительная загрузка или та или иная форма сборки в момент загрузки (разделяемые библиотеки или DLL), при загрузке кода происходит перенастройка адресов. В этом случае возможны два подхода: копировать перенастроенный код в swap или производить перенастройку заново после каждого страничного отказа.

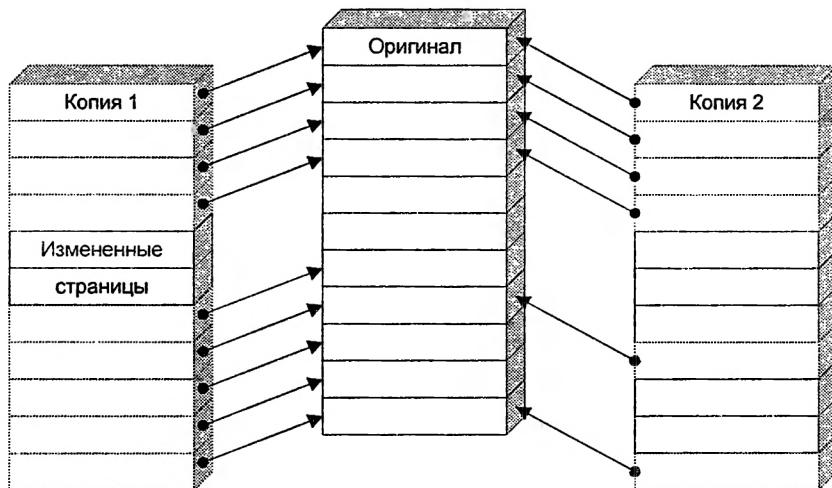


Рис. 5.24. Копирование при модификации

Отображение файлов в память в Unix

Системы семейства Unix предоставляют пользователям доступ к механизмам, используемым при загрузке программ, в виде системного вызова `mmap`. Этот вызов отображает участок файла в память. Отображение возможно в двух режимах: `MAP_SHARED` (изменения в памяти отображаются в файле — таким образом `mmap` можно использовать для реализации разделяемой памяти) и `MAP_PRIVATE` (соответственно, изменения памяти в файле не отображаются — при этом измененные страницы копируются в своп-файл).

Широко используется выделение памяти с помощью отображения псевдофайла `/dev/zero` (файл бесконечной длины, состоящий из одних нулей) в память в режиме `MAP_PRIVATE`.

Даже когда место под своп-файл выделяется динамически, система обычно предоставляет возможность ограничивать его рост. У интерактивных систем при приближении к границе емкости своп-файла система часто начинает выдавать предупреждения пользователю. Однако основным видом реакции на переполнение или превышение лимитов роста своп-файла является отказ выделять память прикладным программам. Поэтому грамотно написанные программы всегда должны проверять, нормально ли завершился запрос на выделение памяти, и по возможности разумно обрабатывать ненормальное завершение. Это нужно не только в том случае, когда программа будет переноситься в систему без виртуальной памяти, но и во вполне штатной (хотя и относительно редкой) ситуации переполнения свопа.

Иногда, впрочем, система может выделять память (точнее, не память, а только адресное пространство) программам без оглядки на то, сколько есть свободного свопа. Эта довольно опасная стратегия, называемая оптимистиче-

ской аллокацией (*optimistic allocation*) или *overcommit*, на первый взгляд кажется бессмысленной или полезной только в очень специальных случаях, например при использовании разреженных массивов. В действительности эта стратегия оправдана и тогда, когда мы можем быть уверены, что большинство выделенных процессу страниц никогда не будут использованы, например, при широком применении стратегии *copy-on-write*.

Оптимистическая аллокация в Unix

В системах семейства Unix копирование при записи применяется не только при загрузке сегментов данных программ и отображении файлов в память, но и при создании задач. Системный вызов *fork* (подробнее обсуждался в главе 3) создает полную копию адресного пространства процесса, выполнившего этот вызов. Физического копирования, естественно, не происходит. Вместо этого система отображает память родительского процесса в адресное пространство потомка и устанавливает защиту от записи на все страницы обеих задач. Когда какая-то из задач пытается осуществить запись, соответствующая страница физически копируется, и запись осуществляется уже в копию. Большинство рожденных задач исполняют системный вызов *exec* вскоре после создания, изменив лишь несколько переменных в своем адресном пространстве. При таком стиле работы с памятью, действительно, многие выделяемые страницы не используются никогда, а большинство из используемых только прочитываются, поэтому *overcommit* является стандартной стратегией выделения памяти в Unix.

5.7. Одноуровневая память

И каждый уже десять лет учит роли,
О которых лет десять как стоит забыть.
Б. Гребенников

Эффективное управление рабочими наборами пользовательских программ и, с другой стороны, эффективное кэширование запросов к дискам позволяют если и не скрыть полностью, то в значительной мере сгладить различие в производительности оперативной и внешней памяти компьютера. Поэтому сразу же после возникновения первых машин с виртуальной памятью начались попытки спрятать все остальные различия между этими двумя типами памяти от программиста, реализовав так называемую *одноуровневую память*.

Интерес к этой идеи сохраняется до сих пор. Например, на сайте [dz.yandex.ru] в конце 2000 года была серия публикаций и довольно бурная дискуссия о достоинствах и недостатках "персистентных объектов" — объектов в терминах объектно-ориентированного программирования, которые переживают перезагрузки и выключения питания системы. Для хранения таких объектов может использоваться как флэш-память, так и другие формы энергонезависимой памяти, те же жесткие диски. Владелец сайта и инициатор дис-

куссии, Дмитрий Завалишин, отстаивал тезис о том, что такие объекты представляют собой сокровенную мечту и своего рода Священный Грааль всего программирования и развития вычислительной техники.

Одноуровневая память в Multix

Пионером в реализации одноуровневой памяти была ОС Multix фирмы Honeywell. Эта система была разработана в конце 60-х годов и оказала огромное влияние на развитие вычислительной техники как прямо, так и посредством своего потомка Unix. Несколько машин с этой ОС эксплуатировались и были доступны через Internet (во всяком случае, отвечали на запрос ping) еще в 1997 году.

Multix предоставляла средства отображения файлов на адреса оперативной памяти наравне с более традиционными средствами ввода-вывода. Ранние версии Unix предназначались в том числе и для работы на машинах без диспетчера памяти или с виртуальной памятью на основе базовых регистров, где возможен лишь традиционный ввод-вывод. Однако современные системы этого семейства отчасти вернулись к истокам и также предоставляют этот способ доступа к файлам.

Впрочем, как отмечалось ранее, имеющийся в современных Unix-системах mmap скорее представляет собой документированный внутренний интерфейс загрузчика, чем полноценное средство организации одноуровневого доступа: работа с отображенными файлом не полностью прозрачна для пользователя. Например, изменения содержимого отраженного ОЗУ и наоборот, изменения, внесенные в файл с момента отображения, необходимо синхронизировать друг с другом вручную, используя системный вызов msync. Средства, предоставляемые для этой цели Windows NT/2000/XP, более прозрачны и просты в использовании, но тоже применяются относительно редко.

Для того чтобы понять, возможна ли полностью одноуровневая память, и если да, то в какой мере, давайте сначала установим различия между ОЗУ и наиболее распространенным типом внешней памяти, жестким магнитным диском.

Во-первых, объем оперативной памяти в современных компьютерах измеряется десятками и сотнями мегабайт, а у систем коллективного пользования достигает нескольких гигабайт. Характерная емкость жесткого диска начинается с нескольких гигабайт (диски меньшего объема просто не производятся) и заканчивается сотнями гигабайт. Если мы хотим адресовать все это единым образом, мы должны расширить адресное пространство. 32 бит явно недостаточно, 64 бит на ближайшие годы хватит, но рост емкости дисковых массивов идет по экспоненте.

Впрочем, расширение адресного пространства само по себе не представляет большой проблемы — мало 64 бит, сделаем 128, тем более что речь идет не об адреснойшине процессора, используемой только для адресации ОЗУ, а о виртуальном адресе. Современные технологии без особых проблем позволя-

иот упаковать арифметико-логическое устройство и регистры такой разрядности в один кристалл, сделать корпус с надлежащим числом ног, печатную плату, в которую можно впаять такой корпус, и автоматическую линию, которая будет распаивать корпуса по платам. Да, это будет не Spectrum, вручную не спаяешь, — но и материнскую плату современного РС-совместимого компьютера вручную невозможно развести и спаять. Ну и что?

Во-вторых, оперативная память теряет свое содержимое при выключении питания, а жесткий диск — сохраняет, поэтому нередко используется еще одна характеристика дисковой памяти как противопоставление оперативной — постоянная память. Можно вспомнить и о промежуточных решениях, например о флэш-памяти и ее аналогах, которые адресуются почти как ОЗУ, а данные хранят почти как жесткий диск. Наличие промежуточных решений наводит на мысль, что особых проблем с этой стороны ждать не приходится. На самом деле, проблема здесь есть, но обсудим мы ее чуть позже.

В-третьих, и это связано сразу с обеими ранее названными причинами, человек снисходит до ручного наведения порядка на диске (удаления мусора и пр.) гораздо чаще, чем до выполнения той же операции в ОЗУ. Поэтому жесткие диски обычно снабжаются еще одной схемой адресации, ориентированной на использование человеком: когда вместо адреса, представляемого целым числом, используется символическое имя.

Трансляцию имени в адрес (сектор, поверхность и дорожку жесткого диска) осуществляет файловая система. Вопросы организации файловых систем, каталогов и управления структурами свободной и занятой дисковой памяти обсуждаются в главе 10.

Единственная из используемых в настоящее время архитектур, предоставляющая "честную" одноуровневую память, AS/400, имеет два представления указателя, неразрешенное — с именем в качестве селектора сегмента, и разрешенное — с бинарным представлением этого селектора. Можно себе представить и другие механизмы трансляции имен в адреса, например получение указателя посредством исполнения системного вызова

```
void *resolve(char * object_name, int flags)
```

или чего-нибудь в этом роде. Особых технических проблем это не представляет, вопрос в том, надо ли это.

Изложение одного из доводов в пользу того, что это надо далеко не всегда, мы предлагаем начать издалека, а именно с весьма банального тезиса, что писать программы без ошибок человечество до сих пор не научилось и вряд ли научится в обозримом будущем. Исполнение программы, содержащей ошибки, может порождать не только обращения по неверным указателям и выход за границы массивов (наиболее разрушительные типы ошибок, от ко-

торых сегментные и страницные диспетчеры памяти предоставляют определенную защиту), но и более тонкие проблемы — фрагментацию и/или утечку свободной памяти и различные рассогласования (в СУБД применяют более точный термин — нарушения целостности данных).

Накопление этих ошибок рано или поздно приводит к тому, что программа теряет способность функционировать. Потеря этой способности может быть обнаружена и пользователем ("что-то прога глючит", "зависла"), и системой (исчерпание квоты памяти или других ресурсов, превышение лимитов роста своп-пространства или доступ по недопустимому адресу), и даже самой программой — если существуют формальные критерии целостности данных, в различных местах кода могут встречаться проверки этих критериев.

Учебники по программированию, например [Дейкстра 1978], настоятельно рекомендуют вырабатывать такие критерии и вставлять соответствующие проверки везде, где это целесообразно. Понятно, что забывать о здравом смысле и вставлять их после каждого оператора, или даже лишь перед каждой операцией, для исполнения которой требуется целостность, далеко не всегда оправдано с точки зрения производительности, так что при реальном программировании надо искать баланс.

Иногда в ходе таких проверок даже удается восстановить целостность (примеры алгоритмов проверки и восстановления структур файловой системы приводятся в главе 12), но очевидно, что далеко не всегда это возможно. В этом случае остается лишь проинформировать пользователя, что у нас "Assertion failed" (предположение нарушено), и по возможности мирно завершиться (рис. 5.25).

Сохранять при этом данные в постоянную память опасно: если мы не можем восстановиться, мы часто не можем и знать, насколько далеко зашло нарушение целостности, поэтому сохранение чего бы то ни было в таком состоянии чревато полной или частичной (что тоже неприятно) потерей информации. В частности, именно из этих соображений ОС общего назначения, обнаружив ошибку в ядре, сразу рисуют регистры на консоли (рискуя при этом целостностью файловых систем и пользовательских данных), а не предлагают пользователю предпринять какие-либо меры по сохранению данных.

Смысл останова задачи или всей системы с последующим ее перезапуском состоит в том, чтобы заново проинициализировать структуры данных, используемые при работе программного обеспечения. Это действие можно описать как "контролируемое забывание" всего плохого, что накопилось в памяти за время работы программы, и начало с более или менее чистого листа.

Сервис автоматического перезапуска в различных формах предоставляется многими приложениями, ОС и даже аппаратными архитектурами. Некоторые формы реализации этого сервиса рассматриваются в разд. 12.2.1.

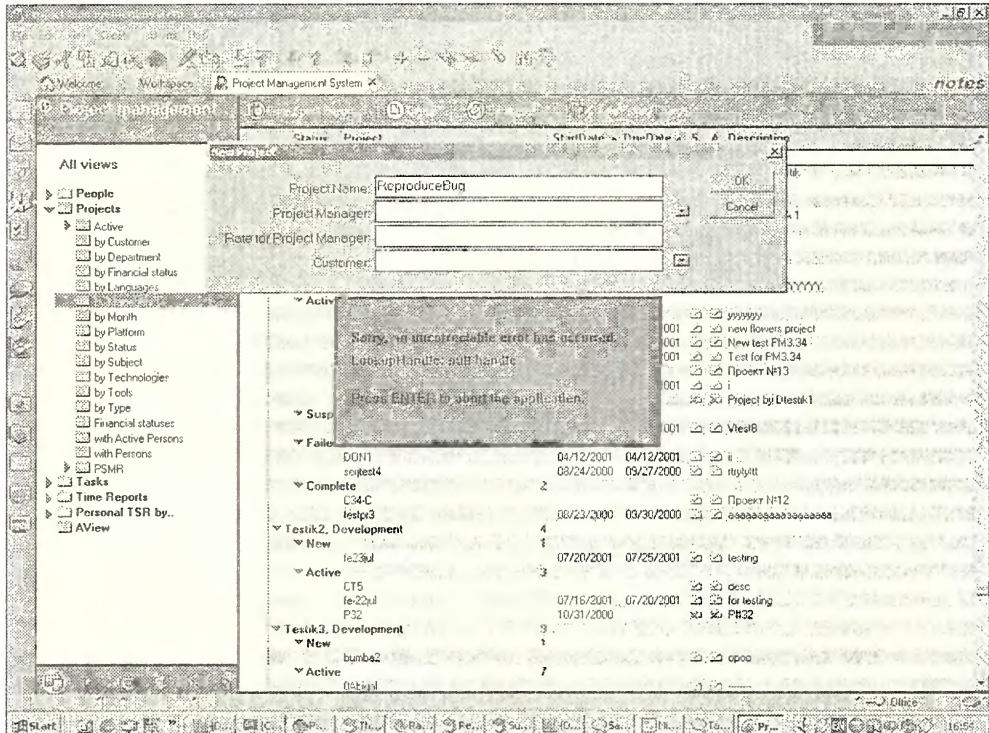


Рис. 5.25. Красный экран смерти клиента Lotus Notes

Понятно, что средства перезапуска не могут заменить собой отладку и исправление ошибок в прикладных и системных программах и являются скорее последней линией обороны (если даже не действиями, предпринимаемыми от отчаяния) в борьбе с системными сбоями. Но пока человечество не научится писать программы без ошибок, возможность "привести в чувство" обезумевшую программу путем ее перезапуска остается жизненно необходимой.

Понятно также, что если программа полностью сохраняет свое состояние в постоянной памяти, ее перезапуск нам ничем не поможет: программа честно восстановит весь тот мусор, который накопился в ее сегментах и файлах данных за время предыдущей сессии, и честно воспроизведет снова тот сбой, из-за которого и потребовался рестарт.

В этом смысле крайне желательно держать под контролем перенос данных из постоянной памяти в оперативную, а особенно в обратном направлении, избегая его полной "прозрачности". Каждая дополнительная точка сохранения состояния системы повышает риск воспроизведения сбоя или даже возникновения новых проблем, порожденных в файлах сохранения состояния во время

чрезмерно "жесткого" перезапуска. Тем более недопустимо неконтролируемое возникновение этих точек при "прозрачном" сохранении объектов.

Реестр Win32

В свете этих рассуждений системный реестр Win32, в который все приложения сохраняют все, что считают нужным, не имеющий адекватных средств восстановления и самоконтроля, представляет собой если и не сознательное вредительство, то, во всяком случае, недостаточно продуманное техническое решение — из-за него многие проблемы, для исправления которых в более продуманных с этой точки зрения системах достаточно перезагрузки или очистки конфигурационного файла, в Win32 приходится решать переустановкой ОС.

Если говорить именно о настройках ОС, радикальнее всего эта проблема решена в современных версиях FreeBSD (свободно распространяемой системе семейства Unix), в которой все файлы настройки ОС и системных сервисов включены в систему контроля версий, обеспечивающую полный или частичный откат на неограниченное число модификаций назад. Собственно, это может быть реализовано в любой ОС, которая хранит свою конфигурацию в текстовом формате, стандартными средствами контроля версий, используемыми для разработки программного обеспечения, — CVS и др.

В свете приведенных ранее рассуждений полезно разделять оперативные и хранимые объекты не только по способу адресации, но и по представлению данных. Эти представления должны удовлетворять различным и не всегда совместимым требованиям: при выборе внутреннего, *оперативного представления данных* основные критерии — это скорость, удобство доступа и умопостижимость кода, который будет с этими данными работать, а для внешнего, *хранимого представления* — прежде всего легкость проверки и восстановления, если это возможно, целостности данных. При разработке внешнего формата данных желательно также принять во внимание соображения межмашинной совместимости — возможные различия в порядке байтов и даже битов в целочисленных значениях, особенности представления чисел с плавающей точкой, различия кодировки текста и т. д.

Если хранимое и оперативное представления объектов различны, одноуровневая память для нас скорее вредна, чем бесполезна: до того как программа сможет работать с объектом, она должна преобразовать его во внутреннее представление (в объектно-ориентированных языках это может делать один из конструкторов объекта). Эту процедуру обычно оказывается целесообразно совместить со считыванием внешнего представления объекта из файла. "Прозрачная" же для пользовательской программы запись данных во внешний формат бывает и просто опасна — нередко для обеспечения целостности данных оказывается необходим контроль над порядком записи тех или иных полей и структур.

Объединение оперативной и долговременной памяти, таким образом, оказывается применимо лишь в тех ситуациях, когда нам, во-первых, удалось

разработать модель данных, одновременно удовлетворяющую требованиям, предъявляемым и к оперативному, и к хранимому представлениям, и вторых, когда нас не беспокоит опасность нарушения нашей модели данных из-за неполного их сохранения в момент системного сбоя (или когда мы имеем какие-то средства предотвращения этой опасности). Если эти условия не соблюдаются, программист обязан, как минимум, сохранять контроль над тем, когда данные переносятся из оперативной памяти в долговременную, даже если он и не контролирует, как это происходит.

В этом смысле интересен пример современных реляционных баз данных, в которых модификация данных обычно осуществляется не над рабочими таблицами, а над их копией, создаваемой в так называемом сегменте отката. Перенос модифицированных данных в рабочую таблицу, которая, собственно, и является хранимым представлением данных, происходит только при исполнении специального оператора `COMMIT` (исполнить, завершить). Если программа или система в целом "упадет", не выполнив этот оператор, изменения будут потеряны, но, во всяком случае, рабочие таблицы останутся в согласованном состоянии. Несколько подробнее подобные стратегии работы с хранимыми данными обсуждаются в разд. 11.4.3. Легко понять, что автоматически расставлять операторы `COMMIT` в произвольном коде невозможно, программист должен следить за тем, когда началась группа модификаций, результат которой надлежит сохранить; в ряде случаев также оказывается целесообразно предоставить пользователю контроль над моментами такого сохранения.

Возможно, один из путей к достижению полной однородности памяти (или, как любят говорить современные программисты, работающие с объектно-ориентированными языками, полной персистентности) — это введение в языки программирования аналогов оператора `COMMIT`; при этом для реализации сегмента отката можно было бы использовать аналоги механизма `copy-on-write`, т. е. современные процессоры со страничной виртуальной памятью уже имеют все необходимое для реализации таких механизмов. К сожалению, теория, которая бы позволила создавать при работе с произвольными объектами в оперативной памяти точки восстановления и точки сохранения состояния, находится в зачаточном состоянии.

Безусловно, средства для отображения файлов в память лучше иметь, чем не иметь. К тому же их можно использовать и для других целей, кроме собственно организации одноуровневого доступа к данным — для загрузки программ, выделения памяти или эмуляции сегментов данных, разделяемых между задачами и даже между машинами (при доступе к файлу по сети).

Важно еще подчеркнуть, что разделение представлений данных на внешние и внутренние не обязано полностью соответствовать способу их хранения —

в ОЗУ или на диске. Кроме хранения оперативных данных в own-пространстве и разного рода "виртуальных дисков" можно привести и более радикальный пример: таблицы, индексы и прочие файлы данных сервера реляционной СУБД играют роль, скорее, оперативного представления данных, роль же хранимого представления в данном случае играют форматы, используемые для экспорта и резервного копирования содержимого таблиц.

Благодаря этому примеру становится понятнее, почему единственная из коммерчески применяемых в настоящее время систем с одноуровневой адресацией — AS/400 — ориентирована на использование в качестве сервера СУБД. В литературе, особенно в рекламной, даже встречается ее описание как "аппаратного сервера баз данных".

Примечание

Вообще, описание специализированных компьютеров как "аппаратное что-то там" — нередко встречающийся, остроумный и довольно эффективный маркетинговый прием. Понятно, что чем более короткий и однозначный ответ даст технический специалист на вопросы "принимающих решения", тем легче ему будет обосновать конкретный выбор. Поэтому наравне с грамотным и исчерпывающим описанием технических достоинств, хорошее рекламное описание должно в явном или неявном виде содержать и варианты ответов на многие распространенные вопросы со стороны нетехнического персонала.

Так, если начальник спрашивает администратора: "Вот, купим мы этот компьютер, — моя секретарша сможет на нем в Lines играть?", тот может ему ответить: "А это не компьютер, это аппаратный..." маршрутизатор (Cisco), сервер СУБД (AS/400), веб-сервер (попытки продавать такие серверы на основе Linux делались, но большого успеха не имели), нужное подставить.

Вопросы для самопроверки

1. Какие проблемы призвана решать страничная и сегментная виртуальная память?
2. Какие преимущества сегментная и страничная виртуальная память имеют по сравнению с виртуальной памятью на основе базовой адресации? Каковы их недостатки?
3. Чем сегментная виртуальная память отличается от страничной?
4. Почему большинство современных процессоров использует многоуровневые таблицы трансляции?
5. Как, несмотря на размещение таблиц трансляции в ОЗУ, удается избежать падения производительности? Какое из времен обращения к памяти при этом оптимизируют — среднее (наиболее распространенные на практике сценарии работы с памятью) или максимальное (теоретически возможный

наихудший случай)? Иными словами, для каких приложений оптимизированы механизмы ускорения доступа — для приложений разделенного или реального времени?

6. Каковы недостатки двухуровневой схемы управления доступом (пользователь/супервизор), используемой большинством моделей современных процессоров с виртуальной памятью и большинством современных ОС? Могут ли эти недостатки быть устранены посредством перехода к трех- или четырехуровневой схеме управления доступом, например, такой как в процессорах VAX и x86?
7. Почему диспетчеры памяти, реализующие взаимно недоверяющие подсистемы, не достигли коммерческого успеха? Можно ли получить преимущества, характерные для таких диспетчеров памяти, другими средствами? Например, нельзя ли вместо отзыва мандатов использовать те или иные формы сборки мусора? Действительно, ведь сборка мусора гарантирует, что мы можем повторно использовать только ту память, на которую не осталось ни одного указателя.
8. Благодаря каким свойствам реальных программ возможно и целесообразно использовать страничную подкачку?
9. Какие преимущества дает страничная подкачка? Каковы ее недостатки?
10. Для какого из классов приложений (интерактивных, разделенного и реального времени) наиболее важны преимущества страничной подкачки? Для какого — недостатки?
11. Почему так важно, чтобы прерывание по страничному отказу обрабатывалось процессором как исключение (т. е. с отменой уже совершенных действий текущей команды), а не как собственно прерывание (после завершения текущей команды)? (Для ответа на этот вопрос, возможно, придется ознакомиться с материалом главы 6.)
12. Почему в этой главе утверждается, что влияние алгоритма поиска жертвы на производительность системы со страничной подкачкой относительно невелико? Что тогда оказывает решающее влияние?
13. Может ли реализация стратегии поиска жертвы FIFO в VAX/VMS и Windows NT рассматриваться как приближенная реализация clock-алгоритма? Если нет, то в чем именно состоит отличие?
14. Если алгоритм поиска жертвы слабо влияет на производительность, то почему Windows NT так плохо ведет себя при недостатке физического ОЗУ?
15. Какие проблемы возникают при использовании разделяемых библиотек в системах с виртуальной памятью? Как эти проблемы разрешаются в

Win32? Какие еще вы знаете ОС, использующие это же решение? Как эти же проблемы разрешаются в Unix-системах, использующих формат ELF?

16. Каковы преимущества и недостатки оптимистической аллокации виртуальной памяти? Почему в этой главе утверждается, что системы семейства Unix вынуждены использовать оптимистическую аллокацию?
17. Какие проблемы ограничивают применение систем с одноуровневой (т. е. не разделяющейся на оперативную и долговременную) памятью и отображения файлов в оперативную память? Можно ли доказать, что эти проблемы в общем случае неразрешимы?
18. Приведите примеры хранения оперативных данных на диске и долговременных данных — в оперативной памяти.
19. Почему программист должен иметь контроль над временем и порядком переноса данных из оперативной в долговременную память и обратно? Приведите примеры случаев, когда целесообразно и когда необходимо давать пользователю возможность контролировать момент такого переноса.



ГЛАВА 6

Компьютер и внешние события

Мы ждали его слишком долго.
Что может быть глупее, чем ждать?
Б. Гребенников

Практически все функции современных вычислительных систем так или иначе сводятся к обработке внешних событий. Единственная категория приложений, для которых внешние события совершенно неактуальны, — это так называемые *пакетные приложения*, чаще всего — вычислительные задачи. Доля таких задач в общем объеме компьютерных приложений в наше время невелика и постоянно падает. В остальных же случаях, даже если не вспоминать о специализированных управляющих компьютерах, серверы обрабатывают внешние по отношению к ним запросы клиентов, а персональный компьютер — реагирует на действия пользователя. Различие между управляющими системами (приложениями реального времени) и системами общего назначения состоит лишь в том, что первые должны обеспечивать гарантированное время реакции на событие, в то время как вторые "всего лишь" должны предоставить хорошее среднее время такой реакции и/или обработку большого количества событий в секунду.

Единственный способ, которым фон-неймановский компьютер может отреагировать на что бы то ни было — это выполнить программу, последовательность команд. В случае внешнего события естественным решением кажется предоставить команду условного перехода, условием которого является признак события. В системах команд микроконтроллеров часто реализуют именно такие переходы (см., например, табл. 2.1). В качестве признака события в этом случае используется значение одного из битов специального регистра, биты которого соответствуют входам микросхемы контроллера. Бит равен единице, если на соответствующий вход подано высокое напряжение, и нулю — если низкое.

Наличие таких команд полезно, но решает проблему не полностью: да, если событие произошло, мы можем вызвать программу и осуществить реакцию, но что мы будем делать, если события еще не происходило?

6.1. Опрос

Наивное решение состоит в том, что нам следует циклически опрашивать признак события (пример 6.1). Это решение хорошо не только концептуальной простотой, но и тем, что если цикл опроса короток, время реакции будет очень маленьким. Поэтому такой метод нередко используют для обработки последовательностей событий, следующих друг за другом с небольшим интервалом. Однако это решение, называемое *опросом* (polling), имеет и большой недостаток: загрузив процессор опросом, мы не можем занять его чем бы то ни было другим.

Пример 6.1. Пример использования режима опроса

```
; Приведенный фрагмент кода использует опрос таймера TMR0,  
; работающего от "часового" кварцевого генератора с частотой 32768Гц.  
; Цикл опроса в чистом виде  
; TMR0 – регистр таймера,  
; TimerValue – просто переменная,  
; регистр 0 – аккумулятор, обозначаемый также как W.  
; Такой цикл ожидает одного отсчета таймера.  
  
MovF TMR0, 0  
MovWF TimerValue  
G5H_Continue1  
MovF TimerValue, 0  
SubWF TMR0, 0  
BNZ G5H_Continue1  
  
; Код содержит два цикла опроса: первый цикл генерирует  
; сигнал высокого напряжения, второй – низкого.  
; В результате получается периодический сигнал, называемый меандром.  
; Фрагмент определителя номера на основе микроконтроллера PIC  
; (c) 1996, Дмитрий Иртегов.  
; Запрос к АТС на выдачу номера.  
; Генератор меандра с частотой 501.5 Гц. Выдает 50 периодов (100 мс).  
; Генерирует 2 периода по 16 тиков и один – по 17.  
; Получается очень похоже.
```

```
Generate500Hz
MovLW 50
MovWF AONByteCounter
MovLW 3
MovWF Tmp1
MovF TMRO, 0
MovWF TimerValue
G5H_NextPeriod
MovLW 8
AddWF TimerValue, 1
BSF LINE_CTL_PORT, LINE_ANSWER
G5H_Continue1
MovF TimerValue, 0
SubWF TMRO, 0
BNZ G5H_Continue1
MovLW 8
AddWF TimerValue, 1
BCF LINE_CTL_PORT, LINE_ANSWER
DecFSZ Tmp1, 1
GoTo G5H_Continue0
MovLW 3
MovWF Tmp1
IncF TimerValue, 1
G5H_Continue0
MovF TimerValue, 0
SubWF TMRO, 0
BNZ G5H_Continue0
DecFSZ AONByteCounter, 1
GoTo G5H_NextPeriod
Return
```

Этот недостаток можно переформулировать и иначе: если процессор занят чем-то другим, он может узнать о событии, только завершив текущую деятельность. Если событие действительно важное, впрочем, мы можем расставить команды проверки его признака по всему коду программы, но для сложных программ, обрабатывающих много различных событий, это решение вряд ли можно считать практичным.

С точки зрения встраиваемых приложений, режим опроса имеет еще один существенный недостаток: опрашивающий процессор нельзя выключить. В же время, выключенный процессор потребляет гораздо меньше энергии

и не создает электромагнитных помех, поэтому при разработке программ для таких приложений считается хорошим тоном выключать (переводить в режим ожидания) процессор всегда, когда это возможно. В этом случае, конечно, необходимо предусмотреть какие-либо средства для вывода процессора из этого состояния при возникновении интересующего нас события.

6.2. Канальные процессоры и прямой доступ к памяти

Одно из решений состоит в том, чтобы завести отдельный процессор и поручить ему всю работу по запросу. Процессор, занимающийся только организацией ввода-вывода, называют *периферийным* или *канальным* (*channel*). Канальные процессоры получили распространение во второй половине 60-х годов XX века. Наиболее известны канальные процессоры в машинах IBM System 360/370/390. Эти машины имеют один или несколько полностью программируемых канальных процессоров, которые подключены непосредственно к системной шине и могут обслуживать одно или несколько устройств, в зависимости от топологии подключения устройств и загруженной в процессор канальной программы. Как правило, канальный процессор стоит значительно дешевле центрального — так, IBM нередко использовала в очередной машине System 370 центральный процессор одной из предыдущих моделей (разумеется, с измененным микрокодом — системы команд центрального и канальных процессоров System 370 различаются) в качестве канального. Таким образом, производительность системы с канальным процессором выше, чем у однопроцессорной системы, а стоимость — ниже, чем у системы с двумя одинаковыми процессорами.

Периферийные процессоры находят широкое применение в современных вычислительных системах. Так, типичный современный персональный компьютер, кроме центрального процессора, обычно имеет и специализированный видеопроцессор, так называемый *графический ускоритель*. Во многих кэширующих дисковых контроллерах и аппаратных реализациях RAID (см. разд. 9.6.2) обычно также есть собственный процессор; в данном случае, как правило, используются полностью программируемые процессоры. Лазерные и струйные печатающие устройства имеют процессор, который интерпретирует команды языка управления принтером (PCL или Postscript); в акустических, радио- и кабельных модемах часто используются специализированные процессоры DSP. Нередко, если немедленно после события требуется лишь простая обработка, а сложные вычисления можно отложить на потом, канальный процессор можно упростить и сделать существенно дешевле центрального.

Так, при работе с контроллерами дисков, лент и других устройств массовой памяти возникает задача копирования отдельных байтов (или, в зависимости от разрядности шины контроллера, полуслов или слов) из контроллера в память и обратно. Передача одного блока (512 байт у большинства современных контроллеров) состоит из 128 операций передачи слова, идущих друг за другом с небольшими интервалами. Темп передачи данных определяется скоростью вращения диска или движения ленты. Этот темп обычно ниже скорости системной шины, поэтому передача данных должна включать в себя опрос признака готовности контроллера принять или предоставить следующее слово. Интервал между словами обычно измеряется несколькими циклами шины. Нередко бывает и так, что частоты шины и контроллера не кратны, поэтому последовательные слова надо передавать через различное число циклов.

Дополнительная сложность состоит в том, что, не предоставив вовремя следующее слово для записи, мы испортим весь процесс — эта проблема особенно серьезна на устройствах однократной записи, например прожигателях компакт-дисков. Аналогично, не успев прочитать очередное слово, мы потеряем его и вынуждены будем отматывать ленту назад или ждать следующего оборота диска.

Видно, что это именно та ситуация, которую мы ранее описывали как показание к использованию режима опроса: поток следующих друг за другом с небольшим интервалом событий, каждое из которых нельзя потерять, а нужно обязательно обработать.

Обработка события, которая нужна, чтобы избежать такой неприятности, крайне проста, так что устройство, способное с ней справиться, не обязано даже быть полностью программируемым процессором.

При передаче надо всего лишь убедиться, что блок данных не кончился, взять следующее слово из памяти, дождаться готовности устройства, скопировать слово и вернуться к началу алгоритма. Если блок данных кончился или контроллер выдал ошибку, необходимо сообщить об этом центральному процессору.

Для реализации этого алгоритма достаточно трех регистров (указателя в памяти, значения текущего слова и счетчика переданных слов). Реализующее этот алгоритм устройство называют *контроллером прямого доступа к памяти* (Direct Memory Access controller, DMA controller) (рис. 6.1). Такие контроллеры часто рассчитаны на одновременную работу с несколькими устройствами — имеют несколько каналов — и, соответственно, больше регистров. Описание реальной микросхемы контроллера ПДП можно найти в [Паппас/Марри 1993].

Обычно контроллеры ПДП не считают процессорами, однако без большой натяжки можно сказать, что это все-таки канальный процессор, хотя и очень

примитивный. Контроллеры ПДП, рассчитанные на совместную работу с процессором, обладающим виртуальной памятью, часто имеют некий аналог диспетчера памяти ЦП, для того чтобы позволить операционной системе предоставлять указатель для ПДП в виртуальном адресном пространстве, или, во всяком случае, упростить работу по преобразованию виртуального адреса в физический.

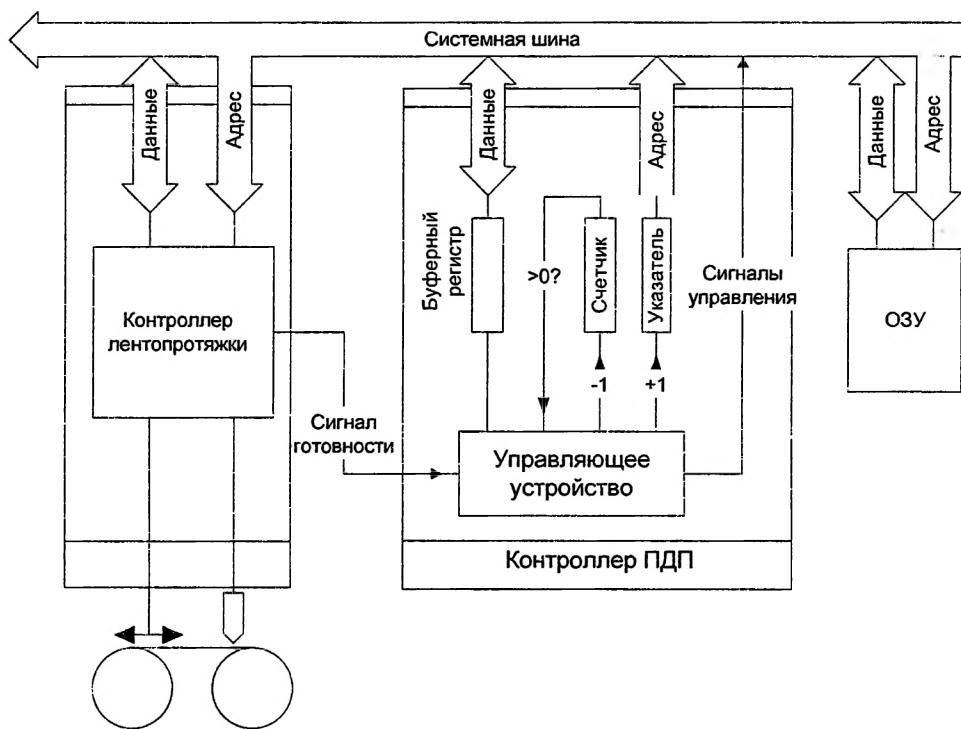


Рис. 6.1. Структура контроллера ПДП

Различают два типа реализаций ПДП:

- мастер шины** (bus master) (в старой русскоязычной литературе встречалось также словосочетание *задатчик шины*), когда устройство имеет свой собственный контроллер ПДП;
- централизованный контроллер, устанавливаемый на системной плате и способный работать с несколькими различными устройствами.

В качестве альтернативы ПДП можно предложить снабжение устройства буфером, который работает с частотой системной шины. Центральный процессор передает данные в буфер, и лишь когда заканчивает передачу, иницииру-

ет операцию устройства. Логика работы самого устройства с этим буфером, впрочем, ничем не отличается от ПДП, с той лишь разницей, что используется не общесистемная, а встроенная память. На практике оба подхода часто используются совместно: ПДП позволяет минимизировать загрузку центрального процессора, а буфер — избежать потери данных, если системная шина занята другим устройством.

Типичный современный дисковый контроллер имеет и средства ПДП, и внутренний буфер. У кэширующих и RAID-контроллеров объем буфера может измеряться многими мегабайтами. Кроме того, современные жесткие диски также имеют собственные буфера.

Канальные процессоры долгое время считались отличительной особенностью больших ЭВМ. В мини- и микрокомпьютерах использование специализированных канальных процессоров, более сложных, чем контроллер ПДП, считалось неприемлемым по стоимостным показателям. Удивительно, что даже современное радикальное удешевление оборудования не изменило положения: предложение консорциума I²O (Intelligent Input/Output) снабжать компьютеры на основе процессоров x86 канальным процессором Intel 960, с энтузиазмом поддержанное практически всеми поставщиками операционных систем, почему-то не было столь же горячо поддержано потребителями.

Потребители мини- и микросистем, нуждающиеся в высокой производительности, предпочитают использовать в качестве дополнительных процессоров устройства с той же архитектурой, что и центральный процессор. Это называется *симметричной многопроцессорностью (SMP)* и позволяет перераспределять между процессорами не только обработку событий, но и собственно вычислительную деятельность. Понятно, что обрабатывать все события по принципу опроса в такой архитектуре — бессмысленная, а зачастую и нетерпимая расточительность.

Понятно, впрочем, что использование канальных процессоров и контроллеров ПДП повышает стоимость системы и не решает проблемы радикально — теперь вместо флагов, непосредственно сигнализирующих о внешних событиях, центральный процессор вынужден опрашивать флаги, выставляемые канальным процессором. В зависимости от характера событий и требуемой обработки это решение может оказаться и совсем неприемлемым, например, если на каждое событие требуется немедленная реакция именно центрального процессора.

К счастью, еще с 60-х годов практически все процессоры — как центральные, так и канальные, используют стратегию работы с событиями, во многих отношениях гораздо более совершенную, чем опрос.

6.3. Прерывания

Альтернатива опросу, применяемая практически во всех современных процессорах, называется *прерываниями* (*interrupt*), и состоит в значительном усложнении логики обработки команд процессором.

Процессор имеет один или несколько входов, называемых *сигналами* или *линиями запроса прерывания* (IRQ, Interrupt ReQuest). При появлении сигнала на одном из входов процессор дожидается завершения исполнения текущей команды и вместо перехода к исполнению следующей команды инициирует обработку прерывания.

Обработка состоит в сохранении счетчика команд и, возможно, некоторых других регистров и в передаче управления на адрес, определяемый типом прерывания. По этому адресу размещается программа, *обработчик прерывания* (*interrupt handler*), которая и осуществляет реакцию на соответствующее прерыванию событие. Перед завершением обработчик восстанавливает регистры, и исполнение основной программы возобновляется с той точки, где она была прервана. В современных процессорах регистры обычно сохраняются в стеке; благодаря этому обработчик прерывания также может оказаться прерван.

Как правило, адреса программ, соответствующих различным прерываниям, собраны в таблицу, называемую *таблицей векторов прерываний*, размещаемую в определенном месте адресного пространства. У микроконтроллеров каждому возможному сигналу прерывания обычно соответствует свой *вектор*. Процессоры общего назначения часто используют более сложную схему, в которой устройство, запрашивающее прерывание, передает процессору номер прерывания или сразу адрес обработчика.

Прерывания в PDP-11

Для примера рассмотрим организацию прерываний в машинах семейства PDP-11. Процессоры данной архитектуры сейчас практически не используются в машинах общего назначения, но производятся и применяются в качестве микроконтроллеров. Ряд архитектурных решений PDP-11, разработанной в начале 70-х годов, не потерял актуальности и поныне. В частности, подход к реализации прерываний считается классическим [Кейслер 1986].

Процессоры семейства PDP-11 различают 128 типов прерываний и исключений (чем прерывание отличается от исключения см. далее). Каждому типу соответствует процедура-обработчик. Адреса точек входа всех процедур собраны в таблицу векторов прерываний. Эта таблица занимает 256 слов физической памяти, начиная с нулевого адреса. Каждый элемент таблицы (вектор) содержит адрес обработчика и новое слово состояния процессора. Позже будет объяснено, для чего это сделано.

Процессор узнает о возникновении прерывания, если на один из входов запроса подан сигнал. Обычно этот сигнал генерируется одним из внешних уст-

ройств. Например, прерывание может сигнализировать о завершении операции перемещения головки дисковода или передачи данных в режиме ПДП.

Каждый вход соответствует определенному уровню приоритета. PDP-11 имеет восемь уровней приоритета прерывания. Прерывание происходит, только когда уровень приоритета процессора ниже приоритета запрашиваемого прерывания. Если у процессора установлен приоритет, равный 7, внешние прерывания запрещены. Приоритет процессора задается его словом состояния. Получив запрос, процессор завершает исполнение текущей команды и выставляет сигнал готовности к прерыванию. После этого внешнее устройство выставляет нашине данных номер вектора прерывания.

Процессор считывает номер и вызывает соответствующую процедуру из таблицы. При этом вызов обработчика прерывания отличается от вызова обычной процедуры: при обычном вызове в стеке сохраняется только адрес команды, на которую следует возвратить управление. При прерывании же в стеке сохраняются два значения: адреса команды и слова состояния процессора. Новое слово состояния берется из таблицы векторов.

При этом приоритет процессора автоматически устанавливается равным тому значению, которое разработчик программы обработки считает правильным. Обратите внимание: не равным приоритету обрабатываемого прерывания, а тому, которое требует разработчик.

При завершении процедуры обработки вызывается команда RTI (Return from Interrupt — возврат из прерывания). Эта команда выталкивает из стека адрес прерванной команды и старое слово состояния, тем самым продолжая исполнение прерванной программы и восстанавливая приоритет процессора. [Киевчев/Некрасов 1988].

Для сравнения: в процессорах семейства i80x86 вектор прерывания содержит только адрес программы-обработчика, а приоритет процессора задается не словом состояния процессора, а регистром внешнего устройства — контроллером прерываний. Контроллер прерываний обычно устанавливает приоритет равным приоритету прерывания, обрабатываемого в данный момент. Чтобы повысить или понизить этот уровень, обработчик прерывания должен программировать контроллер. Перед завершением обработчика необходимо вернуть контроллер прерываний в исходное состояние, выполнив над ним серию магических команд — эпилог прерывания.

Обработка прерываний в системах с виртуальной памятью несколько усложняется: ведь кроме адреса обработчика нам надо еще задать адресное пространство, в котором этот адрес определен. В моделях PDP-11, имеющих диспетчер памяти, эта проблема решается просто: для процессора в каждый момент времени заданы два адресных пространства: пользовательское и системное. Все прерывания обрабатываются в системном адресном пространстве. Для реализации этого процессор имеет два набора регистров диспетчера памяти. Их наличие, с одной стороны, снимает с обработчика прерывания обязанность переключать адресные пространства, а с другой позволяет ядру при обработке системных вызовов обращаться к сегменту данных пользовательского процесса.

В защищенном режиме процессоров i80x86 использован более гибкий механизм установки адресного пространства для обработчика. По существу, с каж-

дым обработчиком может быть ассоциировано свое виртуальное адресное пространство. О способе, которым это достигается, лучше прочитать в литературе по соответствующим процессорам, например [Паппас/Марри 1993].

Прерывания лишены недостатков, которые мы указали ранее для обработки событий с помощью опроса: ожидая события, процессор может заниматься какой-либо другой полезной работой, а когда событие произойдет, он приступит к обработке, не дожидаясь полного завершения этой работы.

Однако этот механизм имеет и собственные недостатки. В частности, обработка прерывания сопряжена с гораздо большими накладными расходами, чем проверка флага и условный переход в режиме ожидания. У оптимизированных для обработки событий микроконтроллеров разница невелика или даже может быть в пользу механизма прерываний: приведенный в примере 6.1 цикл опроса занимает 5 циклов процессора, а обработчик прерывания у PIC вызывается в течение 3-4 циклов ([www.microchip.com PICMicro] утверждает, что средняя задержка прерывания составляет 3,75 цикла). Таким образом, среднее время реакции на событие в режиме опроса составляет 2,5 цикла (по среднему времени опрос в выигрыше), а максимальное — 5 циклов (в данном случае преимущество на стороне прерываний).

Однако у процессоров общего назначения, которые при обработке прерывания вынуждены сохранять несколько регистров и осуществлять относительно сложный диалог с вызвавшим прерывание устройством, задержка между установкой сигнала прерывания и исполнением первой команды его обработчика — этот интервал и называется *задержкой прерывания* (*interrupt latency*) — составляет десятки тактов.

Современные суперскалярные процессоры при обработке прерываний вынуждены сбрасывать очередь предварительной выборки команд и по крайней мере часть кэшей команд и данных, поэтому у них накладные расходы еще больше. Задержка прерывания у современных реализаций архитектуры x86 лишь ненамного лучше, чем у 80386, хотя по скорости исполнения последовательных программ современные процессоры превосходят 80386 на несколько порядков. Поэтому младшие модели процессоров с архитектурой x86, 8086 и даже 8085, хотя и не находят применения в персональных компьютерах, но продолжают выпускаться для использования во встраиваемых приложениях или в качестве периферийных процессоров.

Так, например, "марсоход" Sojourner использовал в качестве управляющего процессора 8085 на сапфировой подложке (для обеспечения радиационной устойчивости).

Это же обстоятельство является дополнительным доводом в пользу включения в систему канальных процессоров, в данном случае с целью освобождения центрального процессора не от опроса, а от обработки прерываний.

6.4. Исключения

Многие процессоры используют механизм, родственный прерываниям, для обработки не только внешних, но и внутренних событий: мы с вами уже сталкивались с *исключительными ситуациями* (exception) отсутствия страницы и ошибки доступа в процессорах с виртуальной памятью, а также некоторыми другими — ошибкой шины при доступе к невыровненным словам, заполнению и очистке регистра окна у SPARC и т. д. Большинство современных процессоров предоставляют *исключения* при неизвестном коде операции, делении на ноль, арифметическом переполнении или, например, выходе значения операнда за допустимый диапазон в таких операциях, как вычисление логарифма, квадратного корня или арксинуса.

Исключительные ситуации обрабатываются аналогично внешним прерываниям: исполнение программы останавливается, и управление передается на процедуру-обработчик, адрес которой определяется природой исключения.

Отличие состоит в том, что прерывания обрабатываются после завершения текущей команды, а возврат из обработчика приводит к исполнению команды, следующей за прерванной. Исключение же приводит к прекращению исполнения текущей команды (если в процессе исполнения команды мы уже успели создать какие-то побочные эффекты, они отменяются), и сохраненный счетчик команд указывает на прерванную инструкцию. Возврат из обработчика, таким образом, приводит к попытке повторного исполнения операции, вызвавшей исключение.

Благодаря этому, например, обработчик страничного отказа может подкачать с диска содержимое страницы, вызвавшей отказ, перенастроить таблицу дескрипторов и повторно выполнить операцию, которая породила отказ. Обработчик исключения по неопределенному коду операции может использоватьсь для эмуляции расширений системы команд.

Например, при наличии арифметического сопроцессора операции с плавающей точкой исполняются им, а при отсутствии — пакетом эмулирующих подпрограмм. Благодаря этому может обеспечиваться полная бинарная совместимость между старшими (имеющими сопроцессор) и младшими (не имеющими его) моделями одного семейства компьютеров.

Исключения, возникающие при исполнении привилегированных команд в пользовательском режиме, могут использоваться системой виртуальных машин. Работающее в виртуальной машине ядро ОС считает, что исполняется в системном режиме. На самом же деле оно работает в пользовательском режиме, а привилегированные команды (переключения режима процессора, настройка диспетчера памяти, команды ввода/вывода) приводят к вызову СВМ.

При грамотной реализации обработчиков таких исключений их обработка произойдет полностью прозрачно для породившей эти исключения программы. Конечно, "подкачка" страницы с диска или программная эмуляция плавающего умножения займет гораздо больше времени, чем простое обращение к памяти или аппаратно реализованное умножение, но, наверное, потребитель вычислительной системы знал, что делал, когда устанавливал недостаточное количество памяти или приобретал машину без сопроцессора.

6.5. Многопроцессорные архитектуры

В дверь диетической столовой
Вошел дракон семиголовый,
Он хором "Здравствуйте" сказал
И, улыбаясь, заказал:
— Для этой головы, пожалуйста, халвы!
— Для этой пасти — прочис сласти!
— Для этой головки — перловки!
— Для этой глотки — селедки!
— Для этой башки — пирожки!
— Для этой рожки — тоже!
— Для этого личика — два сдобных куличика.
Что еще? Лимонада бутылку,
Семь салфеток, ножик и вилку.

В. Берестов

Как уже говорилось, относительно большие накладные расходы, связанные с обработкой прерываний, нередко делают целесообразным включение в систему дополнительных процессоров. Есть и другие доводы в пользу создания многопроцессорных вычислительных систем.

Одним из доводов является повышение надежности вычислительной системы посредством многократного резервирования. Если один из процессоров многопроцессорной системы отказывает, система может перераспределить загрузку между оставшимися. Для компьютеров первых поколений, у которых наработка аппаратуры процессора на отказ была относительно невелика, повышение надежности таким способом часто оказывалось целесообразным, особенно в приложениях, требовавших круглосуточной доступности.

Примечание

Понятно, что для обеспечения непрерывной доступности недостаточно просто поставить много процессоров, и даже недостаточно уметь своевременно обнаружить отказ и исключить сломавшийся процессор из системы. Необходима также возможность заменить отказавший узел без выключения и без перезагрузки системы, что накладывает весьма жесткие требования и на конструкцию корпуса, и на электрические параметры межмодульных соединений, и, наконец, на программное обеспечение, которое должно обнаружить вновь добавленный процессорный модуль и включить его в конфигурацию системы.

Другим доводом в пользу включения в систему дополнительных процессоров является тот факт, что алгоритмы, используемые для решения многих прикладных задач, нередко поддаются *распараллеливанию*: разделению работы между несколькими более или менее независимо работающими процессорами. В зависимости от алгоритма (и, косвенно, от природы решаемой задачи) уровень достижимого параллелизма может сильно различаться. Отношение производительности системы к количеству процессоров и производительности однопроцессорной машины называют *коэффициентом масштабирования*. Для различных задач, алгоритмов, ОС и аппаратных архитектур этот коэффициент различен, но всегда меньше единицы и всегда убывает по мере увеличения количества процессоров.

Некоторые задачи, например, построение фотoreалистичных изображений методом трассировки лучей, взлом шифров полным перебором пространства ключей [www.distributed.net] или поиск внеземных цивилизаций [www.seti.org] поддаются масштабированию очень хорошо: можно включить в работу десятки и сотни тысяч процессоров, передавая при этом между ними относительно малые объемы данных. В этих случаях часто оказывается целесообразно даже не устанавливать процессоры в одну машину, а использовать множество отдельных компьютеров, соединенных относительно низкоскоростными каналами передачи данных. Это позволяет задействовать процессоры, подключенные к сети (например, к Интернету) и не занятые в данный момент другой полезной работой.

Другие задачи, например, работа с базами данных, поддаются распараллеливанию в гораздо меньшей степени, однако и в этом случае обработка запросов может быть распределена между несколькими параллельно работающими процессорами. В разделе *введения* "Системы разделенного времени" рассматривался сценарий, при котором включение дополнительного процессора в систему может увеличить количество запросов, обрабатываемых в секунду. В современных серверах СУБД и приложений обычно устанавливается несколько процессоров, которые подключены к общейшине, совместно используют одну и ту же оперативную память и внешние устройства.

В последние годы возник интерес к так называемым гипертрединговым процессорам (*hyperthreading*, это слово еще не имеет устоявшегося русскоязычного эквивалента). Гипертрединговый процессор представляет собой вариант суперскалярного процессора с несколькими арифметико-логическими устройствами. Как и обычный суперскалярный процессор, такое устройство способно исполнять несколько команд за такт. Однако у простого суперскалярного устройства все эти команды принадлежат к одному потоку команд, и для них создается иллюзия последовательного исполнения (параллельно выполняются только команды, не имеющие зависимостей по данным). Напротив, гипертрединговый процессор может одновременно исполнять несколько по-

токов команд. С точки зрения ОС такое устройство выглядит как два или несколько процессоров.

Некоторые гипертрединговые процессоры не пытаются совместить во времени команды, относящиеся к разным потокам. Так, процессорные ядра в микросхемах UltraSPARC T1 вообще не являются суперскалярными. Они могут исполнять только одну команду за такт, поэтому выполнение различных потоков на таком ядре происходит квазипараллельно, а не параллельно в строгом смысле этого слова. Тем не менее такие устройства обеспечивают измеримый (и достигающий на некоторых задачах десятков процентов) выигрыш по сравнению с аналогичными однопоточными процессорами за счет того, что позволяют занять процессорное ядро полезной работой при промахах кэша.

Гипертрединговые процессоры лишь ненамного превосходят по стоимости изготовления свои однопоточные аналоги и полностью совместимы с ними по разъемам. Первой с такими устройствами вышла на рынок компания Intel, сначала реализовав такую возможность в серверных процессорах линии Xeon, а затем и в ориентированных на рабочие станции Pentium IV. В настоящее время многие производители суперскалярных процессоров, в том числе Sun и AMD, предлагают гипертрединговые устройства.

Главный смысл гипертрединга заключается в том, что даже тщательно оптимизированная для суперскалярного процессора программа лишь изредка способна использовать все арифметико-логические устройства. Кроме того, при промахах кэша процессор вообще вынужден простоять. Разумеется, параллельно исполняемые нити конкурируют и за АЛУ, и за доступ к кэшам и памяти, поэтому гипертрединговый процессор оказывается медленнее настоящей двухпроцессорной системы. Насколько именно медленнее — зависит от задачи и даже от компилятора, или, точнее, от применявшихся этим компиляторов суперскалярных оптимизаций, т. е. от того, сколько АЛУ каждая из задач способна реально использовать. Однако, как уже отмечалось, гипертрединговый процессор может использоваться в системах, рассчитанных на один процессор, поэтому общая стоимость гипертрединговой системы намного ниже, чем полноценной двухпроцессорной.

Впрочем, практическое применение гипертрединга натолкнулось на проблемы. Так, в конце 2005 года появились сообщения, что включение гипертрединга на серверах MS SQL с процессорами Intel Xeon приводит к значительному, порой полутора-двукратному падению производительности [Goodwin 2005]. Аналогичные явления наблюдались и при запуске на гипертрединговых машинах других серверных приложений. В настоящее время трудно сказать, является ли это свидетельством ошибок в процессорах Intel или принципиальным недостатком гипертрединга как технологии. Возможно, дело в том, что серверные приложения не видят разницы между настоящей двух-

процессорной системой и гипертрединговой, и применяют какие-то эвристики (скорее всего, просто создают большое количество дополнительных процессов и нитей, что приводит к росту накладных расходов на синхронизацию и промыванию кэш-памяти), которые оправданы на двухпроцессорной машине, но не подходят в данной ситуации. Если это так, то, возможно, для получения преимуществ новой технологии нужна если не переделка, то, во всяком случае, перенастройка серверных приложений и ядер ОС.

Еще одна новая тенденция — это многоядерные процессоры. Концептуально многоядерный процессор гораздо проще гипертредингового — это просто несколько процессорных ядер, реализованных на одном кристалле (при этом каждое из ядер может быть гипертрединговым). В некоторых реализациях — например, в многоядерных процессорах Intel, — эти ядра имеют общий кэш и интерфейс системной шины (благодаря чему можно обеспечить совместимость по разъему с обычным одноядерным процессором), в других — например, в реализациях AMD — у каждого процессора свой собственный кэш и интерфейс, так что общее количество "ног" у кристалла оказывается таким же, как у соответствующего количества одноядерных устройств. При обоих подходах достигается значительное снижение общей стоимости системы по сравнению с процессорами в отдельных корпусах, хотя стоимость самого кристалла обычно пропорциональна количеству ядер.

В 2004 году компания Sun продемонстрировала восьмиядерный процессор Ultrasparc III T1 (CoolThread), в котором каждое ядро обеспечивает четырехпоточный гипертрединг. Операционная система воспринимает такое устройство как тридцать два процессора. Микросхема имеет четыре интерфейса системной шины, управляющие четырьмя параллельно работающими банками ОЗУ. Процессоры соединены с этими интерфейсами коммутируемой магистралью.

Многопроцессорность обычно применяется только для повышения производительности, но очевидно, что ее же можно использовать и для повышения надежности: когда функционируют все процессоры, система работает быстро, а с частью процессоров работает хоть что-то, пусть и медленнее.

Некоторые многопроцессорные системы поддерживают исполнение на разных процессорах различных ОС — так, на IBM z90 часть процессоров может исполнять Linux, а остальные — z/OS. В такой конфигурации работающий под управлением Linux Web-сервер может взаимодействовать с работающим под z/OS сервером транзакций через общую физическую память. Многопроцессорные серверы Sun Fire могут исполнять несколько копий Solaris.

Современные суперкомпьютеры (IBM SP6000, Cray Origin), используемые для таких задач, как численное решение эллиптических дифференциальных уравнений и численное же моделирование методом конечных элементов в

геофизических, метеорологических и некоторых других приложениях, состоят из десятков, сотен, а иногда и тысяч отдельных процессорных модулей. Каждый модуль представляет собой относительно самостоятельную вычислительную систему, обычно многопроцессорную, с собственной памятью и, нередко, с собственной дисковой подсистемой. Эти модули соединены между собой высокоскоростными каналами. Именно к этому типу относился шахматный суперкомпьютер Deep Blue, выигравший в 1997 году матч у чемпиона мира по шахматам Гарри Каспарова [www.research.ibm.com].

Многопроцессорные системы различного рода получают все более и более широкое распространение. Если производительность отдельного процессора удваивается в среднем каждые полтора года ("закон Мура" [www.intel.com Moore]), то производительность многопроцессорных систем удваивается каждые десять месяцев [www.sun.com 2001-05]. Однако нельзя не отметить, что интерес к многопроцессорным системам, особенно к массивно параллельным, носит волнобразный характер. Нарастание такого интереса обычно происходит в те периоды, когда замедляется рост тактовых частот процессоров. Действительно, первый на моей памяти пик моды на многопроцессорные и вообще параллельные компьютеры возник в первой половине 90-х годов XX века, когда доминировало представление, что кремниевые устройства не могут работать на тактовых частотах, существенно превосходящих 66 МГц. Считалось, что более высокие тактовые частоты требуют использования других полупроводников (например, германия или арсенида галлия), т. е. полной перестройки всего технологического процесса производства микросхем. Когда этот барьер был успешно преодолен за счет использования КМОП-логики с малым размером транзисторов и пониженным напряжением питания, интерес к многопроцессорным машинам довольно быстро угас.

Новая волна моды на "многоголовые" вычислительные системы поднялась в середине первого десятилетия XXI века, когда лидер гонки частот — Intel — неожиданно остановился на частоте 3.4 ГГц. Преобладающая точка зрения состоит в том, что переход этого барьера может потребовать значительной или даже полной перестройки производственных процессов и/или подходов к проектированию микросхем, поэтому кратко- и даже среднесрочные перспективы повышения производительности снова, как и десять лет назад, связывают с увеличением параллелизма.

На практике даже хорошо распараллеливаемые алгоритмы практически никогда не обеспечивают линейного роста производительности с ростом числа процессоров. Это обусловлено, прежде всего, расходами вычислительных ресурсов на обмен информацией между параллельно исполняемыми потоками. На первый взгляд, проще всего осуществляется такой обмен в системах с процессорами, имеющими общую память, т. е. собственно многопроцессорных компьютерах.

В действительности, оперативная память имеет конечную и небольшую по сравнению с циклом центрального процессора, скорость доступа. Даже один современный процессор легко может занять все циклы доступа ОЗУ, а несколько процессоров будут непроизводительно тратить время, ожидая доступа к памяти. *Многопортовое ОЗУ* могло бы решить эту проблему, но такая память намного дороже обычной, однопортовой, и применяется лишь в особых случаях и в небольших объемах.

Одно из основных решений, позволяющих согласовать скорости ЦПУ и ОЗУ, — это снабжение процессоров высокоскоростными *кэшами команд и данных*. Такие кэши нередко делают не только для центральных процессоров, но и для адаптеров шин внешних устройств. Это значительно уменьшает количество обращений к ОЗУ, однако мешает решению задачи, ради которой мы и объединяли процессоры в единую систему: обмена данными между потоками, исполняющимися на разных процессорах (рис. 6.2).

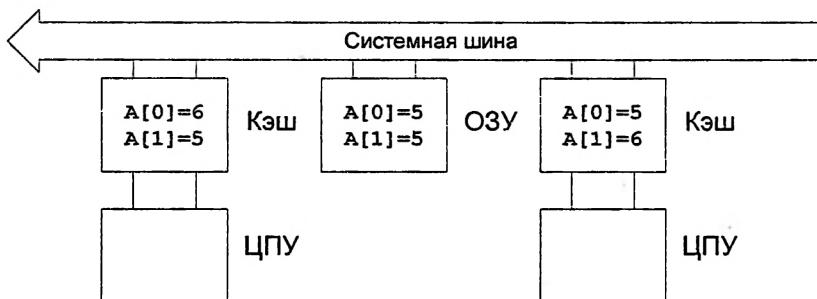


Рис. 6.2. Некогерентный кэш

Большинство контроллеров кэшей современных процессоров предоставляют средства обеспечения *когерентности кэша* (*cache coherency*) — синхронизацию содержимого кэш-памяти нескольких процессоров без обязательной записи данных в основное ОЗУ.

Алгоритм обеспечения когерентности кэша MESI

Рассмотрим алгоритм обеспечения когерентности кэшей, используемый в Pentium II и многих других современных процессорах. Этот алгоритм известен как MESI по первым буквам названий состояний записи в кэше [Paramarcos/Patel 1984]. Эти состояния будут перечислены далее.

Процессоры Pentium II подключены к центральному хранилищу (оперативной памяти, ОЗУ) посредством общей шины. Каждый процессор имеет собственный кэш, объем которого у современных процессоров составляет несколько мегабайт. Записи кэша называются строками (*cache lines*) и имеют объем 64 байта. У более современных процессоров объем строки бывает больше; у Pentium IV строка имеет длину 128 байт. Для простоты далее мы также будем называть соответствующие байты ОЗУ строками.

Для обеспечения когерентности каждый процессор (или, точнее, каждый контроллер кэша) отслеживает все обращения к оперативной памяти со стороны других процессоров и изменяет состояние записи кэша, если он увидел обращение к соответствующей строке ОЗУ. В некоторых случаях процессор рассыпает по шине сообщения другим процессорам, которые приводят к тому, что эти процессоры изменяют состояние соответствующих строк своих кэшей.

Идеологию алгоритма MESI можно описать как продолжение идеологии кэширования: скачивать в кэш и вообще передавать по шине только те записи, которые нужны, и торопиться с выполнением только тех операций, которые кого-то задерживают. Эта идеология оправдана приводившимися в разд. 5.5 рассуждениями про распределение Парето, т. е., попросту говоря, соображением, что большая часть содержимого ОЗУ большую часть времени никому особенно не нужна.

Состояния, используемые алгоритмом MESI, таковы:

- Exclusive (единственная копия) — ни один другой кэш не содержит копий этой строки; содержимое строки совпадает с содержимым ОЗУ, т. е. обе существующие копии строки действительны;
- Modified (модифицирована) — ни один другой кэш не содержит копий этой строки; строка изменена, но изменения еще не попали в ОЗУ, т. е. значение ОЗУ недействительно;
- Shared (разделяемая копия) — один или несколько других кэшей содержат копии этой строки; все копии, как в ОЗУ, так и в кэшах, действительны;
- Invalid (недействительна) — строка ОЗУ содержит устаревшие данные.

Когда процессор 1 в первый раз считывает некоторую строку памяти, он помечает ее как Exclusive и начинает отслеживать все обращения к этой строке со стороны остальных процессоров. Поскольку емкость кэша намного меньше емкости основного ОЗУ, а емкость ассоциативной памяти кэш-контроллера намного меньше адресного пространства процессора, то кэш-контроллер не в состоянии отслеживать обращения ко всем строкам памяти; ему просто негде было бы хранить информацию о таких обращениях.

Впрочем, этого и не требуется: когда процессор 1 видит, что процессор 2 считал ту же строку, он переводит свою копию строки в состояние Shared и выставляет на шине сигнал, что он имеет копию строки, и адрес этой строки. Процессор 2 видит этот сигнал и также переводит свою копию строки в состояние Shared. У Pentium II сигналы передаются по системнойшине и занимают циклы этой шины, хотя и урезанные (обращения к ОЗУ в этих циклах не происходит). Некоторые другие процессоры, такие как DEC/Compaq Alpha и AMD Athlon, используют для обмена сигналами когерентности кэша отдельную шину, параллельную системной (эта технология была запатентована DEC и приобретена AMD при распродаже имущества DEC).

Когда процессор 2 изменяет значение строки, которая была в состоянии Shared, в своем кэше, он рассыпает по шине сигнал, что строка модифицирована. При этом процессор 2 переводит свою копию строки в состояние Modified, а остальные процессоры — в состояние Invalid.

Немедленной записи в ОЗУ не происходит; действительно, ведь если строка процессору 1 не понадобится или понадобится лишь через некоторое время, такая запись привела бы к непроизводительному расходованию циклов шины.

Тем не менее процессор 1 сохраняет запись о строке в своем кэше и отслеживает ее дальнейшую судьбу.

Если процессор 1 попытается обратиться к строке кэша, которая находится в состоянии Invalid, кэш-контроллер заблокирует эту операцию и позволит процессору продолжить работу, только когда процессор 2 обновит данные в ОЗУ. Некоторые вариации алгоритма предусматривают специальный запрос (требование немедленно произвести запись), который процессор 1 в этой ситуации рассыпает по шине.

Когда процессор 1, отслеживая работу шины, видит операцию записи в строку, которая у него была помечена, как Invalid, он уничтожает соответствующую запись в своем кэше. Если процессору позднее вновь понадобится эта строка, он считает ее, как считал бы при обычном промахе кэша.

Самые интересные события происходят, когда процессор 3 попытается обратиться к строке в состоянии Invalid, при условии, что он не имел записи в кэше об этой строке и не отслеживал ее состояния. Процессор 2 в этом случае обязан оповестить процессор 3 о том, что копия записи в ОЗУ недействительна; в этом случае процессор 3 вынужден перевести строку в состояние Invalid, пока процессор 2 не произведет запись.

Суперскалярные процессоры, у которых порядок реального исполнения операций может не совпадать с порядком, в котором соответствующие команды следуют в программе, дополнительно усугубляют проблему.

Порядок доступа к памяти в SPARC

Современные процессоры предоставляют возможность управлять порядком доступа команд к памяти. Например, у микропроцессоров SPARCv9 [www.sparc.com v9] определены три режима работы с памятью (модели памяти), переключаемые битами в статусном регистре процессора.

Свободный доступ к памяти (Relaxed Memory Order, RMO), когда процессор использует все средства кэширования и динамического переупорядочения команд, и не пытается обеспечить никаких требований к упорядоченности выборки и сохранению операндов в основной памяти.

Частично упорядоченный доступ (Partial Store Order, PSO), когда процессор по-прежнему использует и кэширование, и переупорядочивание, но в потоке команд могут встречаться команды MEMBAR. Встретив такую команду, процессор обязан гарантировать, что все операции чтения и записи из памяти, закодированные до этой команды, будут исполнены (в данном случае под исполнением подразумевается перенос результатов всех операций из кэша в ОЗУ) до того, как процессор попытается выполнить любую из операций доступа к памяти, следующих за MEMBAR.

Полностью упорядоченный доступ (Total Store Order, TSO), когда процессор гарантирует, что операции доступа к памяти будут обращаться к основному ОЗУ точно в том порядке, в котором они закодированы.

Каждый следующий режим повышает уверенность программиста, что его программа прочитает из памяти именно то, что туда записал другой процессор, но одновременно приводит и к падению производительности. Наибольший проигрыш обеспечивает наивная реализация режима TSO, когда мы просто выключ-

чаем и динамическое переупорядочение команд, и кэширование данных (кэширование кода можно оставить, если только мы не пытаемся исполнить код, который подвергается параллельной модификации другим задатчиком шины).

Другим узким местом многопроцессорных систем является *системная шина*. Современные компьютеры общего назначения, как правило, имеют *шинную архитектуру*, т. е. и процессоры, и ОЗУ, и адаптеры шин внешних устройств (PCI и т. д.) соединены общей магистралью данных, системной шиной или *системной магистралью*. В каждый момент времени магистраль может занимать только пара устройств, *задатчик* и *ведомый* (рис. 6.3). Обычно задатчиком служит процессор — как центральный, так и канальный, — или контроллер ПДП, а ведомым может быть память или периферийное устройство. При синхронизации содержимого кэшей процессорный модуль также может оказаться в роли ведомого, хотя наиболее распространенные алгоритмы синхронизации кэшей, такие как MESI, этого не предполагают.

Доступ к шине регулируется *арбитром шины*. Практически применяются две основные стратегии арбитража — *приоритетная*, когда устройство, имеющее более высокий приоритет, всегда получает доступ, в том числе и при наличии запросов от низкоприоритетных устройств, и *справедливая* (fair), когда арбитр гарантирует всем устройствам доступ к шине в течение некоторого количества циклов.

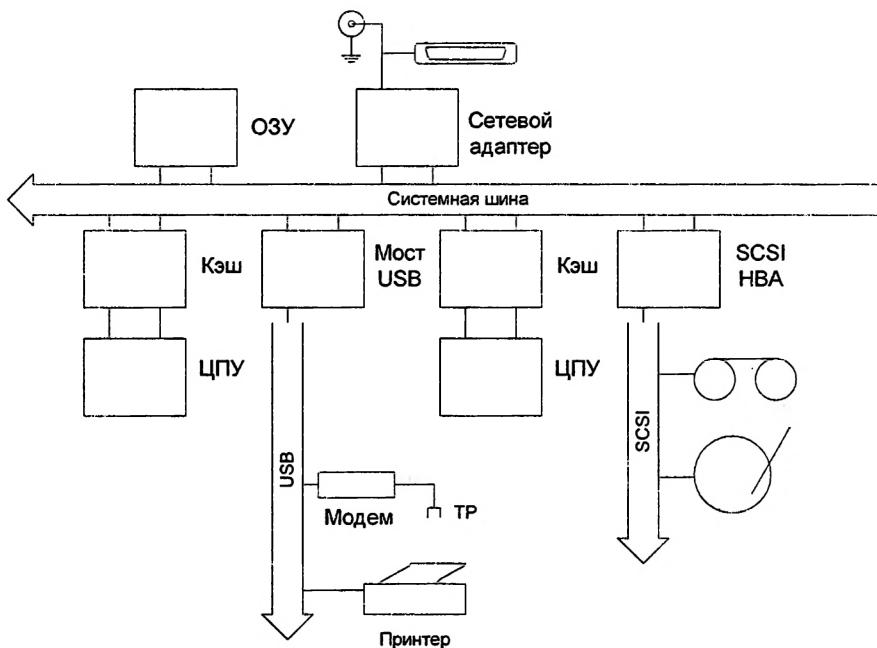


Рис. 6.3. Шинная архитектура

Системы шинной архитектуры просты в проектировании и реализации, к ним легко подключать новые устройства и типы устройств, поэтому такая архитектура получила широкое распространение. Однако, особенно в многопроцессорных системах, шина часто является одним из основных ограничителей производительности. Повышение пропускной способности шины зачастую возможно, но приводит к повышению общей стоимости системы.

Впрочем, при большом количестве узлов проблемы возникают и у систем со столь высокоскоростной шиной, как FireWire. Кроме того, по мере роста физических размеров системы становится необходимо принимать во внимание физическую скорость передачи сигналов — как сигналов самой магистрали, так и запросов к арбитру шины и его ответов. Поэтому шинная топология соединений при многих десятках и сотнях узлов оказывается неприемлема, используются более сложные топологии.

Во многих работах, посвященных таким топологиям, многопроцессорные системы делят на два больших класса:

- UMA (Uniform Memory Access — однородный доступ к памяти), которые обеспечивают для каждого задатчика шины одну и ту же скорость доступа ко всем модулям памяти. Очевидно, что системы шинной архитектуры относятся именно к этому классу;
- NUMA (Non-Uniform Memory Access — неоднородный доступ к памяти), в которых скорость доступа различных задатчиков к разным модулям ОЗУ различна. Обычно такие системы состоят из модулей, каждый из которых имеет собственную шину и содержит несколько процессоров и/или адаптеров периферийных шин и некоторое количество ОЗУ. Скорость доступа к памяти своего модуля намного выше, чем скорость доступа к памяти других модулей.

На практике машины, построенные на многоядерных процессорах с независимыми шинами памяти у разных ядер, также оказываются NUMA-системами. Действительно, если у каждого ядра своя шина памяти и к ней подключены своя кэш-память и свой банк ОЗУ, то для каждого из ядер обращение к "своему" ОЗУ будет происходить быстрее, чем к ОЗУ, принадлежащему другому ядру.

При работе на NUMA-системе операционная система должна учитывать неоднородность памяти при планировании процессов. Действительно, в UMA-системе перенос процесса с одного процессора на другой — относительно дешевая операция, ее стоимость одинакова для любой пары процессоров. Напротив, в NUMA стоимость такого переноса различна для процессоров, принадлежащих к одному модулю или к разным.

Многие ОС для NUMA-систем обеспечивают миграцию страниц ОЗУ, к которым часто обращается модуль, в память этого модуля. Планирование таких

миграций — сложная задача, особенно учитывая, что некоторые страницы используются для межпроцессного взаимодействия и должны были бы находиться в памяти нескольких модулей одновременно.

Системы NUMA-Q

Многопроцессорные серверы IBM NUMA-Q состоят из отдельных процессорных модулей. Каждый модуль имеет собственную оперативную память и четыре процессора x86. Модули называются *quad* (четверки) (рис. 6.4).

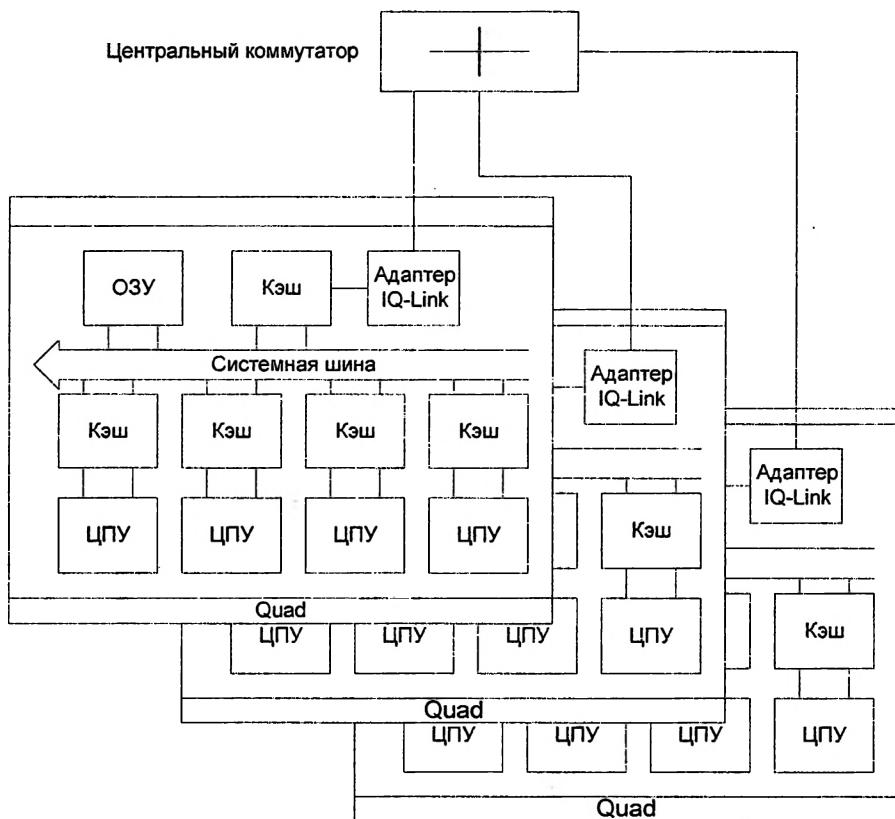


Рис. 6.4. NUMA-Q с тремя четырехпроцессорными модулями

Четверки соединены высокоскоростными каналами IQ-Link с центральным коммутатором. Замена общей шины на звездообразную топологию с центральным коммутатором позволяет решить проблемы арбитража доступа к шине, в частности, устранить задержки при запросе к арбитру шины и ожидании его ответа запрашивающему устройству. NUMA-системы фирмы IBM могут содержать до 16 четверок, т. е. до 64 процессоров.

Архитектура позволяет также включать в эти системы процессоры с системой команд, отличной от x86, например RS/6000 и System/390, позволяя, таким об-

разом, создать в пределах одной машины гетерогенную сеть со сверхвысокоскоростными каналами связи.

При большем числе модулей применяются еще более сложные топологии, например *гиперкубическая*. В таких системах каждый узел обычно тоже содержит несколько процессоров и собственную оперативную память (рис. 6.5).

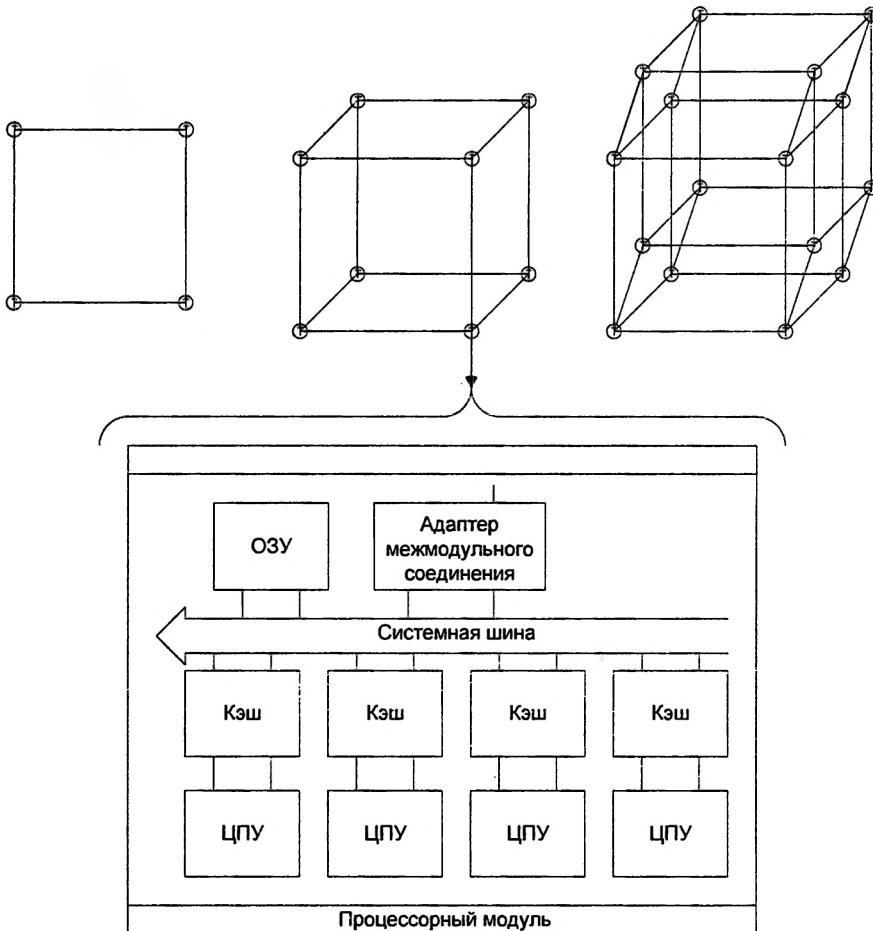


Рис. 6.5. Гиперкубы с 4, 8 и 16 вершинами

При гиперкубическом соединении количество узлов N пропорционально степени двойки, а каждый узел имеет $\log_2 N$ соединений с другими узлами. Каждый узел способен не только обмениваться сообщениями с непосредственными соседями по топологии, но и маршрутизировать сообщения между узлами, не имеющими прямого соединения. Самый длинный путь между

узлами, находящимися в противоположных вершинах куба, имеет длину $\log_2 N$ и не является единственным (рис. 6.6). Благодаря множественности путей, маршрутизаторы могут выбирать для каждого сообщения наименее загруженный в данный момент путь или обходить отказавшие узлы.

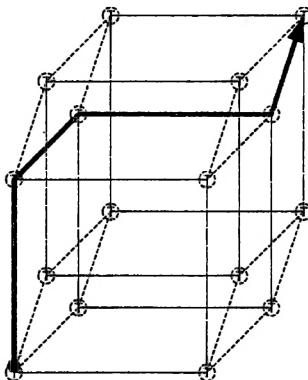


Рис. 6.6. Самый длинный путь в гиперкубе

Массивно параллельные системы Cray/SGI Origin

Узлы суперкомпьютеров семейства Cray/SGI Origin соединены в гиперкуб каналами с пропускной способностью 1 Гбайт/с. АдAPTERЫ соединений обеспечивают не просто обмен данными, а прозрачный (хотя и с падением производительности) доступ процессоров каждого из узлов к оперативной памяти других узлов и обеспечение когерентности процессорных кэшей.

Различие в скорости доступа к локальной памяти процессорного модуля и других модулей является проблемой и при неудачном распределении загрузки между модулями (таком, что межмодульные обращения будут часты) приведет к значительному снижению производительности системы. Известны два основных пути смягчения этой проблемы:

- **COMA** (Cache Only Memory Architecture) — архитектура памяти, при которой работа с ней происходит как с кэшем. Система переносит страницы памяти, с которыми данный процессорный модуль работает чаще других, в его локальную память;
- **CC-NUMA** (Cache-Coherent Non-Uniform Memory Access, неоднородный доступ к памяти с обеспечением когерентности кэшей). В этой архитектуре адAPTERЫ межмодульных соединений снабжаются собственной кэш-памятью, которая используется при обращениях к ОЗУ других модулей. Основная деятельность центрального коммутатора и каналов связи состоит в поддержании когерентности этих кэшей [www.ibm.com NUMA-Q].

Понятно, что обе эти архитектуры не решают в корне проблемы неоднородности доступа: для обеих можно построить такую последовательность межпроцессорных взаимодействий, которая промоет все кэши и перегрузит межмодульные связи, а в случае СОМА приведет к постоянной перекачке страниц памяти между модулями. То же самое, впрочем, справедливо и для симметричных многопроцессорных систем с общей шиной.

В качестве резюме можно лишь подчеркнуть, что *масштабируемость* (отношение производительности системы к количеству процессоров) многопроцессорных систем определяется в первую очередь природой задачи и уровнем параллелизма, заложенным в использованный для решения этой задачи алгоритм. Разные типы многопроцессорных систем и разные топологии межпроцессорных соединений пригодны и оптимальны для различных задач.



ГЛАВА 7

Параллелизм с точки зрения программиста

Ты выбежал на угол купить вина.
Ты вернулся, а вместо дома стена.

Б. Гребенников

На палубу вышел, а палубы нет,
В глазах у него помутилось.

В. Пелевин

В предыдущей главе мы видели, что даже в современном однопроцессорном персональном компьютере происходит множество параллельных процессов: звуковая карта играет, жесткий диск и сетевой интерфейс передают данные, пользователь двигает мышью — работа кипит! А что начнется, если пользователь запустит задание на печать, так и просто страшно подумать.

Написание программ, способных работать в среде с множеством параллельно происходящих процессов, представляет собой нетривиальную задачу. На первый взгляд, сложности здесь никакой нет — аппаратура предоставляет нам механизм прерываний. Обработал прерывание — и наступило счастье. В действительности, никакого счастья от одной только обработки прерывания не наступит, пока мы не сообщим о произшедшем событии основному потоку программы, заинтересованной в этом событии.

Основной поток программы и реализуемые этой программой обработчики прерываний должны взаимодействовать и разделять те или иные данные. При этом в обработчике прерывания мы не всегда можем точно выяснить, в какой точке основной поток программы был прерван (в принципе, можно проанализировать сохраненный счетчик команд и, возможно, локальные переменные основного потока, но это очень сложно и само по себе вряд ли приблизит нас к реализации корректно взаимодействующих потоков), а основной поток не

всегда может знать, в какой момент происходило (и происходило ли) прерывание.

Задача разработки программы, взаимодействующей с обработчиком прерывания, таким образом, может быть переформулирована как написание программы, некоторые переменные которой подвержены изменению в непредсказуемые моменты времени.

Большинство практически применяемых структур данных должны соответствовать тем или иным предположениям, *критериям целостности* (*consistency*). Например, в упорядоченном массиве каждый следующий элемент должен быть больше (то, что в данном конкретном случае подразумевается под "больше", называется критерием или условием сортировки) предыдущего или равен ему. Основной способ модификации упорядоченного массива — это вставка в него дополнительного элемента. Вставка в такой массив может быть осуществлена различными способами, например, добавлением нового элемента в конец и выполнением сортировки методом "пузырька" или поиском места, куда элемент должен быть вставлен, и перемещением элементов с большими индексами.

Важно, что любой способ вставки происходит не мгновенно, и все время работы этой процедуры массив не является упорядоченным либо содержит дублирующиеся элементы. Если вставка происходила в основном потоке программы, обработчик прерывания, который в это время попытается работать с массивом, как с упорядоченным — например, произвести в нем дихотомический поиск — будет жестоко разочарован.

Это обстоятельство резко усложняет анализ алгоритмов (в частности, доказательство корректности программ) и доставило в свое время много волнений теоретикам программирования. Например, в [Дейкстра 1978] один из основателей структурного программирования, Э. Дейкстра, очень эмоционально описывает свою реакцию при первом столкновении с системой, использующей прерывания. Кроме теоретических сложностей, разработка таких программ сопряжена и со сложностями практическими.

При разработке параллельной программы мы можем неявно сделать и использовать при кодировании предположение, что состояние некоторого объекта в некоторый период времени не меняется — а оно *может* измениться. Если такая ошибка сделана в последовательно исполняющейся программе, она может быть выявлена при первом же тестовом прогоне. Для выявления же ее в программе с асинхронно исполняющимися модулями потребуется гораздо больше тестовых запусков, при которых мы должны вызывать прерывание в различные моменты времени.

Для исчерпывающего тестирования необходимо перебрать все возможные относительные моменты вызова прерывания, т. е. обеспечить хотя бы раз вы-

зов прерывания после каждой из команд в каждой из возможных последовательностей исполнения основной программы. Стоимость такого тестирования запретительно высока, поэтому ошибки такого рода (в англоязычной литературе они называются *race condition* (дословно — ошибка соревнования), хороший же русский термин мне неизвестен) практически невозможно искоренить в процессе тестирования.

Таким образом, единственный способ избежать ошибок соревнования — это не делать их. Для того чтобы не делать ошибок, нужна формальная методика разработки и кодирования параллельно исполняющихся программ. Понятно, что и наличие методики не может полностью исключить ошибки. Однако, если выработанная методика адекватна, каждая ошибка будет ее нарушением, поэтому ошибки могут выявляться анализом кода на соответствие формальным требованиям.

К счастью, автору этой книги нет необходимости заниматься разработкой такой методики с чистого листа. Достаточно лишь, по возможности связно, изложить уже изобретенные методы, не утруждая себя и читателя полным доказательством корректности предлагаемых методик и приводя лишь "интуитивное" обоснование их применимости. Сомневающимся могу предложить либо разработать полное доказательство самостоительно, либо обратиться к специальной литературе, например [Хоар 1989].

Приведенная ранее формулировка задачи справедлива не только для взаимодействия основного потока программы с обработчиком прерывания, но и для взаимодействия программ, исполняющихся на различных процессорах, а также для программы, непосредственно взаимодействующей с внешними событиями, например посредством опроса. В разд. 8.2 мы увидим, что [псевдо]параллельные нити исполнения, не являющиеся обработчиками прерываний, довольно легко можно реализовать на однопроцессорной машине, и практически все современные ОС предоставляют такой сервис.

Большинство концепций, обсуждаемых в этой главе, приложимы ко всем перечисленным случаям, поэтому далее в тексте мы будем говорить не о программе и обработчике прерывания, а о двух или более *потоках* или *нитях* исполнения. В действительности, одна из взаимодействующих "нитей" может не быть процессом исполнения программы, а представлять собой физический процесс (например, перемотку ленты, перемещениечитывающей головки дисковода, или химическую или даже ядерную реакцию в установке, которой управляет наш компьютер) или процесс, происходящий в голове или других модулях нервной системы пользователя-человека, но в большинстве случаев нас эта тонкость не интересует.

7.1. Формулировка задачи

Я что-то не вижу пивного ларька.
Должно быть, его
Успели снести за ночь.

Б. Гребенников

Чтобы понять, какие же методики применимы для взаимодействия параллельно исполняющихся нитей, давайте более четко сформулируем задачу. Для этого нам также надо ввести некоторое количество терминов. Кроме того, нам придется высказать ряд соображений, часть которых может показаться читателю банальностями.

Установим, что для каждой из нитей создается иллюзия строго последовательного исполнения (например, обработчик прерывания может создать для основного потока программы такую иллюзию, аккуратно сохраняя при вызове и восстанавливая при завершении все регистры процессора). Если для какой-то из нитей это условие не всегда соблюдается, мы будем считать ее двумя или большим числом различных нитей. Если это условие не соблюдается никогда, мы имеем дело с процессором нефон-неймановской архитектуры.

Общепринятое название для взаимодействия между различными нитями — *асинхронное взаимодействие*, в противоположность *синхронному взаимодействию* между различными модулями последовательно исполняемой программы.

Если нить работает только с объектами (под объектом мы, в данном случае, понимаем не только группу переменных в оперативной памяти или объект в смысле ООП, но и физические объекты, например управляемые компьютером внешние устройства), состояние которых не может быть изменено другими нитями, проблемы взаимодействия, да и самого взаимодействия, как такового, не существует.

Если нить работает с объектами, состояние которых вообще не подвергается модификации, например, с кодом или таблицами констант, проблемы также нет. Проблема возникает тогда и только тогда, когда модификации подвергается объект, разделяемый несколькими нитями. При этом для возникновения проблемы достаточно, чтобы только одна нить занималась модификацией, а остальные нити считывали состояние объекта.

Интервал, в течение которого модификация нарушает целостность разделяемой структуры данных, и, наоборот, интервал, в течение которого алгоритм нити полагается на целостность этой структуры, называется *критической секцией*. Задача написания корректной многопоточной программы, таким образом, может решаться двумя способами: либо искоренением критических

секций из всех используемых алгоритмов, либо обеспечением гарантии того, что никакие две нити никогда одновременно не войдут в критическую секцию, связанную с одним и тем же разделяемым объектом. Наличие в программе критической секции с негарантированным исключением и есть ошибка соревнования, которая рано или поздно сработает.

Полное искоренение критических секций из кода требует глубокой переработки всех алгоритмов, которые используются для работы с разделяемыми данными. Результат такой переработки мы видели в примере 5.2: странная, на первый взгляд, двойная загрузка регистра в настроенной записи PLT в этом примере обусловлена именно желанием избежать проблем при параллельной настройке одной и той же записи двумя разными нитями (в качестве упражнения читателю предлагается разобраться, в каком порядке интерпретатор модифицирует запись, и как выглядит промежуточный код). У меня нет примеров, демонстрирующих невозможность такой переработки в общем случае, но очевидно, что ко многим, даже крайне простым алгоритмам, она практически не применима.

Второй путь — предоставление гарантий *взаимоисключения* (*mutual exclusion*) — также непрост, но, в отличие от первого, практически реализуем и широко применяется.

Примечание

Важно подчеркнуть, что сокращение времени, которое каждая из нитей проводит в критических секциях, хотя и полезно со многих точек зрения, в частности с точки зрения улучшения масштабируемости алгоритма, само по себе проблемы решить не может. Без гарантии взаимоисключения сокращение критического времени лишь понижает вероятность срабатывания ошибки соревнования (и, в частности, затрудняет поиск такой ошибки при тестировании), но не исправляет эту ошибку.

Группа операций модификации разделяемой структуры данных, которая проходит атомарно (неделимо), не прерываясь никакими другими операциями с той же структурой данных, называется *транзакцией*. В разд. 7.3.4 мы увидим более радикальное определение термина "транзакция" как группы операций, которые всегда либо происходят все вместе, либо не происходят вообще, даже если во время попытки их выполнения случится общесистемный сбой. Понятно, что для реализации так понимаемых транзакций одного только взаимоисключения недостаточно.

Программный модуль, внутри которого имеется хотя бы одна критическая секция, для которой не обеспечено взаимное исключение, называется *перенерабельным*. Вызов процедур такого модуля из различных нитей приведет к ошибкам соревнования и допустим лишь при условии, что вызывающая программа реализует взаимное исключение самостоятельно. Соответственно, модуль, в котором таких секций нет, или который сам обеспечивает взаимное

исключение для них, называется *реентерабельным* (от англ. *re-enterable* — способный к повторному вхождению) или *реентрантным* (*reentrant*). В современной англоязычной литературе часто также употребляются термины *thread-unsafe* (для обозначения нереентерабельных процедур) и *thread-safe* (соответственно, для реентерабельных).

Рассмотрим простейший случай разделяемого объекта: флаговую переменную, которая может принимать значения *True* и *False*. Такая переменная, в частности, может использоваться для реализации взаимного исключения в секции, работающей с более сложной структурой данных.

Если в критической секции не находится ни одной нити, переменная равна *False*, иначе — *True*. При входе в секцию нам необходимо проверить значение переменной и, если блокируемый участок свободен, присвоить ей *True*. Данный пример любопытен не только своей простотой, но и тем, что совмещает в себе оба типа критических секций: изменение разделяемых данных и анализ данных, которые могут параллельно модифицироваться кем-то еще.

Наивный способ работы с такой переменной приведен в примере 7.1 (пример реализован на паскалеводобном псевдокоде; операторы *parbegin/parend* символизируют параллельное выполнение заключенных между ними операторов). Казалось бы, трудно представить себе более простую программу, однако именно благодаря простоте легко понять, что она никуда не годится: проверка флага и его установка реализуются двумя различными операторами, в промежутке между которыми другой процесс может получить управление и также установить флаговую переменную! Окно, в котором происходит соревнование, составляет всего две-три команды, но при попадании обоих процессов в это окно мы получаем как раз то, чего стремились избежать: оба процесса могут войти в критическую секцию одновременно.

Пример 7.1. Наивная реализация взаимного исключения на основе флаговой переменной

```
program флаг;
var flag: Boolean;
procedure процессодин;
begin
  while True do
    begin
      while flag do;
      flag := True;
      критическаясекция1;
      flag := False;
```

```
    end;
end;
procedure процессдва;
begin
  while True do
    begin
      while flag do;
      flag := True;
      критическаясекция2;
      flag := False;
      ....
    end
  end;
begin
  flag:= False;
  parbegin
    процессодин;
    процесдва;
  parend
end.
```

7.2. Примитивы взаимоисключений

В классической работе Г. М. Дейтela [Дейтел 1987] предлагается несколько последовательных усовершенствований механизма взаимоисключений, основанного на флаговых переменных, и как завершающий этап этого анализа приводится алгоритм взаимоисключений Деккера (пример 7.2). Идею этого алгоритма можно описать всего одним предложением: каждая из нитей перед входом в критический участок устанавливает свой флаг и проверяет чужой.

Пример 7.2. Алгоритм Деккера (цит. по [Дейтел 1987])

```
program АлгоритмДеккера;
var
  избранныйпроцесс: (первый, второй);
  п1хочетвойти, п2хочетвойти: Boolean;
procedure процессодин;
begin
  while true do
    begin
```

```
п1хочетвойти := True;
while п2хочетвойти do
  if избранныйпроцесс=второй then
    begin
      п1хочетвойти := False;
      while избранныйпроцесс=второй do;
      п1хочетвойти := True;
    end
  критическийучасток1;
  избранныйпроцесс := второй;
  п1хочетвойти := False;
  .....
end
procedure процессдва;
begin
  while true do
    begin
      п2хочетвойти := True;
      while п1хочетвойти do
        if избранныйпроцесс=первый then
          begin
            п2хочетвойти := False;
            while избранныйпроцесс=первый do;
            п2хочетвойти := True;
          end
      критическийучасток2;
      избранныйпроцесс := первый;
      п2хочетвойти := False;
      .....
    end
  end
begin
  п1хочетвойти := False;
  п2хочетвойти := False;
  избранныйпроцесс := первый;
  parbegin
    процессодин;
    процессдва;
  parend
end.
```

Недостатки этого решения очевидны. Первый из них — для доступа к одной и той же критической секции из третьей нити мы должны значительно усложнить код обеих нитей и добавить третью флаговую переменную.

На практике, для решения проблемы работы с флаговыми и другими скалярными переменными в многопроцессорных конфигурациях большинство современных процессоров предоставляют аппаратные *примитивы взаимоисключения*: средства, позволяющие процессору монопольно захватить шину и выполнить несколько операций обращения к памяти. Имея такую возможность, мы могли бы реализовать некоторый атомарный примитив, который в книге [Дейтел 1987] называется *testandset* (проверить и установить). Способ использования такого примитива приводится в примере 7.3. В отличие от алгоритма Деккера, этот код легко распространяется на случай более чем двух нитей.

Пример 7.3. Реализация взаимоисключения посредством атомарной операции *testandset* (цит. по [Дейтел 1987])

```
program примерtestandset
var активный: Boolean;
procedure процессодин;
var первомувходитьнельзя: Boolean;
begin
  while true do
    begin
      первомувходитьнельзя := True;
      while первомувходитьнельзя do
        testandset(первомувходитьнельзя, активный);
      критический участокодин;
      активный := False;
      ....
    end
  end;
procedure процессдва;
var второмувходитьнельзя: Boolean;
begin
  while true do
    begin
      второмувходитьнельзя := True;
      while второмувходитьнельзя do
        testandset(второмувходитьнельзя, активный);
```

```

    критический участокдва;
    активный := False;
    ....
end
end;

```

Реализации этих примитивов различны у разных процессоров. Например, у x86 это команда `xchg` (eXCHanGe, обменять), которая обменивает значения регистра и второго операнда, который обычно размещается в ОЗУ. Эта команда всегда исполняется в режиме монопольного доступа к шине, т. е. процессор аппаратно обеспечивает атомарность её исполнения — в регистре всегда будет находиться именно то значение, которое было в памяти к моменту начала операции, а другие устройства — в том числе и другие процессоры — смогут получить доступ к соответствующей ячейке ОЗУ только тогда, когда операция будет уже завершена.

Самый простой способ использования этой команды для реализации флага взаимоисключения показан в примере 7.4. Видно, что он предполагает ожидание освобождения ресурса в холостом цикле, поэтому в современных ОС такой примитив называют *спинлоком* (*spinlock*, дословно — вращающийся запор). Спинлоки обычно используются в ядрах ОС для разделения доступа к ресурсам между нитями ядра, исполняющимися на разных процессорах.

Пример 7.4. Реализация примитива `testandset` через блокировку шины

```

.data
; 0 = False, 1 = True
flag: .int $0
.text
...
tryagain:
    movel $1, %eax
    xchgl %eax, flag
    tstl %eax
    bnz tryagain

    критическая секция
    ....
; в данном случае о взаимоисключении можно не заботиться
    xorl %eax, %eax
    movel %eax, flag

```

Кроме исполнения в режиме монопольного захвата шины, команда `xchg` имеет еще одну интересную особенность — в суперскалярных реализациях архитектуры x86 она работает аналогично команде `MEMBAR` в ослабленном режиме работы с памятью процессоров SPARC. При ее исполнении процессор гарантирует, что исполнит все встреченные в коде раньше `xchg` обращения к памяти до исполнения самой `xchg` и не начнет до ее исполнения ни одной из операций, которые были закодированы после нее. Видно, что эта особенность также очень полезна при использовании команды `xchg` в качестве примитива взаимоисключения в многопроцессорных системах, но из-за нее же эту команду не следует использовать при обычном программировании, ведь она нарушает конвейеризацию и попытки процессора распараллелить исполнение программы.

Другие процессоры, например VAX, предоставляют более сложные команды, исполняющиеся в режиме монопольного захвата шины. В частности, именно так этот процессор исполняет команды вставки в очередь и исключения из нее.

Имея возможность произвести атомарно исполняющуюся операцию над скалярной переменной, мы можем реализовать более сложные схемы взаимоисключения, используя эту переменную в качестве флаговой, с помощью относительно простого кода (пример 7.4).

В том случае, если одной из нитей является обработчик прерывания, можно воспользоваться сервисом, который предоставляют все современные процессоры: запретить прерывания на время нахождения основной нити программы в критической секции. Впрочем, указанным способом необходимо пользоваться с осторожностью — это приводит к увеличению времени обработки прерывания, что не всегда допустимо, особенно в задачах реального времени.

7.2.1. Мертвые и живые блокировки

Потом ударили морозы.
Замерзло все, лиса ушла в кредит.
Медведь же вмерз в дупло
И до сих пор глядит.

Б. Гребенников

Решив проблему взаимоисключения для одиночных разделяемых ресурсов, мы еще не можем расслабляться. Дело в том, что если мы используем любые механизмы взаимоисключения для доступа к нескольким различным ресурсам, может возникнуть специфическая проблема, называемая *мертвой блокировкой* (*dead lock*).

Рассмотрим две нити, каждая из которых работает с двумя различными ресурсами одновременно. Например, одна нить копирует данные со стриммера

на кассету Exabyte, а другая — в обратном направлении. Доступ к стриммеру контролируется флаговой переменной flag1, а к кассете — flag2 (вместо флаговых переменных могут использоваться и более сложные средства взаимоисключений).

Первая нить сначала устанавливает flag1, затем flag2, вторая поступает наоборот. Поэтому, если вторая нить получит управление и защелкнет flag2 в промежутке между соответствующими операциями первой нити, мы получим мертвую блокировку (рис. 7.1) — первая нить никогда не освободит flag1, потому что стоит в очереди у переменной flag2, занятой второй нитью, которая стоит в очереди у flag1, занятой первой.

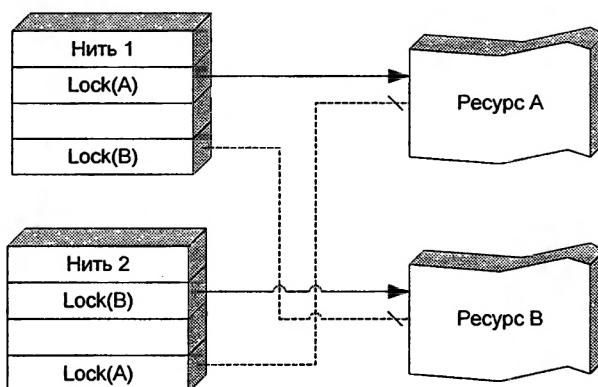


Рис. 7.1. Мертвая блокировка

Все остальные нити, пытающиеся получить доступ к стриммеру или кассете, также будут становиться в соответствующие очереди и ждать, пока администратор не снимет одну из "зашелкнувшихся" задач.

Цикл взаимного ожидания может состоять и из большего количества нитей. Возможна также мертвая блокировка с участием только одной нити и одного ресурса: для этого достаточно попытаться захватить одну и ту же флаговую переменную два раза. Критерием блокировки является образование замкнутого цикла в графе ожидающих друг друга задач.

Эта проблема может быть решена несколькими способами. Часто применяемое решение, обладающее, впрочем, серьезными недостатками — это отправка программе сообщения о том, что попытка установить примитив взаимоисключения приведет к мертвой блокировке. Это решение опасно, во-первых, тем, что сильно усложняет кодирование: теперь мы вынуждены принимать во внимание не только возможность захвата примитива другой нитью, но и более сложные ситуации. Во-вторых, получив ошибку установки флага,

программист испытывает сильный соблазн сделать именно то, чего делать в данном случае нельзя: повторить попытку захвата ресурса.

Рассмотрим ситуацию, когда две нити пытаются захватить необходимые им ресурсы, получают сообщение о возможности мертвой блокировки и тут же повторяют попытку захвата того же ресурса. Поскольку освобождения ресурсов не происходит, взаимозависимость между этими нитями не устраниется, и повторный захват также приводит к сообщению о возможности мертвой блокировки. Если нити будут циклически повторять попытки захвата, мы получим состояние, которое называется *живой блокировкой* (*livelock*) (рис. 7.2). Это состояние реже рассматривается в учебниках, но теоретически оно ничуть не лучше мертвой блокировки. Практически же оно гораздо хуже — если нити, зацепившиеся намертво, тихо висят и причиняют вред только тем нитям, которым могли бы понадобиться занятые ими ресурсы, то нити, зацепившиеся заживо, непродуктивно расходуют время центрального процессора.

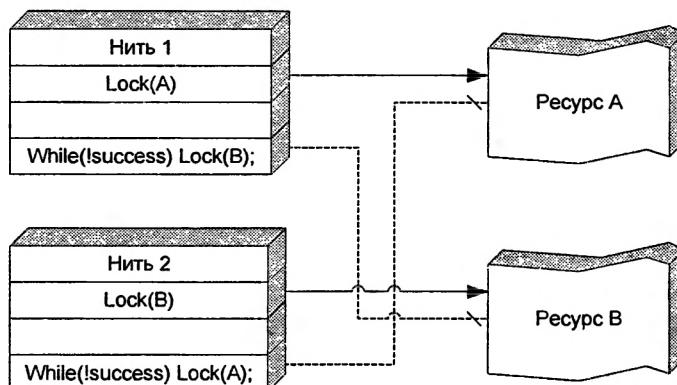


Рис. 7.2. Живая блокировка

Как живая, так и мертвая блокировки возможны и в ситуации, когда ресурс только один, но примитив взаимоисключения не атомарен, т. е. операция захвата выполняется в несколько этапов.

Живая блокировка при арбитраже шины

Рассмотрим процесс арбитража шины: два устройства соревнуются за доступ к среде передачи. Один из алгоритмов арбитража, известный как CSMA/CD (Carrier Sense Multiple Access/Collision Detection, множественный доступ с прослушиванием несущей и обнаружением коллизий), состоит в следующем.

Устройство начинает передачу и при этом сравнивает передаваемый сигнал с принимаемым из шины. Возникновение расхождений между этими сигналами интерпретируется как коллизия (*collision*) — предполагается, что расхождения обусловлены работой другого передатчика. Обнаружив коллизию, устройство прекращает передачу. Сигнал распространяется по шине с конечной скоростью,

поэтому прекратить передачу будут вынуждены оба устройства. Таким образом, оба устройства не могут начать передачу сразу же после того, как в шине "установится тишина": это приведет к живой блокировке. Поэтому реализация CSMA/CD, используемая в протоколах локальных сетей семейства Ethernet, требует от устройства, обнаружившего коллизию, ожидания в течение случайного интервала времени [Иртегов 2004].

Единственно правильная реакция на получение сообщения о мертвой блокировке — это освободить все занятые нитью ресурсы и подождать в течение случайного промежутка времени. Таким образом, поиск возможных блокировок сильно усложняет и логику работы самих примитивов взаимоисключений (нужно поддерживать граф, описывающий, кто кого ждет), и код, использующий эти примитивы.

Простейшее альтернативное решение — разрешить каждой нити в каждый момент времени держать захваченным только один объект — прост и решает проблему в корне, но часто оказывается неприемлемым. Больше подходит соглашение о том, что захваты ресурсов должны всегда происходить в определенном порядке. Этот порядок может быть любым, важно только, чтобы он всегда соблюдался.

Еще один вариант решения состоит в предоставлении возможности объединять примитивы и/или операции над ними в группы. При этом программа может выполнить операцию захвата флагов flag1 и flag2 единой командой, во время исполнения которой никакая другая программа не может получить доступ к этим переменным. Групповая операция выполняется как транзакция, т. е. либо происходит вся целиком, либо не происходит вообще. Если хотя бы одна из операций группы приводит к ожиданию, групповой примитив снимает все флаги, которые успел поставить до этого.

Все три перечисленных решения — и упорядочение доступа к ресурсам, и освобождение всех ресурсов при сообщении о мертвой блокировке, и даже атомарные блоки операций — обладают одним общим недостатком, который крайне затрудняет их практическое применение. А именно, при каждой попытке захвата ресурса код, который это делает, должен знать обо всех остальных ресурсах, которые могут понадобиться нити за то время, в течение которого он намерен удерживать блокировку. Трудно придумать требование, столь же несовместимое с принципами модульного программирования и практическими требованиями к повторно используемому коду.

Ожидание с освобождением ресурсов порождает также некоторые более специфические проблемы. Рассмотрим одну из них: нити A нужны ресурсы X и Y, которые она разделяет, соответственно, с нитями B и C. Если нить A захватывает ресурсы в соответствии с предложенным протоколом (освобождая их при неудачной попытке захвата), возможен такой сценарий исполнения нитей B и C, при котором нить A не получит доступа к ресурсам никогда.

Напротив, захват ресурсов по мере их освобождения другими нитями может (если В и С сцеплены по каким-то другим ресурсам) привести к мертвой блокировке. Общего решения задачи разрешения блокировок и родственных им ситуаций при взаимоисключении доступа ко многим ресурсам на сегодня не предложено.

Описанное явление иногда называют "*проблемой голодного философа*". История этого колоритного названия восходит к модели, которую использовал для демонстрации указанного феномена Э. Дейкстра в книге [Дейкстра 1978].

Некоторое количество (у Э. Дейкстры — пять) философов сидят вокруг стола, на котором стоит блюдо спагетти (вместо спагетти можно взять, например, блины). Спагетти едят двумя вилками. В нашем случае количество вилок равно количеству философов, так что соседи за столом разделяют вилку (гигиенический аспект проблемы мы опускаем).

Каждый философ некоторый (переменный) промежуток времени размышляет, затем пытается взять лежащие рядом с ним вилки (рис. 7.3). Если ему это удается, он принимается за еду. Ест он также переменный промежуток времени, после этого откладывает вилки и вновь погружается в размышления. Проблемы возникают, когда философ не может взять одну из вилок.

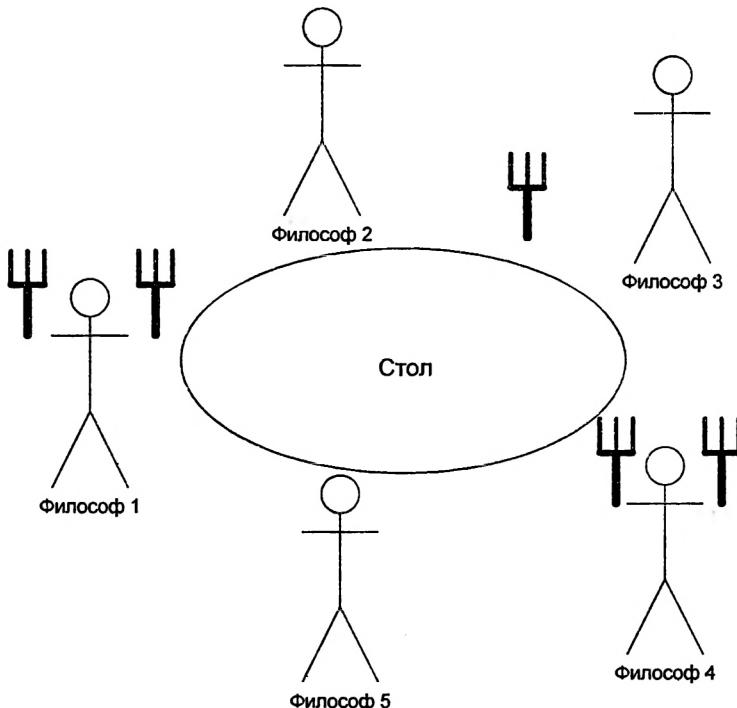


Рис. 7.3. Обедающие философы

Возможны несколько вариантов разрешения происходящих при этом коллизий. Если, например, каждый из философов всегда берет ту вилку, которая свободна, и не отпускает ее, пока не насытится, мы можем получить (хотя и не обязательно получим) классическую мертвую блокировку, в которую будут включены все философы, сидящие вокруг стола (рис. 7.4). Вариант, при котором философ сначала всегда дожидается правой вилки, и лишь взяв ее, начинает дожидаться левой, только повысит вероятность образования цикла.

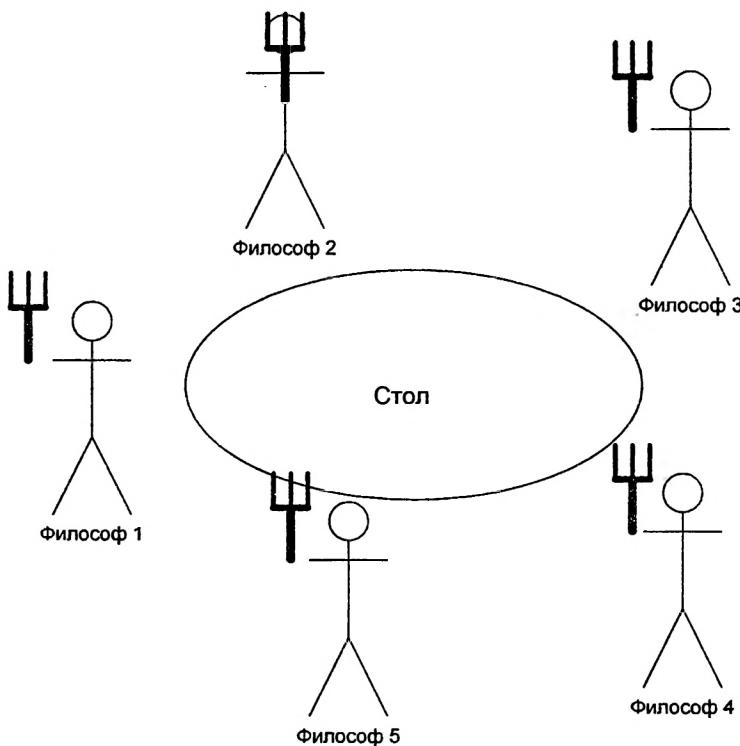


Рис. 7.4. Мертвая блокировка в исполнении пяти философов

Цикл можно разомкнуть, пронумеровав вилки и потребовав, чтобы каждый философ брал сначала вилку с меньшим номером, и только потом — с большим. Если вилки пронумерованы по часовой стрелке, для всех философов, кроме одного, это требование совпадает с высказанным в предыдущем абзаце — сначала брать правую вилку, и лишь потом дожидаться левой. Однако для того, кто попал между пятой и первой вилками, это правило обращается — он должен сначала дожидаться левой вилки, и только потом брать правую. Легко продемонстрировать, что этот философ в среднем будет ждать своих вилок дольше, чем остальные четверо.

Если же философ берет вилки тогда и только тогда, когда они доступны обе одновременно, это может привести к проблеме голодного философа: согласовав свои действия, соседи бедняги могут неограниченно долгое время оставлять его голодным (рис. 7.5).

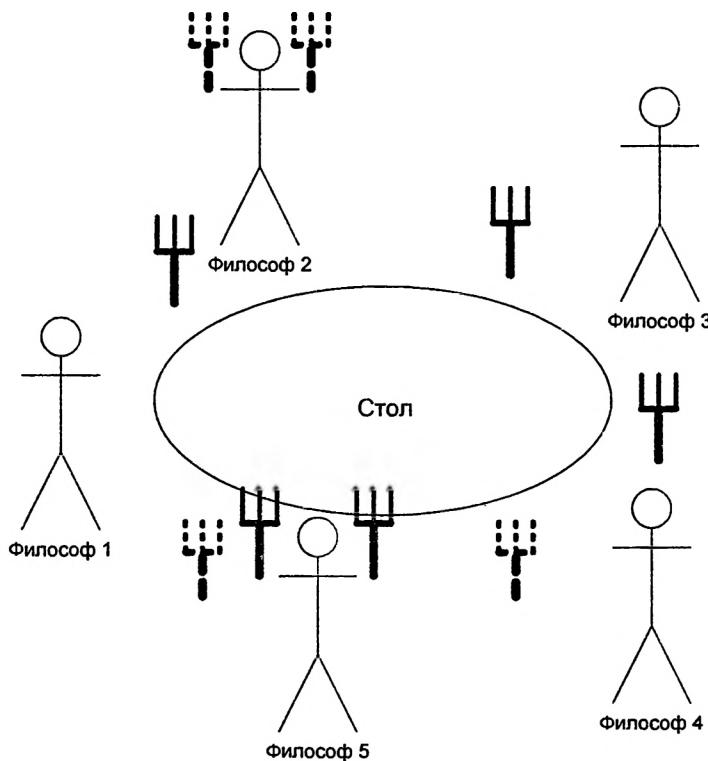


Рис. 7.5. Голодный философ

7.3. Примитивы синхронизации с ожиданием

Посмотрев на примеры 7.2 и 7.3, внимательный читатель должен отметить, что используемая конструкция подозрительно похожа на работу с внешними устройствами в режиме опроса. Действительно, опрос флаговой переменной в цикле хотя и обеспечивает гарантию взаимоисключений, но обладает всеми недостатками, которые мы указывали для опроса внешнего устройства.

В случае исполнения параллельных нитей на одном процессоре данный метод имеет еще один недостаток: пока одна из нитей занимается опросом, ни-

какая другая нить не может исполняться, потому что процессор загружен не-продуктивной работой.

Легко видеть, что в данном случае использование прерываний или какого-то их аналога проблемы не решит: в лучшем случае, "прерывание" будет вызываться не в той нити, в которой нужно, сводя задачу взаимоисключения к предыдущей, только с уже новой флаговой переменной, а в худшем — приведет к возникновению еще одной нити. Задача взаимодействия между асинхронными нитями, таким образом, сводится к требованию того, чтобы нити в какие-то моменты переставали быть асинхронными, *синхронизировались*.

Если у нас уже есть примитив взаимного исключения, мы можем решить задачу синхронизации, предоставив еще один примитив, который позволяет активному процессу остановиться, ожидая, пока флаговая переменная не примет "правильное" значение, и продолжить исполнение после этого. При обработке прерываний роль такого примитива может исполнять команда остановки процессора: у всех современных процессоров прерывание останавливает "исполнение" этой команды, а возврат из обработчика передает управление на следующую команду, таким образом выводя процессор из спящего состояния. В многопроцессорной конфигурации можно ввести средство, посредством которого один процессор может вызывать прерывание другого, — и тогда каждый из процессоров системы сможет ожидать другого, переходя в режим сна. При реализации же многопоточности на одном процессоре (см. разд. 8.2) примитив засыпания (блокировки) нити должен предоставляться модулем, ответственным за переключение потоков.

Впрочем, если операции над флагом, засыпание потока и его пробуждение реализованы разными примитивами, мы рискуем получить новую проблему (пример 7.5). Она состоит в том, что если пробуждающий сигнал поступит в промежутке между операторами `testandset` и `pause`, мы его не получим. В результате операция `pause` приведет к засыпанию нашей нити навсегда.

Пример 7.5. Ошибка потерянного пробуждения (lost wake-up bug)

```
program пауза
var flag: Boolean;
procedure процесс1
  var myflag: Boolean;
  while True do
    begin
      myflag := True;
      testandset(myflag, flag);
      if myflag then
```

```
(* Обратите внимание, что проверка флага *
 * и засыпание – это разные операторы! *)
pause;
критическаясекция();
flag := False;
end
end;
```

Одно из решений состоит в усложнении примитива `pause`: он должен засыпать, если и только если сигнал еще не приходил. Усложнение получается значительное: мало того, что перед засыпанием надо проверять нетривиальное условие, необходимо еще предусмотреть какой-то способ сброса этого условия, если мы предполагаем многократное использование нашего примитива.

Если писать на ассемблере или родственных ему языках, можно пойти и более изощренным путем (пример 7.6). Подмена адреса возврата в обработчике прерывания гарантирует нам, что если прерывание по установке флага произойдет в промежутке между метками `label` и `ok`, мы перейдем на метку `label` и, вместо того, чтобы заснуть навеки, благополучно проверим флаг и войдем в критическую секцию.

Пример 7.6. Обход ошибки потерянного пробуждения

```
.text
flag: .int 0
jmpbuf: .int 0

flag_interrupt:
pushl %eax
tst jmpbuf
bz clrflagonly
; подменяем адрес возврата
pushl %ebx
move jmpbuf, %eax
movl %esp, %ebx ; x86 не умеет адресоваться относительно %esp
movl %eax, RETURN_ADDRESS_OFFSET(%ebx)
popl %ebx
clrflagonly:
xorl %eax, %eax
movl %eax, flag
```

```

popl eax
iret

process1:
.....
movl setjmp, %eax
xchgl %eax, jmpbuf
setjmp:
move 1, %eax
xchgl %eax, flag
tstl %eax
bz ok
halt
ok:
kritическая секция
xor %eax, %eax
movl %eax, flag
.....

```

Более элегантный и приемлемый для языков высокого уровня путь решения этой проблемы состоит в том, чтобы объединить в атомарную операцию проверку флага и засыпание. Теперь нас с читателем можно поздравить с изобретением двоичного семафора Дейкстры.

7.3.1. Семафоры

Но когда ты проснешься, скрой свой испуг
 Это был не призрак, это были только звук
 Это тронулся поезд, на который ты не попадешь.
Б. Гребенщиков

Семафор Дейкстры представляет собой целочисленную переменную, с которой ассоциирована очередь ожидающих нитей. Ставясь пройти через семафор, нить пытается вычесть из значения переменной 1. Если значение переменной больше или равно 1, нить проходит сквозь семафор успешно (семафор открыт). Если переменная равна нулю (семафор закрыт), нить останавливается и ставится в очередь.

Закрытие семафора соответствует захвату объекта или ресурса, доступ к которому контролируется этим семафором. Если объект захвачен, остальные нити вынуждены ждать его освобождения. Закончив работу с объектом (выйдя из критической секции), нить увеличивает значение семафора на единицу,

открывая его. При этом первая из стоявших в очереди нитей активизируется и вычитает из значения семафора единицу, снова закрывая семафор. Если же очередь была пуста, то ничего не происходит, просто семафор остается открытый. Тогда первая нить, подошедшая к семафору, успешно пройдет через него. Это действительно похоже на работу железнодорожного семафора, контролирующего движение поездов по одноколейной ветке (рис. 7.6).



Рис. 7.6. Железнодорожный семафор

Наиболее простым случаем семафора является *двоичный семафор*. Начальное значение флаговой переменной такого семафора равно 1, и вообще она может принимать только значения 1 и 0. Двоичный семафор соответствует случаю, когда с разделяемым ресурсом в каждый момент времени может работать только одна нить.

Семафоры общего вида могут принимать любые неотрицательные значения. В современной литературе такие семафоры называют *семафорами-счетчиками* (counting semaphore). Это соответствует случаю, когда несколько нитей могут работать с объектом одновременно, или когда объект состоит из нескольких независимых, но равноценных частей — например, несколько одинаковых принтеров. При работе с такими семафорами часто разрешают процессам вычитать и добавлять к флаговой переменной значения, большие единицы. Это соответствует захвату/освобождению нескольких частей ресурса.

Многие системы предоставляют также сервисы, позволяющие просмотреть состояние семафора без его изменения и произвести "неблокирующуюся" форму захвата, которая возвращает ошибку в ситуации, когда нормальный захват семафора привел бы к блокировке. Теоретики не очень любят такие примитивы, но при разработке сложных сценариев взаимодействия с участием многих семафоров они бывают полезны.

Во многих современных книгах и операционных системах семафорами называются только семафоры общего вида, двоичные же семафоры носят более краткое и выразительное имя *мутекс* (mutex — от MUTual EXclusion, взаимное исключение). Проследить генезис этого названия мне не удалось, но можно с уверенностью сказать, что оно вошло в широкое употребление не ранее конца 80-х годов XX века. Так, в разрабатывавшейся в середине 80-х годов OS/2 1.0, двоичные семафоры еще называются семафорами, а в Win32, разработка которой происходила в начале 90-х, уже появляется назва-

ние mutex. Операции над мутексом называются *захватом* (acquire) (соответствует входу в критическую секцию) и *освобождением* (release) (соответствует выходу из нее).

В современных ОС термин mutex имеет дополнительный важный оттенок. А именно, предполагается, что нить должна осуществлять операции захвата и освобождения парами: за захватом ресурса всегда должно следовать его освобождение. Недопустимы ситуации, когда одна нить захватывает ресурс, а другая его освобождает. Благодаря этому для каждого заблокированного мутекса можно указать, какая именно нить его заблокировала. Пользуясь данной информацией, можно проверять, не приведет ли захват мутекса к мертвой блокировке и решать некоторые другие своеобразные проблемы, например проблему инверсии приоритета (см. разд. 8.2.3). Однако есть ситуации, когда требование парности операций оказывается слишком стеснительным для программиста.

Задача производитель-потребитель

Две нити обмениваются потоком данных через буфер фиксированного размера. Для простоты допустим, что данные представляют собой поток байтов, а буфер имеет размер один байт. Одна из нитей — производитель, она записывает данные в буфер. Вторая нить — потребитель, она считывает данные из буфера. При этом, если производитель хочет записать данные в буфер, но обнаруживает, что потребитель еще не считал предыдущий байт, производитель должен остановиться и ждать. Аналогично, потребитель должен ждать, если производитель еще не записал новые данные.

Очевидно, что эта задача не сводима к простой защите критической секции, в которой происходит доступ к буферу. Легко понять, что она неразрешима с использованием одиночного примитива синхронизации, будь то мутекс или семафор общего вида. Довольно легко также убедиться в том, что данная задача неразрешима с использованием двух мутексов.

Для доказательства этого утверждения нам придется сначала доказать своего рода лемму, а именно — утверждение, что при решении задачи производитель-потребитель недопустимы ситуации, когда одна из нитей не держит ни одного мутекса. Утверждение это следует из того факта, что при вторая нить в таких условиях не будет останавливаться. Если это потребитель, то он может считать из буфера одни и те же данные несколько раз, а если производитель — перезаписать данные, еще не прочитанные потребителем.

Таким образом, ситуация, когда одна из нитей не удерживает ни одного мутекса, недопустима. Также недопустима ситуация, когда одна из нитей удерживает оба мутекса — ведь при этом вторая нить не будет удерживать ни одного! Поэтому приемлемы только две ситуации: когда производитель удерживает мутекс A, а потребитель — мутекс B, и наоборот. Казалось бы, этих двух приемлемых ситуаций достаточно, чтобы закодировать два возможных состояния системы: если производитель удерживает мутекс A, то его очередь работать с буфером; если он удерживает мутекс B, то с буфером работает потребитель.

Проблема, однако, состоит в том, что мы не можем перейти из одного состояния в другое, не проходя через одно из "запрещенных" состояний. Таким обра-

зом, на двух мутексах, без использования каких-либо дополнительных средств синхронизации, задача неразрешима.

Однако существует решение этой задачи на двух семафорах общего вида, принадлежащее Дейкстре (пример 7.7). Обдумывание, почему вышеприведенное доказательство не распространяется на семафоры общего вида, я оставляю читателю в качестве упражнения.

Пример 7.7. Решение задачи производитель–потребитель на двух семафорах

```
program производитель–потребитель
var sem1, sem2: семафор;
var buf: char;
procedure производитель
  var mychar: char
  while True do
    begin
      mychar:=произвести;
      lock(sem1);
      buf:=mychar;
      unlock(sem2);
    end
  end;
procedure потребитель
  var mychar: char
  while True do
    begin
      lock(sem2);
      mychar:=buf;
      unlock(sem1);
      обработать (mychar);
    end
  end;
```

Многие ОС предоставляют для синхронизации семафоры Дейкстры, мутексы или похожие на них механизмы.

Флаги событий в RSX-11 и VMS

Так, например, в системах RSX-11 и VMS основным средством синхронизации являются **флаги событий** (event flags). Процессы и система могут очищать (clear) или взводить (set) эти флаги. Флаги делятся на локальные и глобальные. **Локальные флаги** используются для взаимодействия между процессом и ядром системы, **глобальные** — между процессами. Процесс может остановиться,

ожидая установки определенного флага, поэтому флаги во многих ситуациях можно использовать вместо двоичных семафоров. Кроме того, процесс может связать с флагом события процедуру-обработчик AST (Asynchronous System Trap — асинхронно [вызываемый] системный обработчик).

AST во многом напоминают сигналы или аппаратные прерывания. В частности, флаги событий используются для синхронизации пользовательской программы с асинхронным исполнением запросов на ввод-вывод. Исполняя запрос, программа задает локальный флаг события. Затем она может остановиться, ожидая этого флага, который будет введен после исполнения запроса. При этом мы получаем псевдосинхронный ввод-вывод, напоминающий синхронные операции чтения/записи в UNIX и MS DOS. Но программа может и не останавливаться! При этом запрос будет исполняться параллельно с исполнением самой программы, и она будет оповещена о завершении операции соответствующей процедурой AST.

Асинхронный ввод-вывод часто жизненно необходим в программах реального времени, но бывает полезен и в других случаях.

7.3.2. Блокировки чтения-записи

В начале главы мы видели, что проблема взаимоисключения возникает только при модификации разделяемых структур данных. Одни только операции чтения немодифицируемой структуры не могут привести к рассогласованию данных и поэтому безопасны. Однако если операции чтения чередуются с операциями записи, мы все равно вынуждены защищать их блокировками — не друг от друга, а от возможности перехвата управления нитью, которая осуществляет запись.

Во многих случаях (по некоторым оценкам, в 90-99% всех программ) операции чтения преобладают над операциями записи, поэтому оказывается целесообразно ввести два типа захватов: для чтения и для записи. *Захват для чтения* (read lock) разрешает другим нитям читать заблокированную структуру данных, но запрещает модифицировать ее. *Захват для записи* (write lock) соответствует обычному мутексу — только одна нить может удерживать такой захват; при этом ни одна нить не может также удерживать и захваты для чтения.

Разумеется, оба этих захвата осуществляются над одним примитивом, который так и называется: *захват чтения-записи* (read-write lock). Как и семафор Дейкстры, этот примитив имеет очередь ожидающих нитей. Как и все остальные блокирующие примитивы, этот примитив может приводить к мертвым блокировкам и остальным проблемам, перечисленным в предыдущих разделах.

При небольшом количестве нитей и при сильном преобладании чтений над записями такие примитивы блокировки обеспечивают значительный рост на-

блюдаемой производительности по сравнению с "честными" простыми блокировками. Однако по мере роста количества модификаций и, особенно, когда модификация осуществляется несколькими нитями, возникает проблема "голодания по чтению" (read starving): когда значительную часть времени оказывается установлена блокировка для записи, читающие нити вынуждены ждать, и преимущество перед простой блокировкой практически исчезает.

Есть соблазн попытаться разрешить эту проблему за счет предпочтения нитей, которые ставят захват для чтения. Для этого не нужно изменение их приоритета с точки зрения планировщика, достаточно лишь разрешать устанавливать захват, даже если в очереди стоит процесс, ожидающий права установить захват на запись. Однако это приведет к "голоданию по записи" (write starving); при наивной реализации данного подхода, с высокой вероятностью модифицирующая нить окажется заблокирована на неограниченное время.

Практика показывает, что единственная схема, более или менее справедливо учитывающая интересы как пишущих, так и читающих нитей — это общая очередь, в которую все запросы устанавливаются по мере их поступления, и из которой выходят по мере того, как соответствующая блокировка может быть захвачена. Понятно, что при таком подходе каждая блокировка для записи задерживает всю следующую за ней очередь, поэтому во многих теоретических работах эту ситуацию все равно называют "голоданием по чтению".

7.3.3. Копирование при записи

Одна из попыток оптимизировать работу блокировок чтения-записи состоит в применении копирования перед модификацией. Модифицируя ресурс, нить создает его копию. При этом модификация никак не мешает чтению, осуществляемому остальными нитями: эти нити продолжают читать согласованную старую копию.

Закончив модификацию, нить даже не переписывает изменения в старый ресурс, а просто обновляет указатель в индексе ресурсов. При этом все нити, обратившиеся к ресурсу до его обновления, видят старое значение, а после обновления — новое. При этом чтение может осуществляться вообще без какой-либо блокировки; если согласованной модификации подвергаются несколько ресурсов, блокировки все равно оказываются нужны, но их надо удерживать только во время обновления указателей на ресурсы, т. е. гораздо меньшее время, чем необходимо для изменения самих ресурсов.

При обычном программировании на С/С++ с явным освобождением объектов попытка реализовать такой подход приведет к множеству проблем, главные из которых — это опасность сохранения ссылок на объект модифицировавшими его нитями и удаление старой копии объекта: она должна уничтожать-

ся, когда последняя из нитей, захвативших ресурс до его обновления, освободит его.

Поэтому наибольший интерес данный подход вызывает в средах программирования со сборкой мусора (таких, как Java/.NET) и в других ситуациях, где обращение к объекту так или иначе проходит через какие-либо транслирующие слои, "мастер-указатели", ручки и т. д. На эти слои можно возложить работу по установке блокировок, копированию объектов, обновлению ссылок и отслеживанию освобожденных объектов. Наибольший успех данный подход имел в серверах реляционных СУБД, где он применяется при реализации транзакций (операций модификации групп записей, для которых создается иллюзия атомарности). Далее в этом разделе мы рассмотрим еще некоторые аспекты копирования при записи, важные для реализации транзакций. Сама по себе тема обработки транзакций достаточно обширна. В этой книге мы будем рассматривать лишь отдельные ее стороны, которые реализуются (или реализация которых планируется) в операционных системах.

Судя по материалам сайта [[research.sun.com scalable](http://research.sun.com/scalable)], во время подготовки к печати второго издания велись исследования и разработки механизмов такого типа для реализаций языка Java, в том числе со специализированной аппаратной поддержкой на уровне кэш-памяти и ММУ.

Копирование при записи обеспечивает возможность одновременно осуществлять модификацию ресурса и считывание его состояния. Однако очевидно, что такое копирование не позволяет одновременно модифицировать ресурс двум нитям. Рассмотрим простой сценарий исполнения двух нитей, аналогичный рассматривавшейся в начале главы параллельной работе двух нитей с разделяемым счетчиком: нить А создает копию ресурса для модификации, затем нить Б делает то же самое, далее нить Б заканчивает модификацию и обновляет мастер-указатель, затем то же самое делает нить А. Изменения, внесенные нитью Б, теряются.

На первый взгляд, кажется, что единственным выходом из этой ситуации является включение в схему блокировок — т. е. получается, что чтение одиночных ресурсов может осуществляться без блокировок, но модификация обязательно требует блокировки.

В некоторых случаях, однако, оказываются целесообразны более сложные пути решения этой проблемы. Один из них состоит в том, чтобы сохранять обе версии состояния ресурса. В зависимости от потребностей приложения, обе эти версии могут продолжать существовать в виде независимых ресурсов либо какая-то специальная процедура может сливать их в единый объект. Способ такого слияния, разумеется, решающим образом зависит от того, что ресурс собой представляет, и что означает его состояние с точки зрения логики приложения. Мне неизвестно систем программирования, в которых такой

подход применялся бы к хранимым в памяти объектам, но именно так разрешаются конфликты при множественной модификации в нереляционных СУБД и в системах управления версиями, таких как CVS.

Например, в Lotus Notes при этом создаются так называемые "конфликты репликации" (replication conflicts) — копии документов, снабженных специальным флагом. Удаление этих документов, возможно (но не обязательно) в сочетании с их слиянием с основным документом, возлагается либо на администратора данных, либо на бизнес-логику приложения.

В системах управления версиями конфликты модификации могут разрешаться при помощи создания ветвей проекта (branch или fork). Доступные на рынке системы предоставляют довольно-таки "интеллектуальные" средства объединения ветвей, но все используемые при этом эвристические приемы рассчитаны, главным образом, на исходные тексты программ. Результат объединения, в любом случае, нуждается в ручном просмотре и анализе корректности.

Еще более радикальный подход к решению проблемы конфликтов при модификации предлагается схемами обработки транзакций в реляционных СУБД.

Введение в обработку транзакций

Сервер реляционной СУБД исполняет запросы к данным, описываемые на специализированном языке SQL (Sequential Query Language — последовательный язык запросов). Операции модификации данных в этом языке могут объединяться в блоки, называемые транзакциями (transaction). Определение транзакции в первом приближении совпадает с определением, которое я приводил ранее, — это группа операций, которые либо исполняются все вместе, либо не исполняются вовсе. Для реализации данного требования СУБД должна так или иначе сохранять состояние БД на момент начала транзакции. Благодаря этому транзакция может завершаться двумя способами — подтверждением (commit) или откатом (rollback).

При подтверждении измененные при транзакции данные записываются в СУБД (впрочем, запросы, запущенные до подтверждения, продолжают работать с копией старых данных). При откате, все изменения отменяются, будто транзакция и не начиналась. Видно, что эти требования естественным образом реализуются посредством копирования при записи; впрочем, многие реальные СУБД модифицируют непосредственно рабочие данные, а старые данные записывают в специальную область памяти, называемую *сегментом отката* (rollback segment). При подтверждении транзакции сегмент отката отбрасывается, а при откате — копируется в рабочий сегмент. Это дает определенные преимущества, если в среднем транзакции чаще заканчиваются подтверждением, чем откатом.

В SQL на транзакции накладывается несколько более жесткое требование *сериализуемости* — если параллельно исполняется несколько транзакций, результат их исполнения должен соответствовать результату последовательного исполнения тех же транзакций в некотором порядке. Это требует разрешения конфликтов при множественной модификации данных.

Основные приемы, применяемые для этого, — пессимистическая блокировка (*pessimistic locking*) и оптимистическая блокировка (*optimistic locking*).

При пессимистической блокировке система определяет, какие данные будут подвергнуты модификации в ходе транзакции, и устанавливает на них блокировку. Некоторые серверы, как, например, ранние версии Sybase/MS SQL, ставили такую блокировку целиком на таблицы, которые упоминались в коде транзакции как модифицируемые. Более совершенные серверы, в том числе и новые версии MS SQL, используют различные (и довольно сложные) эвристические приемы для определения того, какие именно записи следует заблокировать. Таким образом, при пессимистической блокировке транзакции действительно сериализуются за счет блокировок; мертвые блокировки при этом избегаются за счет того, что все блокировки устанавливаются одновременно, в момент начала транзакции.

Попытки отложить установку блокировки непосредственно до момента, когда происходит модификация, опасны в ряде отношений, прежде всего потому, что могут привести к мертвым блокировкам. Впрочем, реальные серверы СУБД нередко все-таки откладывают блокировку записей и, таким образом, обеспечивают определенный параллелизм; при этом возможные мертвые блокировки устраняются либо за счет отката некоторых из вошедших в блокировку транзакций, либо за счет сложных схем предотвращения, основанных на так называемом *алгоритме банкира* (см. [Танненбаум 2006]).

При оптимистической блокировке никаких блокировок, в действительности, не ставится. Сервер исполняет транзакции параллельно. Если он обнаруживает конфликт при модификации какой-то из записей, он откатывает одну из транзакций и перезапускает ее.

Очевидно, что пессимистическая блокировка предпочтительна в условиях, когда бизнес-логика приложения предполагает частые изменения одних и тех же данных и ожидание этих изменений, а оптимистическая — когда наборы записей, над которыми работают эти транзакции, перекрываются редко. В этих условиях транзакции откатываются редко и оптимистическая блокировка обеспечивает малое время обработки каждой отдельной транзакции и высокий уровень параллелизма при их исполнении.

7.3.5. Захват участков файлов

Семафоры удобны при синхронизации доступа к единому ресурсу, такому как принтер или неделимая структура данных. Если же нам нужна синхрони-

зация доступа к ресурсу, имеющему внутреннюю структуру, например к файлу с базой данных, лучше использовать другие методы.

Если говорить именно о файле, оказывается удобным блокировать доступ к участкам файла.

Блокировка участков файла в Unix

Захват участков файла в качестве средства синхронизации был известен еще с 60-х годов XX века, но в том виде, который описан в стандартах ANSI и POSIX, он был реализован в ОС UNIX в начале 70-х.

В UNIX возможны два режима захвата: допустимый (*advisory*) и обязательный (*mandatory*). Как та, так и другая блокировка может быть блокировкой чтения либо записи. Допустимая блокировка является "блокировкой для честных": она не оказывает влияния на подсистему ввода-вывода, поэтому программа, не проверяющая блокировок или игнорирующая их, сможет писать или читать из заблокированного участка без проблем. Обязательная блокировка требует больших накладных расходов, но запрещает физический доступ к файлу: чтение или запись, в зависимости от типа блокировки.

При работе с разделяемыми структурами данных в ОЗУ было бы удобно иметь аналогичные средства, но их реализация ведет к большим накладным расходам даже на системах с виртуальной памятью, поэтому ни одна из известных мне систем таких средств не имеет. Библиотека POSIX threads предоставляет своеобразную форму мутекса (*read/write lock*), который, как и описанные файловые примитивы, может быть многократно захвачен для чтения и лишь однократно — для записи. Однако мы должны заводить такой примитив для каждой единицы разделяемого ресурса и не можем одним вызовом захватить сразу много подобных единиц.

Впрочем, в современных версиях системы UNIX есть возможность отображать файл в виртуальную память. Используя при этом допустимую блокировку участков файла, программы могут синхронизировать доступ к нему (обязательная блокировка делает невозможным отображение в память).

7.3.6. Мониторы и серверы транзакций

Захват участков файла теоретически позволяет реализовать любую требуемую структуру взаимоисключения для процессов, работающих с этим файлом. Однако, практически, при работе с файлом большого количества нитей (например, многопользовательской системы управления базами данных) различные нити часто оказываются вынуждены ждать друг друга, что приводит к резкому увеличению времени реакции системы. С этим явлением хорошо знакомы разработчики и пользователи файловых СУБД, таких как FoxPro или dBase IV.

В случае СУБД решение состоит в создании сервера БД, или *сервера транзакций*, который вместо примитивов захвата участков файлов или таблиц

предоставляет пользователям понятие транзакции, рассматриваемое нами в разд. 7.3.4.

Если пользовательская сессия объявила начало транзакции, изменения, которые она вносит в таблицы, непосредственно в таблицах не отражаются, и другие сессии, которые обращаются к вовлеченным в транзакцию данным, получают их исходные значения. Завершив модификацию, пользовательская сессия объявляет завершение транзакции. Транзакция может завершиться как *выполнением* (*commit*), так и *откатом* (*rollback*).

В случае отката измененные данные просто игнорируются. В случае же выполнения сервер вносит изменения в таблицы, однако во время изменений он все равно предоставляет другим нитям данные в том состоянии, в котором они были до начала транзакции. Такой подход увеличивает потребности в оперативной и дисковой памяти (все данные, изменяемые в ходе транзакции, должны храниться минимум дважды: в измененном и в исходном видах), но обеспечивает резкое сокращение времени реакции и определенное упрощение кодирования: программист теперь не должен явно перечислять все данные, которые ему надо заблокировать в ходе транзакции. Кроме того, двойное хранение данных позволяет восстановить либо результат транзакции, либо состояние данных до ее начала, после сбоя системы.

Современные серверы СУБД представляют собой сложные программные комплексы, которые кроме собственно "развязки" транзакций предоставляют сложные сервисы оптимизации запросов, индексации и кэширования данных. Да и точное описание понятия транзакции в современных языках запросов к реляционным СУБД (SQL, RPG и др.) несколько сложнее приведенного ранее. Однако детальное обсуждение этого вопроса уело бы нас далеко в сторону от темы книги. Читателю, интересующемуся этой темой, можно порекомендовать книги [Дейт 1999, Бобровски 1998].

Аналогичный серверам транзакций подход нередко используется и в более простых случаях межпроцессного взаимодействия. С разделяемым ресурсом ассоциируется специальная нить, называемая *монитором ресурса*. Остальные нити не могут обращаться к ресурсу напрямую и взаимодействуют только с монитором. Монитор может предоставлять нитям-клиентам непротиворечивые состояния контролируемого им ресурса (необязательно совпадающие с текущим состоянием ресурса) и устанавливать очередность запросов на модификацию.

Даже при довольно простой стратегии управления ресурсом монитор полезен тем, что скрывает (инкапсулирует) структуру и особенности реализации разделяемого ресурса от клиентских нитей. Типичный пример мониторного процесса — драйвер внешнего устройства в многозадачных ОС.

7.4. Гармонически взаимодействующие последовательные потоки

В разд. 7.1 мы видели, что проблемы взаимоисключения и синхронизации возникают тогда и только тогда, когда несколько нитей разделяют один и тот же ресурс. Взаимоисключение доступа к разделяемым данным приводит к необходимости вводить дополнительные сущности (семафоры и другие примитивы взаимоисключения и синхронизации) и усложнять программу.

Необоснованное расширение участков исключительного доступа приводит к росту времени реакции системы и снижению производительности, а стремление сократить их может привести к ошибкам соревнования. Поэтому разделяемые структуры данных являются предметом ненависти теоретиков и источником серьезных ошибок при разработке программ.

Желание устранить эти проблемы привело в свое время Э. Дейкстру к концепции, известной как *гармонически взаимодействующие последовательные потоки*. Эта концепция состоит в следующем:

1. Каждый поток (нить) представляет собой независимый программный модуль, для которого создается иллюзия чисто последовательного исполнения.
2. Нити не имеют разделяемых данных.
3. Все обмены данными и вообще взаимодействие происходят с использованием специальных примитивов, которые одновременно выполняют и передачу данных, и синхронизацию.
4. Синхронизация, не сопровождающаяся передачей данных, просто лишена смысла — нити, не имеющие разделяемых структур данных, совершенно независимы и не имеют ни критических точек, ни нереентерабельных модулей.

Третье требование является ключевым. Поток, модифицирующий (в данном случае — генерирующий) данные, исполняет примитив передачи данных тогда и только тогда, когда порожденные им данные уже целостны, или передает их такими блоками, каждый из которых целостен сам по себе.

Примечание

Важно подчеркнуть, что гармоническое взаимодействие не исключает критических секций из алгоритма. Оно лишь сосредотачивает критические секции и связанные с ними примитивы взаимоисключения внутри примитивов передачи данных.

Гармоническое взаимодействие, строго говоря, не исключает проблему мертвых блокировок: замкнув гармонически взаимодействующие нити в кольцо

(А получает информацию от В, В от С, С от А), мы можем получить классическую трехзвенную мертвую блокировку. Впрочем, в данном случае блокировка требует наличия циклической зависимости данных, которая свидетельствует о серьезных ошибках проектирования программы (и, возможно, о внутренней противоречивости требований к этой программе). Такая ошибка может быть относительно легко обнаружена посредством формального анализа спецификаций и потоков данных.

На практике, впрочем, гармоническое взаимодействие обходит основную массу проблем, возникающих при взаимоисключении доступа ко многим ресурсам — и блокировки, и "проблему голодного философа". Дело в том, что гармонически взаимодействующий поток имеет дело не с разделяемым ресурсом непосредственно, а с копией состояния этого ресурса. Если нам нужны два ресурса, мы (не обязательно одновременно) снимаем с них две копии — для этого нам не надо одновременно захватывать сами ресурсы. В терминах *разд. 7.2.1* это можно описать, как предотвращение мертвых блокировок за счет того, что ни одна нить никогда не захватывает более одного ресурса.

В частности, поэтому групповые операции над примитивами гармонического взаимодействия (оператор `select` в Ada, системный вызов `select` в системах семейства Unix, команда `alt` в микропроцессорах Inmos Transputer) работают не по принципу транзакции, а возвращают управление и данные при условии, что хотя бы один из примитивов группы готов эти данные предоставить. Воспроизвести "проблему голодного философа" в этих условиях невозможно.

В современных системах реализован целый ряд средств, которые осуществляют передачу данных совместно с синхронизацией: почтовые ящики (`mailboxes`) в системах линии RSX-11 — VMS, трубы (`pipes`) (в русскоязычной литературе их часто называют программными каналами) в UNIX, randеву (`rendezvous` — свидание) в языке Ada, линки (`link` — соединение) в Transputer и т. д. Большинство этих средств будет обсуждаться подробнее в *разд. 7.4.1*.

Примитивы могут быть двухточечные (допускающие только один приемник и один передатчик) либо многоточечные, допускающие несколько приемников и передатчиков. Многоточечность бывает как симметричная, когда может быть несколько и приемников, и передатчиков, так и асимметричная, когда допускается только один приемник при нескольких передатчиках и наоборот.

Примитив может передавать неструктурированный поток байтов либо структурированные данные, разбитые на сообщения определенного размера. В первом случае передатчик может порождать данные блоками одного размера, а приемник — считывать их блоками другого размера. Во втором случае приемник всегда обязан прочитать сообщение целиком (возможно, от-

бросив какую-то его часть). Сообщения могут быть как фиксированного, так и переменного размера.

Примитив может быть синхронным, буферизованным или с негарантированной доставкой. В первом случае передатчик вынужден ждать, пока приемник не прочитает все переданные данные. Во втором данные складываются в буфер и могут быть прочитаны приемником позднее. В третьем случае, если потенциальный приемник не был готов принять сообщение, оно просто игнорируется.

Синхронные примитивы могут использоваться не только для гармонического взаимодействия, но и для реализации примитивов чистой синхронизации. В частности, в учебниках по языку Ada (например, [Василеску 1990]) и по программированию для транспьютеров (например, [Харп 1993]) почему-то любят приводить примеры реализации семафоров на основе, соответственно, randеву и линков.

Буферизованные примитивы для синхронизации использованы быть не могут. Зато буферизация полезна, если нам надо согласовать скорости работы нитей, имеющих разные приоритеты, — например, если нить реального времени должна быстро обработать событие и передать часть данных для отложенной обработки менее приоритетной нитью, не допустив при этом собственной блокировки.

Буферизованный примитив может быть легко реализован на основе пары синхронных примитивов и нити-монитора буфера (или очереди). В этом смысле, синхронные примитивы являются более низкоуровневыми, чем буферизованные.

Синхронные примитивы по природе своей всегда двухточечные. Да и в случае буферизованных примитивов многоточечное взаимодействие не всегда легко реализуемо, а иногда просто опасно, особенно в случае потоковой передачи данных: операция считывания данных из потока необратима, а естественного разбиения потока на сообщения нет, поэтому если один из приемников по ошибке захватит кусок сообщения, предназначенного не ему, то данные в потоке будут необратимо испорчены.

Впрочем, некоторые потоковые примитивы, например трубы (pipes) в системах семейства Unix, допускают (хотя документация и рекомендует пользоваться этим с осторожностью) наличие нескольких передатчиков при одном приемнике.

Напротив, симметрично многоточечные очереди сообщений широко распространены. Часто такие примитивы позволяют потребителю отбирать сообщения из очереди по какому-то критерию, например по значению специального поля, называемому типом или тегом. Ряд широковещательных и групповых

сервисов передачи данных относится к категории примитивов с негарантированной доставкой.

Подразделения примитивов на потоковые и структурированные с одной стороны и буферизованные и синхронные с другой стороны (если пока отвлечься от сервисов с негарантированной доставкой) практически ортогональны друг другу: существуют все четыре варианта (табл. 7.1). Легко понять, что передавать поток неструктурированных данных в режиме негарантированной доставки бессмысленно: разрывы потока неизбежны, а мы не сможем даже узнать, произошел ли такой разрыв, и если произошел, то где. Все существующие сервисы с негарантированной доставкой передают только сообщения.

Таблица 7.1. Примитивы синхронизированной передачи данных

Примитивы	Синхронные	Буферизованные
Потоковые	Линки (транспьютер)	Трубы (Unix), сокеты (TCP/IP)
Структурированные	Рандеву (Ada)	Очереди сообщений

На первый взгляд, вообще непонятно, почему кому-то может быть полезен сервис с негарантированной доставкой. Но под это описание подходят многие низкоуровневые сетевые протоколы (Ethernet, IP) и некоторые относительно высокоуровневые сетевые сервисы, так называемые дейтаграммные протоколы. Примером такого сервиса является UDP (User Datagram Protocol), входящий в семейство протоколов TCP/IP.

В сетевых протоколах отсутствие гарантии доставки сообщения имеет двойкий смысл — сообщение может быть потеряно не только по невниманию получателя, но и из-за физических ошибок передачи или перегрузки маршрутизаторов и/или каналов связи по дороге от передатчика к приемнику.

В этом смысле можно сказать, что разработчики сетевых протоколов вынуждены использовать негарантированную доставку не потому, что им нужна именно она, а потому, что других средств-то и нет.

В чистом виде негарантированная доставка приемлема для реализации одиночных запросов, на которые должен последовать одиночный же ответ. Если ответа за определенный протоколом интервал времени нет, передатчик повторяет запрос, а после некоторого количества попыток приходит к выводу, что приемник не функционирует, либо вообще отсутствует.

Для реализации надежной связи на основе сервисов с негарантированной доставкой используются различного рода подтверждения, так называемое *квитирование*. Понятно, что при реализации квитирования необходимо принимать во внимание возможность потери не только самого подтверждаемого

сообщения, но и пакета-подтверждения. Понятно также, что в большинстве случаев посылка подтверждения на каждое пришедшее сообщение нецелесообразна. Поэтому реальные протоколы такого рода относительно сложны (см., например, [RFC 0793]) и их обсуждение заслуживает отдельной книги [Иртегов 2004]. Накладные расходы при реализации таких протоколов значительны, поэтому часто оказывается целесообразно смириться с негарантированной доставкой.

Концепция гармонически взаимодействующих процессов очень привлекательна с теоретической точки зрения и позволяет легко писать правильные программы. Однако все примитивы синхронизированной передачи данных плохи именно тем, что требуют передачи, копирования данных. И передатчик, и приемник вынуждены иметь по буферу, в котором данные следует хранить (в случае буферизованных примитивов данные хранятся трижды). Накладные расходы при копировании данных большого объема также могут оказаться значительными.

Если нити исполняются на разных компьютерах, не имеющих общей памяти и соединенных лишь относительно (по сравнению с системной шиной) низкоскоростными сетевыми каналами, мы так или иначе не можем обойтись без передачи и двойного хранения данных. Впрочем, даже и в этом случае иногда имеет смысл подумать о применении многопортовой памяти или реализаций NUMA- или COMA-архитектуры.

При исполнении же нитей на одной машине по соображениям производительности и экономии памяти бывает нецелесообразно отказываться от разделяемой памяти и полностью переходить на примитивы гармонического взаимодействия. Чаще всего это происходит, когда к ресурсу выполняется много обращений для чтения, а модификации относительно редки, и при этом данные имеют большой объем. Даже в этом случае бывает целесообразно заменить прямое разделение памяти на мониторный процесс, а при доступе к данным получать у него лишь непосредственно необходимое их подмножество. Однако ситуации бывают разные, и не всегда такое решение оказывается оптимальным.

В этом смысле разделяемая память напоминает другой предмет ненависти структурных программистов — оператор `goto`. И то, и другое при неразумном использовании является потенциальным источником ошибок и проблем, но иногда без них оказывается нельзя обойтись.

Программные каналы Unix

Одним из наиболее типичных средств такого рода является *труба (pipe)* или, что то же самое, *программный канал* — основное средство взаимодействия между процессами в ОС семейства Unix. В русскоязычной литературе трубы

иногда ошибочно называют конвейерами. В действительности, конвейер — это группа процессов, последовательно соединенных друг с другом односторонними трубами.

Труба представляет собой поток байтов. Поток этот имеет начало (исток) и конец (приемник). В исток этого потока можно записывать данные, а из приемника — считывать. Нить, которая пытается считать данные из пустой трубы, будет задержана, пока там что-нибудь не появится. Наоборот, пишущая нить может записать в трубу некоторое количество данных, прежде чем труба заполнится, и дальнейшая запись будет заблокирована. На практике труба реализована в виде небольшого (несколько килобайтов) кольцевого буфера. Передатчик заполняет этот буфер, пока там есть место. Приемник считывает данные, пока буфер не опустеет.

Трубу можно установить в режим чтения и записи без блокировки. При этом вызовы, которые в других условиях были бы остановлены и вынуждены были бы ожидать партнера на другом конце трубы, возвращают ошибку с особым кодом.

По-видимому, трубы, впервые реализованные не в Unix, а в его предшественнике Multics, являются одной из первых реализаций гармонически взаимодействующих процессов по терминологии Дейкстры.

Самым интересным свойством трубы является то, что чтение данных и запись в нее осуществляется теми же самыми системными вызовами `read` и `write`, что и работа с обычным файлом, внешним устройством или сетевым соединением (сокетом, от англ. `Socket` — разъем). На этом основана техника переназначения ввода-вывода, широко используемая в командных интерпретаторах UNIX. Она состоит в том, что большинство системных утилит получают данные из потока стандартного ввода (`stdin`) и выдают их в поток стандартного вывода (`stdout`). При этом, указывая в качестве потоков терминальное устройство, файл или трубу, мы можем использовать в качестве ввода, соответственно: текст, набираемый с клавиатуры, содержимое файла или стандартный вывод другой программы. Аналогично мы можем выдавать данные сразу на экран, в файл или передавать их на вход другой программы.

Так, например, компилятор GNU C состоит из трех основных проходов: препроцессора, собственно компилятора, генерирующего текст на ассемблере, и ассемблера. При этом внутри компилятора, на самом деле, также выполняется несколько проходов по тексту (в описании перечислено восемнадцать), в основном для оптимизации, но нас это в данный момент не интересует, поскольку все они выполняются внутри одной задачи. При этом все три задачи объединяются трубами в единую линию обработки входного текста — конвейер (`pipeline`), так что промежуточные результаты компиляции не занимают места на диске.

В системе UNIX труба создается системным вызовом `pipe(int fildes[2])`. Этот вызов создает трубу и помещает дескрипторы файлов, соответствующие входному и выходному концам трубы, в массив `fildes`. Затем мы можем выполнить `fork`, в различных процессах переназначить соответствующие концы трубы на место `stdin` и `stdout` и запустить требуемые программы (пример 7.8). При этом мы получим типичный конвейер — две задачи, стандартный ввод и вывод которых соединены трубой.

Пример 7.8. Код, создающий конвейер посредством труб

```
#include <unistd.h>

void pipeline(void) {
    /* stage 1 */
    int pipe1[2];
    int child1;
    int pipe2[2];
    int child2;
    int child3;

    pipe(pipe1);

    if ((child1=fork())==0) {
        close(pipe1[0]); /* Закрыть лишний конец трубы */
        close(1); /* Переназначить стандартный вывод */
        dup(pipe1[1]);
        close(pipe1[1]);
        /* Исполнить программу */
        execlp("du", "du", "-s", ".", NULL);
        /* Мы можем попасть сюда только при ошибке exec */
        perror("Cannot exec");
        exit(0);
    }
    close(pipe1[1]);
    if (child1==-1) {
        perror("Cannot fork");
    }
    /* stage 2 */
    pipe(pipe2);
    if ((child2=fork())==0) {
        close(0); /* Переназначить стандартный ввод */
        dup(pipe1[0]);
        close(pipe1[0]);
        close(pipe2[0]); /* Закрыть лишний конец трубы */
        close(1); /* Переназначить стандартный вывод */
        dup(pipe2[1]);
        close(pipe2[1]);
    }
```

```
/* Исполнить программу */
execlp("sort", "sort", "-nr", NULL);
/* Мы можем попасть сюда только при ошибке exec */;
perror("Cannot exec");
exit(0);
}
close(pipe1[0]);
close(pipe2[1]);

if (child2== -1) {
    perror("Cannot fork");
}

/* stage 3 */

if ((child3=fork())==0) {
    close(0); /* Переназначить стандартный ввод */
    dup(pipe2[0]);
    close(pipe2[0]);
    /* Исполнить программу */
    execlp("tail", "tail", "-l", NULL);
    /* Мы можем попасть сюда только при ошибке exec */;
    perror("Cannot exec");
    exit(0);
}
close(pipe2[0]);
if (child3== -1) {
    perror("Cannot fork");
}
while(wait(NULL)!= -1);
return;
}
```

Понятно, что такие трубы можно использовать только для связи родственных задач, т. е. таких, которые связаны отношением родитель-потомок или являются потомками одного процесса.

Для связи между неродственными задачами используется другое средство — *именованные трубы* (named pipes) в System V и UNIX domain sockets в BSD UNIX. В разных системах именованные трубы создаются различными системными вызовами, но очень похожи по свойствам, поэтому стандарт POSIX предлагает для создания именованных труб библиотечную функцию `mkfifo(const char * name, mode_t flags)`. Эта функция создает специальный файл. Открывая такой файл, программа получает доступ к одному из концов трубы. Когда две программы откроют именованную трубу, они смогут использовать ее для обмена данными точно так же, как и обычную.

Современные системы семейства Unix предоставляют возможность для одновременной работы с несколькими трубами (а также с другими объектами, описываемыми дескриптором файла — собственно файлами, сокетами и т. д.) — системный вызов `select`. Этот вызов возвращает список дескрипторов файлов, которые способны передать или принять данные. Если ни один из дескрипторов не готов к обмену данными, `select` блокируется.

Трубы широко используются системами семейства Unix, и они внесены в стандарт POSIX. Ряд операционных систем, не входящих в семейство Unix, например VxWorks, также предоставляют этот сервис.

Почтовые ящики VMS

Система VMS предоставляет средства, отчасти аналогичные трубам, называемые почтовые ящики (*mailbox*). Почтовый ящик также представляет собой кольцевой буфер, доступ к которому осуществляется теми же системными вызовами, что и работа с внешним устройством. Системная библиотека языка VAX C использует почтовые ящики для реализации труб, в основном совместимые с UNIX и стандартом POSIX. Широко используемый сервис сетевой передачи данных, сокеты протокола TCP, также очень похожи на трубу.

Линки транспьютера

В процессорах семейства Inmos Transputer микропрограммно реализованы линки (*link* — связь) — синхронный примитив, отчасти похожий на трубы.

Линки бывают двух типов — физические и логические. Операции над линками обоих типов осуществляются одними и теми же командами. Физический линк представляет собой последовательный интерфейс RS432, реализованный на кристалле процессора. С линком также ассоциировано одно слово памяти, смысл которого будет объяснен далее.

Современные транспьютеры имеют четыре физических линка. Физические линки могут передавать данные со скоростью до 20 Мбит/с и могут использоваться как для соединения транспьютеров между собой (рис. 7.7), так и для подключения внешних устройств. Благодаря этому физический линк может использоваться как для связи между процессами на разных транспьютерах, так и для синхронизации процесса с внешними событиями и даже просто для ввода-вывода.

Логический линк — это просто структура данных, выделенная в физическом адресном пространстве процессора. С точки зрения программы, физический и логический линки ничем не отличаются, кроме того, что описатель физического линка привязан к определенному физическому адресу. Логический линк может использоваться только для связи между процессами (напоминаем, что по принятой в транспьютере терминологии, нити называются процессами), исполняющимися на одном транспьютере.

Транспьютер T9000 предоставляет также виртуальный линк — протокол, позволяющий двум транспьютерам организовать несколько линий взаимодействия через один физический линк, или даже через цепочку маршрутизаторов.

При передаче данных в линк процесс должен исполнить команду `out`. Эта команда имеет три операнда: адрес линка, адрес массива данных и количество данных. Для передачи operandов используется регистровый стек процессора. Процесс, исполнивший такую команду, задерживается до тех пор, пока все данные не будут переданы (рис. 7.8).

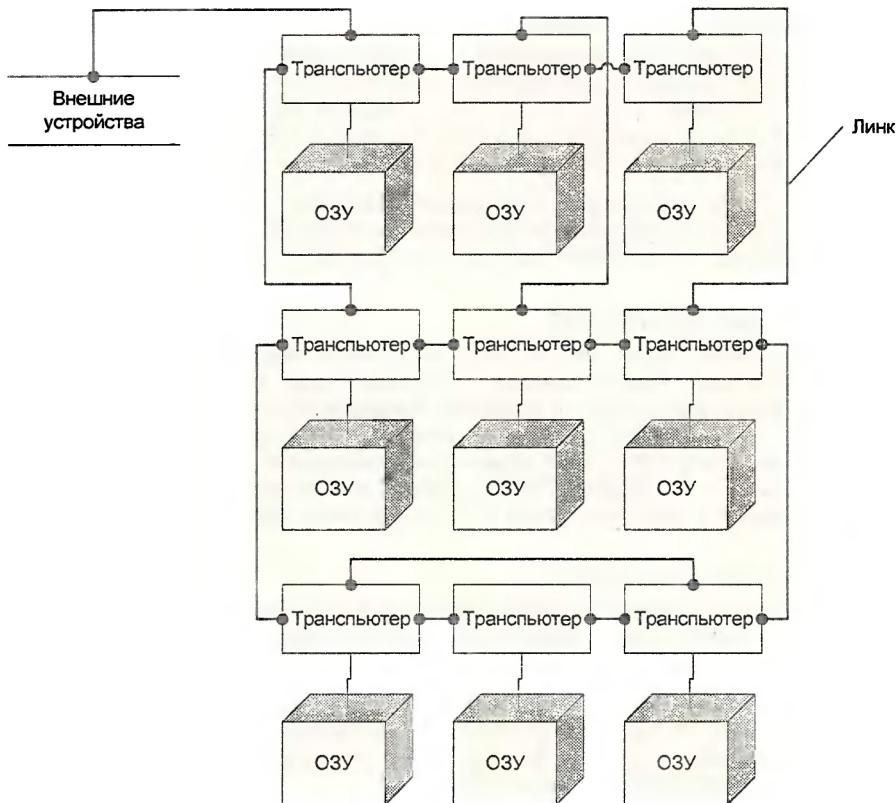


Рис. 7.7. Сеть транспьютеров, соединенных физическими линками

Аналогично, при приеме данных из линка, процесс должен исполнить команду `iP`. Эта команда также имеет три операнда — адрес линка, адрес буфера, куда необходимо поместить данные, и размер буфера. При исполнении такой команды процесс блокируется до тех пор, пока буфер не будет заполнен данными. При этом приемник и передатчик могут использовать буфера разного размера, т. е. приемник может считывать большой массив данных в несколько приемов и т. д.

Существует также команда `alt`, позволяющая процессу ожидать данные из нескольких линков одновременно. В качестве одного из ожидаемых событий можно также использовать сигнал от системного таймера. Слово, связанное с линком, содержит указатель на дескриптор процесса, ожидающего приема или передачи данных через линк. Кроме того, это слово может принимать значение `NotProcessP`, указывающее, что соединения никто не ждет. Остальная информация, такая, как указатель на буфер и размер буфера, хранится в дескрипторе процесса.

Направление передачи данных определяется командой, которую исполнит очередной процесс при обращении к линку. Например, если исполняется команда `out`, предназначенные для записи данные копируются в буфер ожидающего

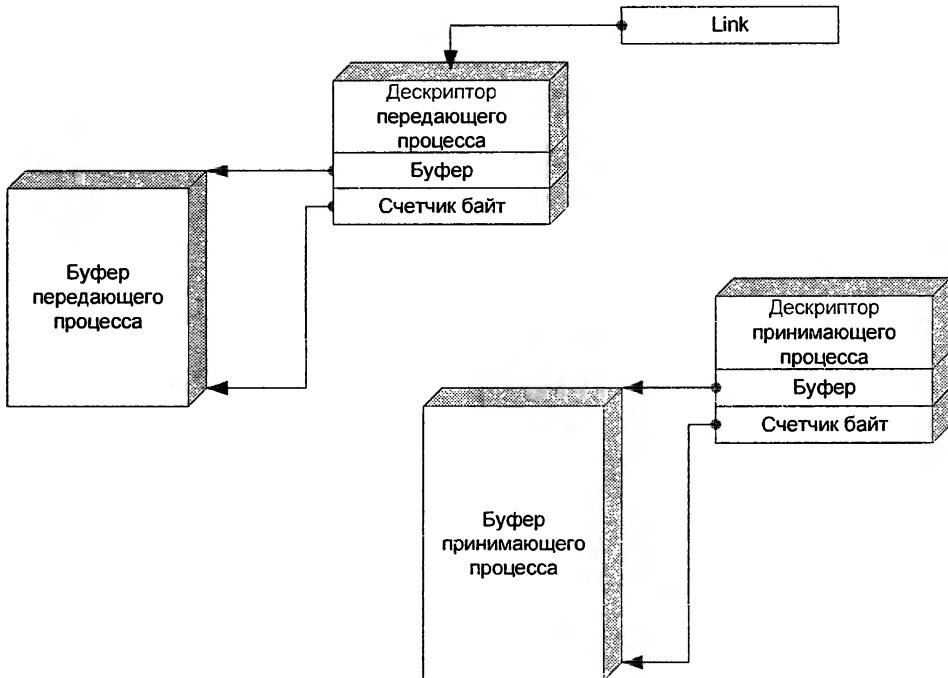


Рис. 7.8. Передача данных через линк

процесса. При этом указатель буфера продвигается, а счетчик размера уменьшается на количество скопированных данных. Если же в линке записано значение NotProcessP, процесс переводится в состояние ожидания и указатель на его дескриптор помещается в линк (рис. 7.9).

Аналогично обрабатываются запросы на чтение. Если мы имеем более двух процессов, пытающихся использовать один линк, то возникает серьезная проблема: внимательный читатель должен был заметить, что мы не сказали, где хранится информация о том, чего ожидает текущий процесс: чтения или записи. Проблема состоит в том, что эта информация нигде не хранится. Если процесс попытается записать данные в линк, на котором кто-то уже ожидает записи, то данные второго процесса будут записаны поверх данных ожидавшего. Если размеры буферов совпадут, то ожидавший процесс будет пребывать в убеждении, что он успешно передал все данные. Поэтому линки рекомендуется использовать только для односторонней передачи данных между двумя (не более!) процессами.

При работе с физическим линком данные не копируются, а передаются или принимаются через физический канал в режиме прямого доступа к памяти. Если на другом конце линка находится другой транспьютер, это все-таки можно считать копированием, но к линку может быть подключено и какое-то другое устройство.

В середине 90-х годов XX века, в эпоху расцвета микропроцессоров этого семейства, фирма Inmos поставляла широкий набор трэмов (trem — TTransputer

Extension Module) — устройств ввода-вывода с линком в качестве интерфейса. В частности, поставлялись трэмы, позволявшие подключить к транспьютеру через линк адаптеры Ethernet или SCSI.

Взаимодействие с внешним устройством через линк позволяет транспьютеру синхронизовать свою деятельность с этими устройствами без использования механизма прерываний. В [INMOS 72 TRN 203 02] приводится пример программной имитации векторных прерываний с передачей вектора по линку и мониторным процессом, который принимает эти векторы из линка и вызывает соответствующие обработчики.

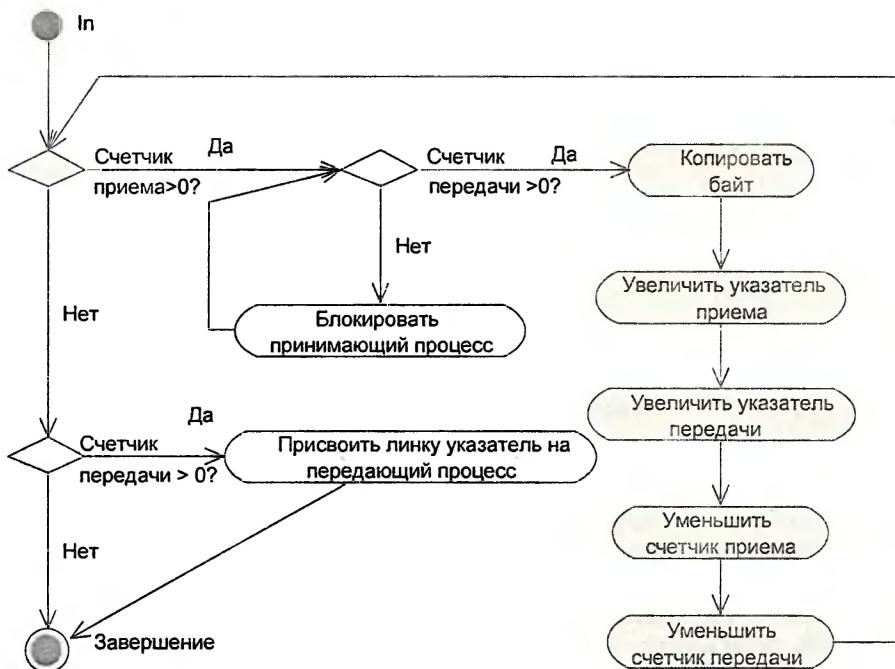


Рис. 7.9. Алгоритм работы команд `in` и `out`

7.5. Системы, управляемые событиями

В начале 70-х годов XX века появилась новая архитектура многозадачных систем, довольно резко отличающаяся от вышеописанной модели последовательных процессов. Речь идет о так называемых *системах, управляемых событиями* (*event-driven systems*).

На первый взгляд, концепция систем, управляемых событиями, близко родственна гармонически взаимодействующим процессам. Во всяком случае, одно из ключевых понятий этой архитектуры, очередь событий, мы упоминали в

числе средств гармонического межпоточного взаимодействия. Различие между этими архитектурами состоит, скорее, во взгляде на то, что представляет собой программа.

В модели гармонически взаимодействующих потоков процесс исполнения программного комплекса представляет собой совокупность взаимодействующих нитей управления. В системе, управляемой событиями, программа представляет собой совокупность объектов, обменивающихся *сообщениями* о событиях, а также реагирующих на сообщения, приходящие из внешних источников.

В идеале, объекты взаимодействуют между собой только через сообщения. Приходящие сообщения побуждают объект изменить свое состояние и, возможно, породить некоторое количество сообщений, предназначенных для других объектов. При такой модели взаимодействия нам неважно, исполняются ли методы объектов как параллельные (или псевдопараллельные) нити, или же последовательно вызываются единой нитью, менеджером или диспетчером сообщений.

Несколько забегая вперед, можно описать отличие следующим образом: гармоническая нить, ожидающая сообщение в очереди, сохраняет свой контекст (понятие контекста нити определяется в следующей главе). Нить обработчика событий, ожидающая сообщения, не имеет контекста как такового. На практике, в событийно-ориентированных системах происходит переиспользование контекста, т. е. обработчики разных событий попеременно исполняются в одной и той же нити или нескольких разных нитях.

Первоначально событийно-ориентированная архитектура применялась в ядрах ОС, в первую очередь при реализации подсистемы ввода-вывода (см. гл. 10). Драйвер устройства при такой архитектуре представляет собой обработчик двух потоков событий — запросов ввода-вывода, поступающих от пользовательских программ или других модулей ядра (например, от драйвера файловой системы), и прерываний от устройства. Благодаря этому, несколько драйверов могут обрабатывать несколько потоков событий от разных устройств, не мешая друг другу и не требуя полноценного вытесняющего планировщика (см. разд. 8.2).

Ряд ОС — главным образом, это системы реального времени, но есть и ОС разделенного времени, например RSX-11 и VAX/VMS — выносит событийно-ориентированную архитектуру на уровень системных вызовов, указывая в описании системного вызова, что запрашиваемая операция может не быть исполнена к моменту возврата управления прикладной программе. Преимущества, которые это дает, мы в полной мере увидим в разд. 8.2.3. Обычно такие ОС позволяют связывать с запросами нечто вроде семафора (в RSX-11

и VAX/VMS — локальные флаги событий) и затем отслеживать статус каждого из запросов. При этом программа продолжает исполняться как последовательный процесс, но знает, что ядро, на самом деле, событийно-ориентированное.

Впервые успешная попытка вынести событийную ориентацию на уровень пользовательских программ была реализована в экспериментальных настольных компьютерах Alto, разработанных в 1973 году в исследовательском центре PARC фирмы Xerox. Целью эксперимента было создание операционной среды, удобной для разработки интерактивных программ с динамичным пользовательским интерфейсом.

В этих системах впервые была реализована многооконная графика, когда пользователь одновременно видит на экране графический вывод нескольких программ и может активизировать любую из них, указав на соответствующее окно манипулятором-мышью.

При каждом движении мыши, нажатии на ее кнопки или клавиши на клавиатуре генерируется событие. События могут также генерироваться системным таймером или пользовательскими программами. Нельзя не упомянуть "визуальные" события, которые порождаются в ситуации, когда пользователь сдвинул или закрыл одно из окон и открыл при этом часть окна, находившегося внизу. Этому окну посыпается событие, говорящее о том, что ему нужно перерисовать часть себя (рис. 7.10).

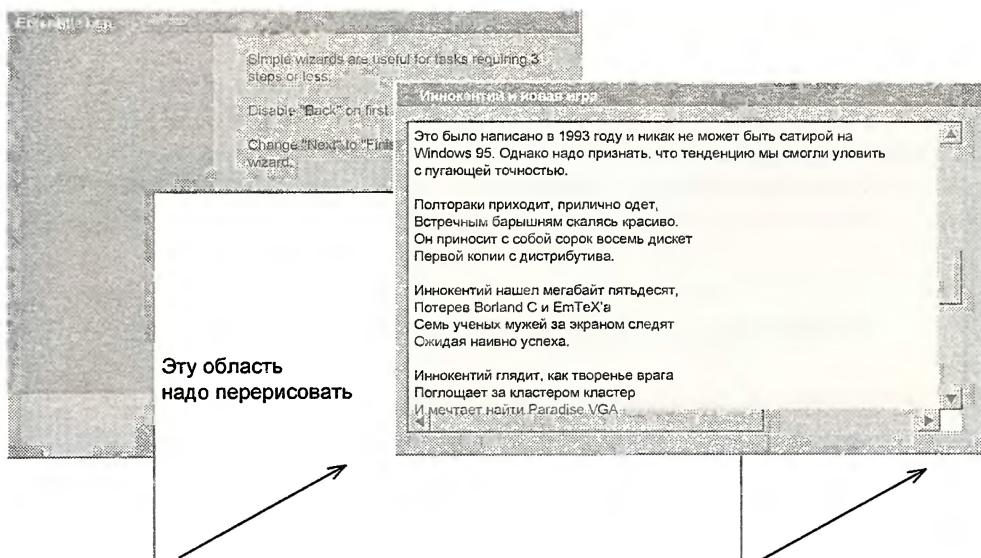


Рис. 7.10. Визуальное событие

Каждое сообщение о событии представляет собой структуру данных, которая содержит код, обозначающий тип события: движение мыши, нажатие кнопки и т. д., а также поля, различные для разных типов событий. Для "мышиных" событий — это текущие координаты мыши и битовая маска, обозначающая состояние кнопок (нажата/отпущена). Для клавиатурных событий — это код нажатой клавиши (обычно ASCII-код символа для алфавитно-цифровых и специальные коды для стрелок и других "расширенных" и "функциональных" клавиш) и битовая маска, обозначающая состояние различных модификаторов, таких как `<Shift>`, `<Ctrl>`, `<Alt>` и т. д. Для визуальных событий — это координаты прямоугольника, который нужно перерисовать. Для командных событий (передающихся от одного обработчика событий к другому) — это параметры команды. И так далее.

Все сообщения о событиях помещаются в очередь в порядке их возникновения.

В системе существует понятие *обработчика событий*. Обработчик событий представляет собой объект, т. е. структуру данных, с которой связано несколько подпрограмм — методов. Один из методов вызывается при поступлении сообщения. Обычно он также называется обработчиком событий. Некоторые системы предлагают объектам-обработчикам предоставлять различные методы для обработки различных событий — например, метод `onClick` будет вызываться, когда придет событие, сигнализирующее о том, что кнопка мыши была нажата и отпущена, когда курсор находился над областью, занимаемой объектом на экране.

Рассмотрим объект графического интерфейса, например меню. При нажатии кнопки мыши в области этого меню вызывается обработчик события. Он разбирается, какой из пунктов меню был выбран, и посыпает соответствующее командное сообщение объекту, с которым ассоциировано меню. Этот объект, в свою очередь, может послать командные сообщения каким-то другим объектам. Например, если была выбрана команда `File | Open`, меню передаст обработчику основного окна приложения сообщение `FILEOPEN`, а тот, в свою очередь, может передать команду `Open` объекту, отвечающему за отрисовку и обработку файлового диалога (рис. 7.11).

Таким образом, вместо последовательно исполняющейся программы, время от времени вызывающей систему для исполнения той или иной функции, мы получаем набор обработчиков,ываемых системой в соответствии с желаниями пользователя. Каждый отдельный обработчик представляет собой конечный автомат, иногда даже вырожденный, не имеющий переменной состояния. Код обработчика и по реализации обычно похож на конечный автомат, он состоит из большого оператора `switch`, выбирающего различные действия в зависимости от типа пришедшего сообщения (пример 7.9).

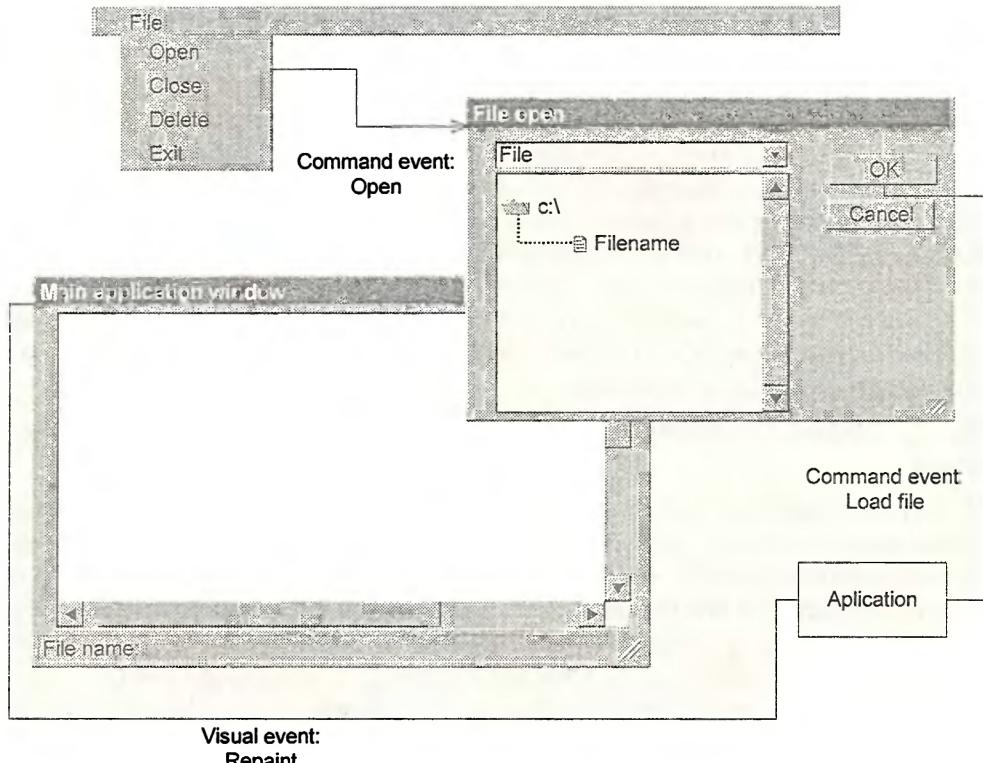


Рис. 7.11. Обмен событиями между элементами пользовательского интерфейса

Пример 7.9. Обработчик оконных событий в OS/2 Presentation Manager

```
/*
 * Фрагмент примера из поставки IBM Visual Age for C++ 3.0.
 * Обработчик событий меню предоставается системой,
 * а обработку командных событий, порождаемых меню,
 * вынужден брать на себя разработчик приложения. */
*****  

*  

* Copyright (C) 1992 IBM Corporation  

*  

* ОТКАЗ ОТ ГАРАНТИЙ. Следующий код представляет собой пример кода
* созданный IBM Corporation. Этот пример кода не является частью ни
* одного стандарта или продукта IBM и предоставляется вам с
* единственной целью – помочь в разработке ваших приложений.
* Код предоставляется "КАК ЕСТЬ", без каких-либо гарантий.
*/
```

* IBM не несет ответственности за какие бы то ни было повреждения,
* возникшие в результате использования вами этого кода, даже если
* она и могла предполагать возможность таких повреждений.
*

```
*****  
* Имя: MainWndProc  
*  
* Описание: Оконная процедура главного окна клиента.  
*  
* Концепции: Обрабатывает сообщения, посыпаемые главному  
* окну клиента. Эта процедура обрабатывает основные  
* сообщения, которые должны обрабатывать все клиентские  
* окна, и передает все остальные [функции] UserWndProc,  
* в которой разработчик может обработать любые другие  
* сообщения.  
*  
* API: Не используются  
*  
* Параметры: hwnd – идентификатор окна, которому адресовано сообщение  
* msg – тип сообщения  
* mp1 – первый параметр сообщения  
* mp2 – второй параметр сообщения  
*  
* Возвращаемое значение: определяется типом сообщения  
*  
*****
```

MRESULT EXPENTRY MainWndProc(HWND hwnd, USHORT msg, MPARAM mp1,
MPARAM mp2)

```
{  
    switch(msg)  
    {  
        case WM_CREATE:  
            return(InitMainWindow(hwnd, mp1, mp2));  
            break;  
  
        case WM_PAINT:  
            MainPaint(hwnd);  
            break;  
    }  
}
```

```
case WM_COMMAND:
    MainCommand(mp1, mp2);
    break;

case WM_INITMENU:
    InitMenu(mp1, mp2);
    break;

case HM_QUERY_KEYS_HELP:
    return (MRESULT) PANEL_HELPKEYS; /* Вернуть Id панели подсказки */
    break;

/*
 * Все необработанные сообщения передаются
 * пользовательской процедуре окна.
 * Она отвечает за передачу всех необработанных
 * сообщений функции WinDefWindowProc();
 */
default:
    return(UserWndProc(hwnd, msg, mp1, mp2));
    break;

}

return (MRESULT) 0; /* Все оконные процедуры должны по умолчанию
возвращать 0 */

} /* Конец MainWndProc() */
```

```
*****\n* Имя:      MainCommand\n*\n* Назначение: Главная процедура окна, обрабатывающая WM_COMMAND\n*\n* Концепции: Процедура вызывается, когда сообщение WM_COMMAND\n*               отправляется главному окну. Оператор switch\n*               переключается в зависимости от id меню, которое\n*               породило сообщение и предпринимает действия,\n*               соответствующие этому пункту меню. Все id меню,\n*               не являющиеся частью стандартного набора команд,\n*               передаются пользовательской процедуре обработки\n*               WM_COMMAND.
```

```
*  
* API : WinPostMsg  
*  
* Параметры : mp1 - первый параметр сообщения  
*               mp2 - второй параметр сообщения  
*  
* Возвращает: VOID  
*  
/**************************************************************************/  
VOID MainCommand(MPARAM mp1, MPARAM mp2)  
{  
    switch(SHORT1FROMMP(mp1))  
    {  
        case IDM_EXIT:  
            WinPostMsg( hwndMain, WM_QUIT, NULL, NULL );  
            break;  
  
        case IDM_FILENEW:  
            FileNew(mp2);  
            break;  
  
        case IDM_FILEOPEN:  
            FileOpen(mp2);  
            break;  
  
        case IDM_FILESOLVE:  
            FileSave(mp2);  
            break;  
  
        case IDM_FILESAVEAS:  
            FileSaveAs(mp2);  
            break;  
  
        case IDM_EDITUNDO:  
            EditUndo(mp2);  
            break;  
  
        case IDM_EDITCUT:  
            EditCut(mp2);  
            break;  
    }  
}
```

```
case IDM_EDITCOPY:
    EditCopy(mp2);
    break;

case IDM_EDITPASTE:
    EditPaste(mp2);
    break;

case IDM_EDITCLEAR:
    EditClear(mp2);
    break;

case IDM_HELPUSINGHELP:
    HelpUsingHelp(mp2);
    break;

case IDM_HELPGENERAL:
    HelpGeneral(mp2);
    break;

case IDM_HELPKEYS:
    HelpKeys(mp2);
    break;

case IDM_HELPINDEX:
    HelpIndex(mp2);
    break;

case IDM_HELPPRODINFO:
    HelpProdInfo(mp2);
    break;
/*
 * Здесь вызывается пользовательская процедура
 * обработки команд, чтобы обработать те id,
 * которые еще не были обработаны.
 */
default:
    UserCommand(mp1, mp2);
    break;
}
} /* MainCommand()
*/
```

Специальная нить, *менеджер событий* (рис. 7.12), просматривает очередь и передает поступающие события обработчикам. События, связанные с экранными координатами, передаются обработчику, ассоциированному с соответствующим окном. Клавиатурные события передаются фокусу клавиатуры — текущему активному обработчику клавиатуры.

Управление событиями позволяет относительно легко разрабатывать динамичные пользовательские интерфейсы, привычные для пользователей современных графических оболочек.

Высокая динамичность интерфейса проще всего обеспечивается, если каждый обработчик быстро завершается. Если же в силу природы запрашиваемой операции она не может завершиться быстро — например, если мы вставили символ, параграф удлинился на строку, и в результате текстовому процессору типа WYSIWYG приходится переформатировать и переразбивать на страницы весь последующий текст — мы можем столкнуться с проблемой.

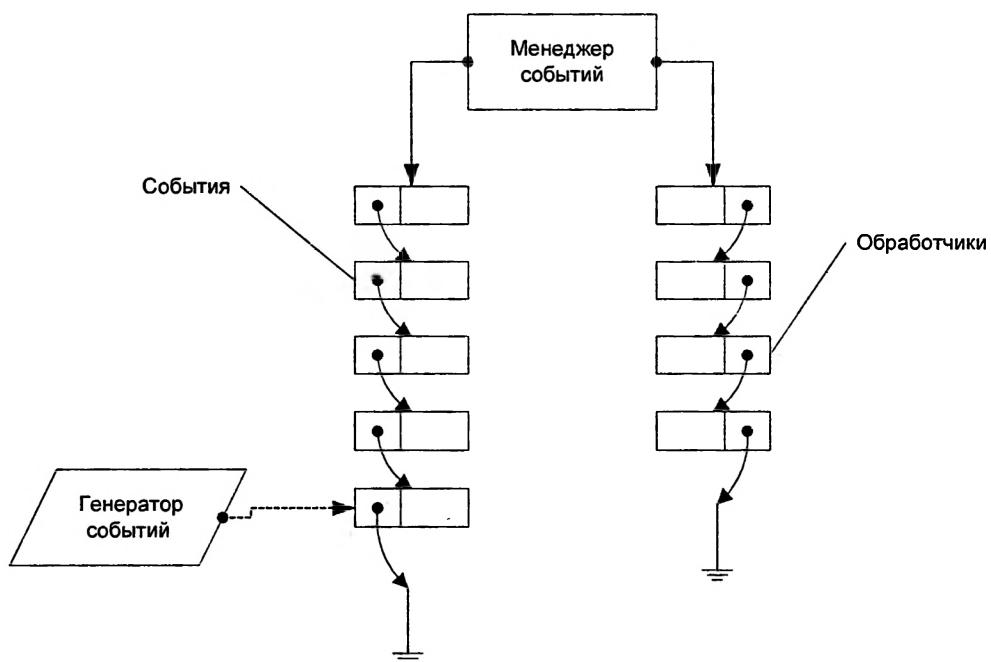


Рис. 7.12. Менеджер и обработчики событий

В такой ситуации (а при написании реальных приложений она возникает сплошь и рядом) мы и вынуждены задуматься о том, что же в действительности представляют собою обработчики — процедуры, синхронно вызываемые единственной нитью менеджера событий, или параллельно исполняющиеся

нити. Первая стратегия называется *синхронной обработкой сообщений*, а вторая, соответственно, — *асинхронной*.

Графические интерфейсы первого поколения — Mac OS, Win16 — реализовывали синхронную обработку сообщений, а когда обработчик задумывался надолго, рисовали неотъемлемый атрибут этих систем — курсор мыши в форме песочных часов.

Несколько более совершенную архитектуру предлагает оконная подсистема OS/2, Presentation Manager. PM также реализует синхронную стратегию обработки сообщений (менеджер событий всегда ждет завершения очередного обработчика), но в системе, помимо менеджера событий, могут существовать и другие нити. Если обработка события требует длительных вычислений или других действий (например, обращения к внешним устройствам или к сети), рекомендуется создать для этого отдельную нить и продолжить обработку асинхронно. Если же приложение этого не сделает (например, обработчик события просто зациклится или заснет на семафоре), системная очередь сообщений будет заблокирована и ни одно из графических приложений не сможет работать. Современные версии PM предоставляют в этом случае возможность отцепить "ненормальное" приложение от очереди или даже принудительно завершить его.

Асинхронные очереди сообщений предоставляют Win32 и оконная система X Window. Впрочем, и при асинхронной очереди впавший в философские размышления однопоточный обработчик событий — тоже малоприятное зрелище, ведь он не может перерисовать собственное окно, поэтому передвижение других окон по экрану порождает любопытные спецэффекты (к сожалению, запечатлеть эти эффекты с помощью утилит сохранения экрана невозможно — все известные автору средства дожидаются, пока все попавшие в сохраняемую область окна перерисуются; а фотографии монитора обычно имеют низкое качество...). Разработчикам приложений для названных систем также рекомендуется выносить длительные вычисления в отдельные нити.

Большинство реальных приложений для современных ОС, обладающих графическим пользовательским интерфейсом, таким образом, имеют двух- или более слойную архитектуру. При этом архитектура ближайшего к пользователю *слоя (frontend)*, как правило, тяготеет к событийно-управляемой, а следующие слои (*backend*) обычно состоят из более традиционных взаимодействующих (не всегда, впрочем, строго гармонически) параллельно исполняющихся нитей, зачастую даже разнесенных по разным вычислительным системам.

Еще один пример использования событийно-ориентированной архитектуры можно увидеть в серверных приложениях. Серверы баз данных или Web-серверы могут обслуживать сотни тысяч соединений одновременно. В таких

приложениях создавать отдельную нить для каждого клиентского соединения оказывается недопустимо дорого. Так, в системах семейства Unix для x86 количество нитей в одном процессе ограничено количеством дескрипторов в LDT, т. е. 8192 штуками (для каждой нити создается дескриптор сегмента, используемый для хранения указателя на контекст нити и приватные данные нити). В Linux каждая нить имеет собственный идентификатор процесса. Идентификатор процесса представляет собой 16-битное знаковое целое число, так что общее количество нитей в Linux не может превосходить 32 767. Кроме того, каждая нить должна иметь свой стек, объем которого по умолчанию составляет 1 Мбайт, минимальный же объем стека в том же Linux составляет 16 384 байт.

Событийно-ориентированная архитектура может резко сократить требуемое количество нитей. Действительно, как мы видели в начале раздела, в событийно-ориентированной архитектуре, нить, ждущая события, может не иметь собственного контекста. На практике происходит переиспользование контекстов.

В простейшем случае одна нить последовательно используется для обработки разных клиентских соединений. Такая архитектура называется *пулом нитей* (thread pool) или, если вместо нитей используются задачи, заранее созданными процессами (preforked processes). Она позволяет сэкономить время центрального процессора, необходимое для создания и уничтожения нити или процесса и происходящего при этом перераспределения памяти и других ресурсов. Однако количество одновременно обслуживаемых клиентских соединений оказывается ограничено количеством нитей в пуле.

Более сложная архитектура предполагает, что нити обрабатывают разные запросы в пределах разных клиентских соединений. С точки зрения логики обработки отдельного запроса, это чистая событийно-ориентированная архитектура. С каждой клиентской сессией оказывается связан блок переменных состояния (нередко реализованный в виде объекта C++ или другого объектно-ориентированного языка). Обработка каждого отдельного запроса реализуется в виде вызова методов этого объекта; причем методы вызываются последовательно, но не обязательно в рамках одной и той же нити. Даже когда методы вызываются в одной и той же нити, между этими вызовами могут произойти вызовы методов других объектов, связанные с обработкой запросов в сессиях других клиентов.

Такая архитектура называется рабочими нитями (worker thread). С точки зрения нитей ее можно описать следующим образом: рабочие нити запрашивают некий центральный диспетчер, есть ли работа (т. е. есть ли необработанный запрос в любом из клиентских соединений). Если работы есть, нить захватывает соответствующий запрос и приступает к его обработке. Если работы нет, нить блокируется.

Частным случаем рабочих нитей может считаться однопоточное событийно-ориентированное приложение.

Некоторые из реализаций рабочих нитей допускают динамическое создание и уничтожение нитей — если количество нитей, ждущих работу, превосходит определенный порог, то некоторые из них уничтожаются; напротив, если есть работа, а все нити заняты, может быть создана дополнительная нить. Впрочем, такие реализации не очень популярны.

Действительно, поскольку создание и уничтожение нити — дорогая операция, то уничтожать нити для того, чтобы потом их вновь создать, невыгодно. С другой стороны, если система не успевает обрабатывать поступающие запросы, далеко не всегда это обусловлено нехваткой рабочих нитей. Часто это может быть связано с нехваткой памяти, недостаточной пропускной способностью жесткого диска или — для многослойных приложений, использующих СУБД — с перегрузкой сервера базы данных. В этих условиях создание дополнительных нитей не решит проблемы производительности, а в случае нехватки памяти — только ухудшит производительность.

В сетевых серверах для систем семейства Unix диспетчер может быть реализован на основе системных вызовов `select` или `poll` (одноименные и функционально аналогичные процедуры доступны также в Win32 в составе Winsock API).

Архитектура рабочих нитей используется во многих серверных приложениях — в большинстве серверов баз данных, в веб-сервере Apache 2.0, в почтовом сервере Sendmail и др.

Архитектуру подсистемы ввода-вывода большинства современных ОС также можно описать как рабочие нити. Количество рабочих нитей при этом соответствует количеству процессоров; на однопроцессорной машине мы получаем вырожденный случай архитектуры с единственной рабочей нитью.

Вопросы для самопроверки

1. Приведите примеры алгоритмов, полагающихся на целостность структур данных, с которыми этот алгоритм работает.
2. Попробуйте привести примеры алгоритмов, которые не полагаются на целостность структур данных (т. е. могут работать с данными, находящимися в произвольном состоянии).
3. Что такое ошибка соревнования?
4. Что такое критическая секция?
5. Что такое реентерабельная процедура? Можно ли сделать произвольную процедуру реентерабельной, и если да, то как?

6. Попробуйте реализовать алгоритм Деккера, пригодный для разделения ресурса между тремя нитями.
7. Почему для реализации спинлока требуется аппаратная поддержка?
8. Чем спинлок отличается от семафора Дейкстры?
9. Чем семафор Дейкстры общего вида (семафор-счетчик) отличается от мутекса?
10. Почему задача производитель-потребитель разрешима на двух семафорах общего вида, но неразрешима на двух мутексах? Постарайтесь сформулировать ответ как объяснение, какие из пунктов доказательства неразрешимости задачи на двух мутексах, приводимого в *разд. 7.3.1*, неприменимы к семафорам.
11. Какие способы предотвращения мертвых блокировок вы знаете? Почему ни один из этих способов не применим универсально?
12. Почему линки транспьютера названы небуферизованным (синхронным) примитивом, а трубы в системах семейства Unix — буферизованным (асинхронным)? Как вообще в данном случае буферизация связана с асинхронностью?
13. Чем событийно-ориентированная программа отличается от многопоточной программы, использующей гармоническое межпроцессное взаимодействие, например те же самые очереди сообщений?
14. Каким образом многослойная реализация приложения помогает разрешить или хотя бы контролировать проблемы разработки многопоточных приложений?
15. Если у вас есть опыт настройки какого-либо серверного приложения, попробуйте понять по документации и/или наблюдая за работой приложения, использует ли оно пул потоков или рабочие нити, и ответить на следующие вопросы.

Каким образом настраивается количество нитей? Используется ли динамический подбор количества нитей и если да, то есть ли ограничения сверху и снизу на количество нитей? Исполняются эти нити в рамках одного процесса, или для каждой нити создается свой процесс, или же создается группа процессов, в каждом из которых создается несколько нитей? Как, по-вашему, из каких соображений была выбрана та, а не иная архитектура? Из каких соображений рекомендуется определять количество нитей? Как слишком большое или слишком малое количество нитей влияет на производительность? Можете ли вы заметить влияние количества нитей на производительность при тех нагрузках, под которыми работает ваш сервер?

В Windows количество нитей для каждой задачи можно увидеть в Task Manager, на вкладке **Processes** (для этого необходимо выбрать соответствующую колонку через меню **View**). Впрочем, некоторые приложения используют пользовательские нити, так называемые "волокна" (*fibers*), количество которых практически невозможно определить, не используя отладчик.

В Linux каждая нить имеет собственный идентификатор процесса и видна как отдельная запись в каталоге `/proc` и в выводе команд `ps`, `top` и др.; принадлежность нитей к одному процессу может быть определена по выводу команды `ps -L`.

В Solaris и других системах, использующих двухуровневую реализацию нитей (SVR4, AIX, HP/UX), легко определить количество системных нитей, так называемый LWP (Light-Weight Process — легковесный процесс). В Solaris каждому LWP соответствует подкаталог в каталоге процесса в `/proc`; количество LWP для каждого процесса может быть получено командой `ps -f`, а список — командой `ps -L`. В старых версиях Solaris количество пользовательских нитей может быть определено только анализом адресного пространства задачи, например с помощью отладчика. Любители ковыряться в двоичных данных могут попытаться определить его просмотром адресного пространства задачи, которое доступно в виде файла в псевдофайловой системе `/proc`. В Solaris 10 создается один LWP на каждую пользовательскую нить, поэтому определить количество нитей в процессе просто.



ГЛАВА 8

Реализация многозадачности на однопроцессорных компьютерах

Теперь у нее есть дочь —
Другое поколение, другие дела;
Ей только пять лет, но время летит, как стрела;
И хотя она пока что не умеет читать,
Она уже знает больше, чем знала мать,
Ведь она видит сразу много программ,
Глядя в телевизор...

Б. Гребеников

В предыдущей главе мы упоминали о возможности реализовать параллельное (или, точнее, псевдопараллельное) исполнение нескольких потоков управления на одном процессоре. Понятно, что такая возможность дает значительные преимущества. В частности, это позволяет разрабатывать прикладные программы, которые могут исполняться без переделок и часто даже без перенастроек и на одно-, и на симметричных многопроцессорных машинах. Кроме того, *многопоточность* полезна и сама по себе, хотя и сопряжена с определенными неудобствами (перечисленными в предыдущей главе) при реализации взаимодействия параллельных нитей.

Примечание

Внимательный читатель может обратить внимание на некоторую терминологическую непоследовательность, появляющуюся в этой главе. В соответствии с принятой в главе 3 терминологией, правильно было бы говорить о *многонитности* (дословный перевод английского термина *multithreading*), но это слово, хотя и состоит только из славянских корней, звучит очень уж не по-русски, термин же *многозадачность* прижился в компьютерной лексике давно иочно поэтому мы будем его употреблять наравне с правильным термином *многопоточность*.

8.1. Кооперативная многозадачность

По-видимому, самой простой реализацией многозадачной системы была бы библиотека подпрограмм, которая определяет следующие процедуры.

`struct Thread;`

В тексте будет обсуждаться, что должна представлять собой эта структура, называемая *дескриптором нити*.

`Thread * ThreadCreate(void (*ThreadBody)(void));`

Создать нить, исполняющую функцию `ThreadBody`.

`void ThreadSwitch();`

Эта функция приостанавливает текущую нить и активизирует очередную, готовую к исполнению.

`void ThreadExit();`

Прекращает исполнение текущей нити.

Сейчас мы не обсуждаем методов синхронизации нитей и взаимодействия между ними (для синхронизации были бы полезны также функции `void DeactivateThread();` и `void ActivateThread(struct Thread *);`). Нас интересует только вопрос: что же мы должны сделать, чтобы переключить нити?

Функция `ThreadSwitch` называется *диспетчером* или *планировщиком* (*scheduler*) и ведет себя следующим образом.

1. Она передает управление на следующую активную нить.
2. Текущая нить остается активна, и через некоторое время снова получит управление.
3. При этом она получит управление так, как будто `ThreadSwitch` представляла собой обычную функцию и возвратила управление в точку, из которой она была вызвана.

Очевидно, что функцию `ThreadSwitch` нельзя реализовать на языке высокого уровня, вроде С, потому что это должна быть функция, которая не возвращает [немедленно] управления в ту точку, из которой она была вызвана. Она вызывается из одной нити, а передает управление в другую. Это требует прямых манипуляций стеком и записью активизации и обычно достигается использованием ассемблера или ассемблерных вставок. Некоторые ЯВУ (Ada, Java, Occam) предоставляют примитивы создания и переключения нитей в виде встроенных функций или специальных синтаксических конструкций.

Самым простым вариантом, казалось бы, будет простая передача управления на новую нить, например, командой безусловной передачи управления по указателю. При этом весь описатель нити (`struct Thread`) будет состоять

только из адреса, на который надо передать управление. Беда только в том, что этот вариант не будет работать.

Действительно, каждая из нитей исполняет программу, состоящую из вложенных вызовов процедур. Для того чтобы нить нормально продолжила исполнение, нам нужно восстановить не только адрес текущей команды, но и стек вызовов (*см. разд. 2.6.4*). Поэтому мы приходим к такой архитектуре:

- каждая нить имеет свой собственный стек вызовов;
- при создании нити выделяется область памяти под стек, и указатель на эту область помещается в дескриптор нити;
- `ThreadSwitch` сохраняет указатель стека (и, если таковой есть, указатель кадра) текущей нити в ее дескрипторе и восстанавливает `SP` из дескриптора следующей активной нити (переключение стеков необходимо реализовать ассемблерной вставкой, потому что языки высокого уровня не предоставляют средств для прямого доступа к указателю стека — (пример 8.1));
- когда функция `ThreadSwitch` выполняет оператор `return`, она автоматически возвращает управление в то место, из которого она была вызвана в этой нити, потому что адрес возврата сохраняется в стеке.

Пример 8.1. Кооперативный переключатель потоков для процессора 8086

```
Thread * thread_queue_head;
Thread * thread_queue_tail;
Thread * current_thread;
Thread * old_thread;

void TaskSwitch() {
    old_thread=current_thread;
    add_to_queue_tail(current_thread);
    current_thread=get_from_queue_head();
    asm {
        move bx, old_thread
        push bp
        move ax, sp
        move thread_sp[bx], ax
        move bx, current_thread
        move ax, thread_sp[bx]
        pop bp
    }
    return;
}
```

Если система программирования предполагает, что при вызове функции должны сохраняться определенные регистры (как, например, С-компиляторы для 8086 сохраняют при вызовах регистры `SI` и `DI` (ESI/EDI в x86)), то они также сохраняются в стеке. Поэтому предложенный нами вариант также будет автоматически сохранять и восстанавливать все необходимые регистры.

Понятно, что кроме указателей стека и стекового кадра, `struct Thread` должна содержать еще некоторые поля. Как минимум, она должна содержать указатель на следующую активную нить. Система должна хранить указатели на описатель текущей нити и на конец списка. При этом `ThreadSwitch` переставляет текущую нить в конец списка, а текущей делает следующую за ней в списке. Все вновь активизируемые нити также ставятся в конец списка. При этом список не обязан быть двунаправленным, ведь мы извлекаем элементы только из начала, а добавляем только в конец.

Часто в литературе такой список называют *очередью нитей* (*thread queue*) или *очередью процессов*. Такая очередь присутствует во всех известных мне реализациях многозадачных систем. Кроме того, очереди нитей используются и при организации очередей ожидания различных событий, например, при реализации семафоров Дейкстры.

Нить в такой системе (как, впрочем, и в более сложных системах, рассматриваемых далее в этой главе) может находиться в одном из трех состояний:

1. Исполнение.
2. Ожидание исполнения.
3. Ожидание события.

Состояние ожидания исполнения соответствует готовой к исполнению нити, которая стоит в очереди активных нитей, пока процессор занят другой нитью.

В однопроцессорной системе исполняющаяся нить может продолжать находиться в голове очереди активных, однако в многопроцессорных системах планировщик вынужден извлекать активную нить из очереди, иначе есть риск, что одну и ту же нить запланируют на нескольких процессорах.

Ожидание события означает, что нить не готова к исполнению. Многие реальные ОС разделяют это состояние на различные классы в зависимости от того, какого именно события ждет нить. Например, в системах семейства Unix есть два типа состояний ожидания, обозначаемые в выводе команды `ps` разными способами. Буквой `s` обозначаются нити, ожидающие завершения блокирующегося системного вызова — это могут быть вызовы ввода/вывода или ожидание на примитивах межпроцессного взаимодействия. Буквой `t` обозначаются нити, ожидающие сигнала `SIGCONT`; в это состояние нити входят при получении сигналов управления заданиями (`SIGSTOP/SIGTSTP`) и при

отладке. Многие ОС также разделяют ожидание операции ввода/вывода и ожидание на примитиве межпроцессного взаимодействия.

Количество одновременно исполняющихся нитей, как легко догадаться, не может превосходить количества процессоров в системе; в однопроцессорной системе такая нить может быть только одна. В условиях, когда ни одна нить не готова к исполнению, система обычно исполняет специализированную задачу, так называемую холостую нить (*idle task* или *idle thread*). В кооперативных системах такая нить должна циклически вызывать `TaskSwitch()` для того, чтобы как можно скорее передать управление первой из нитей, которая перейдет в состояние ожидания исполнения.

В рассматриваемых далее в этой главе вытесняющих системах холостая нить может выполнять простой холостой цикл или команду `halt` в холостом цикле. Второй вариант снижает энергопотребление и нагрев процессора и поэтому предпочтительнее.

Планировщик, основанный на `ThreadSwitch`, т. е. на принципе переключения по инициативе активной нити, используется в ряде экспериментальных и учебных систем. Этот же принцип, называемый *кооперативной многозадачностью*, реализован в библиотеках языков Simula 67 и Modula-2. MS Windows 3.x также имеют средство для организации кооперативного переключения задач — системный вызов `GetNextEvent`.

Часто кооперативные нити называют не нитями, а *сопрограммами* — ведь они вызывают друг друга, подобно подпрограммам. Единственное отличие такого вызова от вызова процедуры состоит в том, что такой вызов не иерархичен — вызванная программа может вновь передать управление исходной и остаться при этом активной.

Основным преимуществом кооперативной многозадачности является простота отладки планировщика. Кроме того, снимаются все коллизии, связанные с критическими секциями и тому подобными трудностями — ведь нить может просто не отдавать никому управления, пока не будет готова к этому.

С другой стороны, кооперативная многозадачность имеет и серьезные недостатки.

Во-первых, необходимость включать в программу вызовы `ThreadSwitch` усложняет программирование вообще и перенос программ из однозадачных или иначе организованных многозадачных систем в частности.

Особенно неприятно требование регулярно вызывать `ThreadSwitch` для вычислительных программ. Чаще всего такие программы исполняют относительно короткий внутренний цикл, скорость работы которого определяет скорость всей программы. Для "плавной" многозадачности необходимо вызывать `ThreadSwitch` из тела этого цикла. Делать вызов на каждом шаге цикла

нецелесообразно, поэтому необходимо будет написать код, похожий на приведенный в примере 8.2.

Пример 8.2. Внутренний цикл программы в кооперативно многозадачной среде

```
int counter; // Переменная-счетчик.  
  
while{condition} {  
  
    // Вызывать ThreadSwitch каждые rate циклов.  
    counter++;  
    if (counter % rate == 0) ThreadSwitch();  
  
    .... // Собственно вычисления  
}
```

Условный оператор и вызов функции во внутреннем цикле сильно усложняют работу оптимизирующими компиляторами и приводят к разрывам конвейера команд, что может очень заметно снизить производительность. Вызов функции на каждом шаге цикла приводит к еще большим накладным расходам и, соответственно, к еще большему замедлению.

Во-вторых, злонамеренная нить может захватить управление и никому не отдавать его. Просто не вызывать ThreadSwitch, и все. Это может произойти не только из-за злых намерений, но и просто по ошибке.

Поэтому такая схема оказывается непригодна для многопользовательских систем и часто не очень удобна для интерактивных однопользовательских.

Почему-то большинство коммерческих программ для Win16, в том числе и поставлявшиеся самой фирмой Microsoft, недостаточно активно использовали вызов GetNextEvent. Вместо этого такие программы монопольно захватывали процессор и рисовали известные всем пользователям этой системы "песочные часы". В это время система никак не реагирует на запросы и другие действия пользователя, кроме нажатия кнопки **RESET** или клавиш **<Ctrl>+<Alt>+**.

В-третьих, кооперативная ОС не может исполняться на симметричной много-процессорной машине, а приложения, написанные в расчете на такую ОС, не могут воспользоваться преимуществами многопроцессорности.

Простой анализ показывает, что кооперативные многозадачные системы пригодны только для учебных проектов или тех ситуаций, когда программисту

на скорую руку необходимо сотворить многозадачное ядро. Вторая ситуация кажется несколько странной — зачем для серьезной работы может потребоваться быстро сделанное ядро, если существует много готовых систем реального времени, а также общедоступных (freeware или public domain) в виде исходных текстов реализаций таких ядер?

8.2. Вытесняющая многозадачность

Все вышесказанное подводит нас к идее вызывать ThreadSwitch не из пользовательской программы, а каким-то иным способом. Скорее всего, это должно происходить по прерываниям. Например, если в результате прерывания система узнает, что завершилась операция ввода-вывода, и высокоприоритетная нить, запросившая эту операцию, может продолжить исполнение, наша система могла бы отнять управление у текущей нити и передать его новой. Наиболее интересные возможности открывает вызов планировщика по сигналам системного таймера. Он позволяет обеспечить равномерное или в каком-либо смысле "справедливое" распределение времени центрального процессора между задачами, например, по такой схеме:

1. Каждой нити выделяется *квант времени*.
2. Если нить не освободила процессор в течение своего кванта, ее снимают и переставляют в конец очереди.

Этот механизм, называемый *time slicing* или, как говорят в современных документах, *time sharing (разделение времени)*, реализован в микрокоде транспьютера и практически во всех современных ОС. Общим названием для всех методов переключения нитей по инициативе системы является термин *вытесняющая (preemptive) многозадачность*. Таким образом, вытесняющая многозадачность противопоставляется кооперативной, в которой переключение происходит только по инициативе самой задачи. Разделение времени является частным случаем вытесняющей многозадачности.

Выбор кванта времени является нетривиальной проблемой. С одной стороны, чрезмерно короткий квант приведет к тому, что большую часть времени система будет заниматься переключением потоков. С другой стороны, в интерактивных системах или системах реального времени слишком большой квант приведет к недопустимо большому времени реакции.

В системе реального времени мы можем объявить нити, которым надо быстро реагировать, высокоприоритетными и на этом успокоиться. Однако нельзя так поступить с интерактивными программами в многопользовательской или потенциально многопользовательской ОС, как UNIX на настольной машине x86 или Sun.

Из психологии восприятия известно, что человек начинает ощущать задержку ответа при величине этой задержки около 100 мс. Поэтому в системах разделенного времени, рассчитанных на интерактивную работу, квант обычно выбирают равным десяткам миллисекунд. В старых системах, ориентированных на пакетную обработку вычислительных задач, таких как ОС ДИСПАК на БЭСМ-6, квант мог достигать десятых долей секунды или даже секунд. Это повышает эффективность системы, но делает невозможной — или, по крайней мере, неудобной — интерактивную работу. Многие современные системы подбирают квант времени динамически для разных классов планирования и приоритетов процесса.

Системы реального времени обычно имеют два *класса планирования* — реального и разделенного времени. Класс планирования, как правило,дается не отдельным нитям, а целиком процессам. Процессы реального времени не прерываются по сигналам таймера и могут быть вытеснены только активизацией более приоритетной нити реального времени. Нити реального времени высочайшего приоритета фактически работают в режиме кооперативной многозадачности. Зато нити процессов разделенного времени вытесняются и друг другом по сигналам таймера, и процессами реального времени по мере их активизации.

Вытесняющая многозадачность имеет много преимуществ, но если мы просто будем вызывать описанный в предыдущем разделе `ThreadSwitch` по прерываниям от таймера или другого внешнего устройства, то такое переключение будет непоправимо нарушать работу прерываемых нитей.

Действительно, пользовательская программа может использовать какой-то из регистров, который не сохраняется при обычных вызовах. В кооперативно многозадачной системе прерываемая программа знает, что происходит что-то необычное (правда, в реализации, которую мы рассматривали в предыдущем разделе, она думает, что происходит обычный вызов процедуры) и сохраняет регистры; в вытесняющей же системе вызовы планировщика происходят неконтролируемо для программы.

Поэтому, например, обработчики аппаратных прерываний сохраняют в стеке все используемые ими регистры. Кстати, если наша функция `ThreadSwitch` будет сохранять в стеке все регистры, то произойдет именно то, чего мы хотим. `ThreadSwitch` вызывается по прерыванию, сохраняет регистры текущей нити в текущем стеке, переключается на стек новой нити, восстанавливает из ее стека ее регистры, и новая нить получает управление так, будто и не теряла его.

Полный набор регистров, которые нужно сохранить, чтобы нить не заметила переключения, называется *контекстом нити* или, в зависимости от принятой в конкретной ОС терминологии, *контекстом процесса*. К таким регистрам,

как минимум, относятся все регистры общего назначения, указатель стека, счетчик команд и слово состояния процессора. Если система использует виртуальную память, то в контекст входят также регистры диспетчера памяти, управляющие трансляцией виртуального адреса (пример 8.3).

Пример 8.3. Функция переключения контекста в ядре Linux/x86

```
/* Фрагмент файла \arch\i386\kernel\process.c.
 * Сохранение и восстановление регистров общего назначения
 * и сегментных регистров CS, DS и SS осуществляется при входе в ядро
 * и при выходе из него соответственно. */
/*
 *      switch_to(x,y) должна переключать задачи с x на y
 *
 * Мы используем fsave/fwait, поэтому исключения [сопроцессора]
 * сбрасываются в нужный момент времени (пока вызов со стороны
 * fsave или fwait обрабатывается) и не могут быть посланы
 * другому процессу. Отложенное сохранение FP более не имеет
 * смысла на современных ЦПУ и это многое упрощает (SMP и UP
 * [uniprocessor, однопроцессорная конфигурация] теперь
 * обрабатываются одинаково).
 *
 * ЗАМЕЧАНИЕ! Раньше мы использовали аппаратное переключение
 * контекста. Причина, по которой мы больше так не делаем
 * становится очевидна, когда мы пытаемся аккуратно восстановиться
 * из сохраненного состояния, которое стало недопустимым
 * (в частности, висящие ссылки в сегментных регистрах).
 * При использовании аппаратного переключения контекста нет способа
 * разумным образом выйти из плохого состояния [контекста].
 *
 * То, что Intel документирует аппаратное переключение контекста как
 * медленное – откровенная ерунда, этот код не дает заметного ускорения.
 * Однако здесь есть некоторое пространство для улучшения, поэтому
 * вопросы производительности могут рано или поздно оказаться актуальны.
 * [В данном случае], однако, нам важнее, что наша реализация
 * обеспечивает большую гибкость.
 */
void __switch_to(struct task_struct *prev_p, struct task_struct *next_p)
{
    struct thread_struct *prev = &prev_p->thread,
```

```
*next = &next_p->thread;
struct tss_struct *tss = init_tss + smp_processor_id();

unlazy_fpu(prev_p);

/*
 * Перезагрузить esp0, LDT и указатель на таблицу страниц:
 */
tss->esp0 = next->esp0;

/*
 * Сохранить %fs и %gs. Не нужно сохранять %es и %ds,
 * потому что при исполнении в контексте ядра это
 * всегда сегменты ядра.
 */
asm volatile("movl %%fs,%0":"=m" (*(int *)&prev->fs));
asm volatile("movl %%gs,%0":"=m" (*(int *)&prev->gs));

/*
 * Восстановить %fs и %gs.
 */
loadsegment(fs, next->fs);
loadsegment(gs, next->gs);

/*
 * Если это необходимо, перезагрузить отладочные регистры
 */
if (next->debugreg[7]){
    loaddebug(next, 0);
    loaddebug(next, 1);
    loaddebug(next, 2);
    loaddebug(next, 3);
    /* не 4 и 5 */
    loaddebug(next, 6);
    loaddebug(next, 7);
}

if (prev->ioperm || next->ioperm) {
    if (next->ioperm) {
        /*

```

```
* Копирование четырех линий кэша .... не хорошо, но
* и не так уж плохо. У кого-нибудь есть идея лучше?
* Оно воздействует только на процессы, использующие ioperm().
* [Размещение этих TSS в области 4к-tlb и игры с виртуальной
* памятью для переключения битовой маски ввода/вывода на
* самом деле неприемлемы.]
*/
memcpy(tss->io_bitmap, next->io_bitmap,
       IO_BITMAP_SIZE*sizeof(unsigned long));
tss->bitmap = IO_BITMAP_OFFSET;
} else
/*
* Смещение битовой маски, указывающее за пределы ограничителя TSS,
* порождает контролируемое SIGSEGV, если процесс пытается
* использовать команды обращения к портам. Первый вызов
* sys_ioperm() устанавливает битовую маску корректно.
*/
tss->bitmap = INVALID_IO_BITMAP_OFFSET;
}
}
```

Теоретически, довольно легко представить себе процессор, на котором сохранение контекста невозможно. Для этого достаточно, чтобы какой-то из регистров, влияющих на выполнение программы (возможно, но не обязательно, счетчик команд или слово состояния процессора), нельзя было явным образом сохранить или загрузить. У суперскалярных процессоров может возникать более сложная проблема, связанная с динамическим отображением физических регистров на логические регистры. При этом часть логических регистров имеет одновременно несколько разных значений. С практической точки зрения это означает, что значение такого логического регистра не определено.

Сам по себе процесс спекулятивного опережающего исполнения означает, что команды исполняются не в том порядке, в котором закодированы, — т. е. при этом не определено значение счетчика команд.

Такой процессор мог бы использоваться под управлением однопоточной или кооперативно многозадачной ОС, но вытесняющую многозадачную систему на нем реализовать нельзя. Ни один такой процессор не имел коммерческого успеха, во всяком случае в качестве процессора общего назначения.

Как мы видели в главе 6, большинство реальных суперскалярных процессоров при прерываниях выполняют сериализацию, т. е. приводят все свои логи-

ческие регистры (в том числе и логический счетчик команд) к определенным значениям. Поэтому планировщик, вызываемый по прерываниям, вполне может сохранять контекст процессора в том состоянии, в котором он был на момент прерывания.

Однако в вытесняющих ОС вызовы планировщика могут также происходить по инициативе текущей нити. При некоторых типах суперскалярного исполнения это также может приводить к неопределенным значениям логических регистров, так что перед сохранением контекста планировщику необходимо специально попросить процессор о сериализации.

У современных процессоров x86 сериализация происходит при исполнении всех команд, имеющих префикс `lock` (префикс монопольного захвата шины), а также при исполнении команды `xschg` и некоторых других команд, которые работают в режиме захвата шины даже без префикса `lock`. Поскольку спинлоки в ядре реализуются с помощью именно таких команд, разработчик ОС может убить одним ударом двух зайцев: проверяя спинлок, защищающий семафор и/или очередь активных процессов, планировщик одновременно просит процессор сериализоваться.

Впрочем, сериализация, как мы видели в главе 6, — это дорогая операция, которая может потребовать десятки и даже сотни тактов. С этой точки зрения, привлекательной альтернативой сериализации могло бы быть предоставление планировщику доступа к физическим регистрам вместо логических. Таким образом, вместо логических регистров общего назначения, контекст процесса мог бы включать в себя физические регистры, в том числе служебные регистры, описывающие состояние конвейера. Вместо счетчика команд необходимо было бы сохранять список адресов уже исполненных команд (возможно, в виде указателя на текущую линию кэша команд в сочетании с битовой маской, в которой единицы соответствуют исполнившимся командам, а нули — еще не исполнившимся). И так далее.

Попытки выпустить на рынок такие устройства несколько раз предпринимались, но ни разу не имели полного успеха. Возможно, одной из причин является сложность обеспечения бинарной совместимости: большинство изменений микроархитектуры процессора требовало бы изменения планировщика. При этом увеличение количества физических регистров требовало бы простого увеличения объема контекста, но изменение структуры конвейеров и усложнение алгоритмов спекулятивного исполнения могло бы потребовать значительной переработки логики работы планировщика.

Более того, поскольку физических регистров гораздо больше, чем логических, планировщик должен был бы быть реализован с использованием совершенно иной системы команд, нежели пользовательская задача, возможно, непосредственно на микрокоде процессора. На первый взгляд, можно было бы

придумать какие-то альтернативные решения, например, отобразить физические регистры на адреса оперативной памяти или на специализированное адресное пространство. Впрочем, все эти отображения, в конечном итоге, должны делаться микрокодом процессора — т. е. мы должны будем иметь один микрокод для обслуживания пользовательских программ, а другой — для обслуживания нужд планировщика. Из общих соображений совершенно неочевидно, что этот специализированный микрокод окажется проще, меньше по объему или быстрее, чем микрокод, осуществляющий сериализацию.

Процессор Intel 860

Показательна в этом смысле история процессора Intel 80860. Этот процессор, разрабатывавшийся в середине 80-х годов XX века, имел радикальную для своего времени конвейерную архитектуру с тремя независимыми исполнительными устройствами (целочисленной арифметики, арифметики с плавающей точкой и доступа к памяти). Программа имела довольно широкие возможности для доступа к состоянию конвейеров; предполагалось, что это позволит компилятору тонко оптимизировать конвейеризацию. На практике, однако, это привело к тому, что при прерывании требовался сложный код, сохраняющий состояние конвейеров. Задержка прерывания (с учетом работы кода сохранения контекста) в лучшем случае составляла 64 такта, а в худшем — 2000 тактов и более; средняя задержка оказывалась более близка к худшему случаю. Важно отметить, что, вопреки расхожей легенде, i860 все-таки мог сохранять и перезагружать контекст процесса, но делал это очень медленно и с большими сложностями. Первоначально процессор планировалось вывести на рынок рабочих станций и серверов, но проблемы при реализации сохранения контекста привели к отказу от этих планов.

Процессор пытались позиционировать как предназначенный для вычислительных задач; так, в одной из версий компьютера NeXT он использовался в качестве графического сопроцессора. В начале 90-х годов я видел рекламу плат расширения для IBM PC, в которых был установлен i860; предполагалось использовать эту плату в качестве "ускорителя" при научных вычислениях. Широкую известность получили массивно параллельные суперкомпьютеры Intel iPSC/860, Intel Paragon и их модификации. Однако на рынке процессор успеха не имел, в большую серию не пошел, и к середине 90-х годов его производство было свернуто.

Как правило, оказывается неудобным сохранять контекст именно в стеке. Тогда его сохраняют в какой-то другой области памяти, чаще всего в дескрипторе процесса. Многие процессоры имеют специальные команды сохранения и загрузки контекста. Для реализации вытеснения достаточно сохранить контекст текущей нити и загрузить контекст следующей активной нити из очереди. Необходимо предоставить также и функцию переключения нитей по их собственной инициативе, аналогичную ThreadSwitch в кооперативно многозадачных системах. Впрочем, эту функцию обычно совмещают с блокировкой нити на каком-либо примитиве синхронизации или средстве гармонического взаимодействия.

Вытесняющий планировщик с разделением времени ненамного сложнее кооперативного планировщика — и тот, и другой в простейшем случае реализуются несколькими десятками строк на ассемблере. В работе [Прохоров 1990] приводится полный ассемблерный текст приоритетного планировщика системы VAX/VMS, занимающий одну страницу (мне неизвестно, не нарушает ли авторские права фирмы DEC публикация этого текста). Впрочем, планировщики, рассчитанные на многопроцессорные машины, часто бывают несколько сложнее (пример 8.4), т. к. они, во-первых, должны защищать очередь процессов от копий планировщика, работающих на других процессорах (обычно для этого используются спинлоки), и, во-вторых, применяют различные эвристические приемы, направленные на то, чтобы по возможности планировать процесс на одном и том же процессоре. Это позволяет оптимизировать работу процессора с кэш-памятью, а на процессорах с регистровыми окнами, таких как SPARC, может заметно сократить количество перезагрузок окон.

Пример 8.4. Планировщик Linux 2.5

```
/*
 * 'schedule()' — функция планировщика. Это очень простой и
 * приятный планировщик: он не совершенен, но несомненно работает
 * в большинстве случаев.
 *
 * The goto is "interesting".
 *
 * ЗАМЕЧАНИЕ!! Задача 0 является 'пустой' задачей, которая вызывается,
 * когда ни одна другая задача не может выполняться. Она не может быть
 * "убита" и не может "спать". Информация о состоянии в task[0] никогда
 * не используется.
 */
asmlinkage void schedule(void)
{
    struct schedule_data * sched_data;
    struct task_struct *prev, *next, *p;
    struct list_head *tmp;
    int this_cpu, c;

    if (!current->active_mm) BUG();
need_resched_back:
    prev = current;
    this_cpu = prev->processor;
```

```
if (in_interrupt ())
    goto scheduling_in_interrupt;

release_kernel_lock(prev, this_cpu);

/* Выполнить административную работу здесь, пока мы не держим
 * ни одной блокировки.
 */
if (softirq_active(this_cpu) & softirq_mask(this_cpu))
    goto handle_softirq;

handle_softirq_back:

/*
 * 'shed_data' защищена неявно, тем фактом, что мы можем исполнять
 * только один процесс на одном ЦПУ.
 */
sched_data = & aligned_data[this_cpu].schedule_data;

spin_lock_irq(&runqueue_lock);

/*Переместить исчерпанный процесс RR в конец [очереди] */
if (prev->policy == SCHED_RR)
    goto move_rr_last;

move_rr_back:

switch (prev->state) {
    case TASK_INTERRUPTIBLE:
        if (signal_pending(prev)) {
            prev->state = TASK_RUNNING;
            break;
        }
    default:
        del_from_runqueue(prev);
    case TASK_RUNNING:;
}

prev->need_resched = 0;

/*
 * это собственно планировщик:
 */
repeat_schedule:
```

```
/*
 * Выбрать процесс по умолчанию...
 */
next = idle_task (this_cpu);
c = -1000;
if (prev->state == TASK_RUNNING)
    goto still_running;

still_running_back:
list_for_each(tmp, &runqueue_head) {
    p = list_entry(tmp, struct task_struct, run_list);
    if (can_schedule(p, this_cpu)) {
        int weight = goodness(p, this_cpu, prev->active_mm);
        if (weight > c)
            c = weight, next = p;
    }
}

/* Следует ли пересчитывать счетчики? */
if (!c)
    goto recalculate;
/*
 * с этого момента ничто не может помешать нам
 * переключиться на следующую задачу, отметить этот
 * факт в sched_data.
 */
sched_data->curr = next;
#endif CONFIG_SMP
    next->has_cpu = 1;
    next->processor = this_cpu;
#endif
    spin_unlock_irq(&runqueue_lock);

    if (prev == next)
        goto same_process;

#endif CONFIG_SMP
/*
 * Поддерживать значение 'last schedule' для каждого процесса
 * (его необходимо пересчитать, даже если мы планируем тот же
```

```
* процесс). Сейчас это значение используется только в SMP, и оно
* приблизительно, поэтому мы не обязаны поддерживать его,
* пока захвачена блокировка runqueue.
*/
sched_data->last_schedule = get_cycles();

/*
* Мы снимаем блокировку планировщика рано (это глобальная
* блокировка), поэтому мы должны защитить предыдущий процесс
* от повторного планирования во время switch_to().
*/
#endif /* CONFIG_SMP */

kstat.context_swtch++;
/*
* Переключение контекста воздействует на три процесса:
*
* prev == .... ==> (last => next)
*
* Это 'prev', 'далеко предшествующий' размещенному в стеке 'next',
* но функция switch_to() устанавливает prev на (только что
* работавший) процесс 'last'.
* Описание несколько запутанно, но не лишено глубокого смысла.
*/
prepare_to_switch ();
{
    struct mm_struct *mm = next->mm;
    struct mm_struct *oldmm = prev->active_mm;
    if (!mm) {
        if (next->active_mm) BUG();
        next->active_mm = oldmm;
        atomic_inc(&oldmm->mm_count);
        enter_lazy_tlb(oldmm, next, this_cpu);
    } else {
        if (next->active_mm != mm) BUG();
        switch_mm(oldmm, mm, next, this_cpu);
    }
}

if (!prev->mm) {
```

```
    prev->active_mm = NULL;
    mmdrop(oldmm);
}
}

/*
 * Этот оператор только переключает состояние регистров
 * и стека.
 */
switch_to(prev, next, prev);
__schedule_tail(prev);

same_process:
reacquire_kernel_lock(current);
if (current->need_resched)
    goto need_resched_back;

return;

recalculate:
{
    struct task_struct *p;
    spin_unlock_irq(&runqueue_lock);
    read_lock(&tasklist_lock);
    for_each_task (p)
        p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
    read_unlock(&tasklist_lock);
    spin_lock_irq(&runqueue_lock);
}
goto repeat_schedule;

still_running:
c = goodness(prev, this_cpu, prev->active_mm);
next = prev;
goto still_running_back;

handle_softirq:
do_softirq();
goto handle_softirq_back;

move_rr_last:
```

```
if (!prev->counter) {  
    prev->counter = NICE_TO_TICKS (prev->nice);  
    move_last_runqueue (prev);  
}  
goto move_rr_back;  
  
scheduling_in_interrupt:  
    printk("Scheduling in interrupt\n");  
    BUG();  
    return;  
}
```

Контексты современных процессоров

У современных процессоров, имеющих десятки регистров общего назначения и виртуальную память, размер контекста процесса измеряется сотнями байтов. Например, у процессора VAX контекст процессора состоит из 64 32-разрядных слов, т. е. 256 байт. При этом VAX имеет только 16 регистров общего назначения, а большая часть остальных регистров так или иначе относится к системе управления виртуальной памятью.

У микропроцессоров SPARC, имеющих регистровый файл объемом до нескольких килобайтов, контекст, на первый взгляд, должен быть чудовищного размера. Однако программе одновременно доступны лишь 32 регистра общего назначения, 24 из которых образуют скользящее по регистровому файлу окно. Благодаря этому факту, контекст процессора SPARC состоит только из первых восемью регистров общего назначения и служебных регистров. Регистровое окно новой нити выделяется в свободной области регистрового файла, а его передвижение обрабатывается с помощью исключений заполнения и очистки окна.

Если в системе всего несколько активных процессов, может оказаться так, что их регистровые окна постоянно "живут" в регистровом файле, поэтому объем данных, реально копируемых при переключении нитей, у SPARC не больше, чем у CISC-процессоров с небольшим количеством регистров общего назначения. Впрочем, и у SPARC, и у CISC-процессоров основную по объему часть контекста процесса составляют регистры диспетчера памяти.

На этом основано преимущество транспьютера перед процессорами традиционных и RISC-архитектур. Дело в том, что транспьютер не имеет диспетчера памяти и у него вообще очень мало регистров. В худшем случае при переключении процессов (в транспьютере, как и в старых ОС, нити называются процессами) должно сохраняться 7 32-разрядных регистров. В лучшем случае сохраняются только 2 регистра — счетчик команд и статусный регистр. Кроме того, перенастраивается регистр *wptr*, который выполняет по совместительству функции указателя стека, базового регистра сегмента статических данных процесса и указателя на дескриптор процесса.

Транспьютер имеет три арифметических регистра, образующих регистровый стек. При этом обычное переключение процессов может происходить только, когда этот стек пуст. Такая ситуация возникает довольно часто; например, этот стек обязан быть пустым при вызовах процедур и даже при условных и безусловных переходах, поэтому циклическая программа не может не иметь точек,

в которых она может быть прервана. Упомянутые в предыдущем разделе команды обращения к линкам также исполняются при пустом регистровом стеке. Поэтому оказывается достаточно перезагрузить три управляющих регистра, и мы передадим управление следующему активному процессу.

Операция переключения процессов, а также установка процессов в очередь при их активизации полностью реализованы на микропрограммном уровне.

Деактивизация процесса происходит только по его инициативе, когда он начинает ожидать сигнала от таймера или готовности линка. При этом процесс исполняет специальную команду, которая устанавливает его в очередь ожидающих соответствующего события, и загружает контекст очередного активного процесса. Когда приходит сигнал таймера или данные по линку, то также вызывается микропрограмма, которая устанавливает активизированный процесс в конец очереди активных.

У транспьютера также существует микропрограммно реализованный режим разделения времени, когда по сигналам внутреннего таймера активные процессы циклически переставляются внутри очереди. Такие переключения, как уже говорилось, могут происходить, только когда регистровый стек текущего процесса пуст, но подобные ситуации возникают довольно часто.

Кроме обычных процессов в системе существуют так называемые высокоприоритетные процессы. Если такой процесс получает управление в результате внешнего события, то текущий низкоприоритетный процесс будет прерван независимо от того, пуст его регистровый стек или нет. Для того чтобы при этом не разрушить прерванный процесс, его стек и весь остальной контекст записываются в быструю память, расположенную на кристалле процессора. Это и есть тот самый худший случай, о котором говорилось ранее. Весь цикл переключения занимает 640 нс по сравнению с десятками и порой сотнями микросекунд у традиционных процессоров [INMOS 72 TRN 203 02, Харп 1993].

Благодаря такой организации транспьютер не имеет равных себе по времени реакции на внешнее событие. На первый взгляд, микропрограммная реализация такой довольно сложной конструкции, как планировщик, снижает гибкость системы. В действительности, в современных системах планировщики имеют довольно стандартную структуру, и реализация, выполненная в транспьютере, очень близка к этому стандарту, известному как *микроядро* (*microkernel*) (см. разд. 8.4).

8.2.1. Планировщики с приоритетами

Гермиона: Вы можете умереть... или вас даже могут выгнать из школы.

Рон (в сторону): Ей надо пересмотреть приоритеты.

К. ф. "Гарри Поттер и философский камень"

В многозадачных системах часто возникает вопрос: в каком порядке выполнять готовые процессы? Как правило, бывает очевидно, что одни из процессов важнее других. Например, в системе может существовать три процесса, имеющих готовые к исполнению нити: процесс — сетевой файловый сервер,

интерактивный процесс — текстовый редактор и процесс, занимающийся плановым резервным копированием с диска на ленту. Очевидно, что хотелось бы в первую очередь разобраться с сетевым запросом, затем — отреагировать на нажатие клавиши в текстовом редакторе, а резервное копирование может подождать сотню-другую миллисекунд. С другой стороны, мы должны защитить пользователя от ситуаций, в которых какой-то процесс вообще не получает управления, потому что система постоянно занята более приоритетными заданиями.

Действительно, вы запустили то же самое резервное копирование и сели играть в тетрис или писать письмо любимой женщине, а после получаса игры обнаружили, что ни одного байта не скопировано — процессор все время был занят.

Самым простым и наиболее распространенным способом распределения процессов по *приоритетам* является организация нескольких очередей в соответствии с приоритетами. При этом процесс из низкоприоритетной очереди получает управление тогда и только тогда, когда все очереди с более высоким приоритетом пусты.

Приоритеты процессов в транспьютере

Простейшим случаем такой организации является транспьютер, имеющий две очереди. В транспьютере при этом планировщик не может отобрать управление у высокоприоритетного процесса. В этом смысле низкоприоритетные задачи вынуждены полагаться на "порядочность" высокоприоритетных, т. е. на то, что те освобождают процессор в разумное время.

Отчасти похожим образом организован планировщик системы VAX/VMS. Он имеет 32 приоритетных очереди, из которых старшие 16 называются процессами реального времени, а младшие — разделенного.

При этом процесс реального времени исполняется всегда, когда готов к исполнению, и в системе нет более приоритетных процессов. ОС и процессы разделенного времени также вынуждены полагаться на его порядочность. Поэтому привилегия запускать такие процессы контролируется администратором системы.

Легко понять, что разделение времени обеспечивает более или менее справедливый доступ к процессору для задач с одинаковым приоритетом. В случае транспьютера, который имеет только один приоритет и, соответственно, одну очередь для задач разделенного времени, этого оказывается достаточно. Однако современные ОС как общего назначения, так и реального времени, имеют много уровней приоритета.

В системах с пакетной обработкой, когда для задачи указывают верхнюю границу времени процессора, которое она может использовать, часто более короткие задания идут с более высоким приоритетом. Кроме того, более высокий приоритет дают задачам, которые требуют меньше памяти.

В системах разделенного времени велика доля интерактивных задач, для которых сложно заранее указать потребности в ресурсах.

Так, если программа начала вычисления, не прерываемые никакими обращениями к внешней памяти или терминалу, мы можем предположить, что она будет заниматься такими вычислениями и дальше. Напротив, если программа сделала несколько запросов на ввод/вывод, следует ожидать, что она и дальше будет активно выдавать такие запросы. Предпочтительными для системы будут те программы, которые захватывают процессор на короткое время и быстро отдают его, переходя в состояние ожидания внешнего или внутреннего события. Таким процессам система стремится присвоить более высокий приоритет. Если программа ожидает завершения запроса на обращение к диску, то это также выгодно для системы — ведь на большинстве машин чтение с диска и запись на него происходят параллельно с работой центрального процессора.

Таким образом, система динамически повышает приоритет тем заданиям, которые освободили процессор в результате запроса на ввод/вывод или ожидание события и, наоборот, снижает тем заданиям, которые были сняты по истечении кванта времени.

При этом наиболее высокий приоритет автоматически получают интерактивные задачи и программы, занятые интенсивным вводом/выводом.

На первый взгляд, не очень понятно, зачем это нужно. Впрочем, вспомним определение системы разделенного времени, которое мы приводили во введении, — это система, оптимизирующая среднее время реакции на запрос пользователя. В современных многопоточных системах реакция на пользовательские запросы осуществляется за счет пробуждения нитей, ждущих соответствующих событий. Таким образом, чтобы улучшить время реакции, системе выгодно повышать приоритет для тех нитей, которые чего-то ожидают. Напротив, нити, которые снимаются по истечению кванта времени, по определению ничего не ожидают и, как следствие, не вносят вклада в наблюдаемое пользователем время реакции.

При таком описании динамическая приоритизация может выглядеть как обман клиента: клиент заказывал среднее время реакции — ему улучшили среднее время реакции за счет отъема ресурсов у процессов, которые не вносят вклада в оптимизируемый показатель.

Как ни странно, пользователи почему-то очень рады обманываться именно таким образом. Я имею опыт работы на ОС с разными стратегиями планирования, в том числе и с системами вовсе без динамической приоритизации, например Диспак для БЭСМ-6, и должен отметить, что интерактивная работа в системе с динамической приоритизацией гораздо приятнее.

Частично это объясняется тем, что при работе с хорошо спроектированными приложениями пользователь может предсказывать, какие операции приводят к быстрой реакции приложения, а какие — вычислительно дороги и, соответственно, могут привести к задержке реакции системы.

Так, при работе с базой данных пользователь ожидает, что операции по обновлению нескольких полей в нескольких записях будут происходить быстро, а построение отчета по всей базе может работать долго. При работе с приложением CAD пользователь ожидает быстрой реакции при редактировании "проводочной" модели, но готов некоторое время подождать при расчете конструкции на прочность или при построении фотoreалистичного изображения. При подготовке текста в LaTeX пользователь ожидает быстрой редакции от текстового редактора, готов немного подождать при сохранении файла или, тем более, при сохранении файла в репозиторий системы контроля версий и готов подождать несколько большее время при компиляции текста, его разбиении на страницы, построении индексов и других операций.

Интерактивная работа и фоновые нити в MS Word/Win32

В этом смысле текстовые процессоры WYSIWYG сложно признать хорошо спроектированными приложениями. Приемлемая скорость реакции на действия пользователя в них обеспечивается не всегда и не всегда предсказуемым образом. Так, в старых версиях Word переразбиение на страницы осуществлялось по мере набора текста, так что вставка текста в начало документа из нескольких страниц превращалась в чистое издевательство над пользователем.

Незнакомые с техникой подготовки печатных документов читатели, возможно, нуждаются в пояснении, что такого сложного в задаче разбиения текста на страницы. Дело в том, что в типографской практике хорошим тоном считается минимизировать количество разбиений параграфов при переносе со страницы на страницу. Также ряд явлений, возникающих при наивном разбиении текста на страницы, например "висячие строки", когда на страницу попадает только одна строка из параграфа, или "висячие заголовки", когда заголовок раздела попадает на конец страницы, а тело раздела размещается на следующей, считаются совершенно нетерпимыми.

При этом также считается хорошим тоном делать текст на страницах по возможности одинаковой высоты. В условиях, когда текст содержит крупные неразбиваемые объекты, такие как рисунки и таблицы, это требование может оказаться трудновыполнимым. Во всяком случае, минимизация "дефекта" заполнения страниц сводится к так называемой задаче о рюкзаке, которая относится к классу NP-полных задач (т. е. задач, для которых неизвестно решения, работающего быстрее, чем полный перебор вариантов). На практике, системы подготовки печатных документов — как TeX и troff, так и WYSIWYG-системы — ищут лишь относительно приемлемое решение, а для подготовки документа типографского качества, т. е. такого, который не стыдно издать в виде книги, нуждаются в помощи верстальщика. Тем не менее даже поиск относительно приемлемого решения в сложных случаях может продолжаться десятки секунд.

Ряд других функций — например, построение индексов и перекрестных ссылок — в Word реализованы с учетом ограничений, налагаемых подходом WYSIWYG, т. е., попросту говоря, совершенно непотребным образом.

Например, Word не способен решить задачу, которую мог решать любой ассемблер в 50-е годы прошлого века, — он не может поддерживать в актуальном состоянии ссылки на номера глав и разделов в тексте документа, состоящего из нескольких файлов. Причем неспособность эта обеспечена на архитектурном уровне — формат файлов сохранения не допускает возможности использовать ссылки на другой файл в вычисляемых полях. Номера разделов и глав в ссылках в тексте этой книги расставлялись сотрудниками издательства вручную. Более сложные функции, например работа с библиографическими базами данных, в Word просто отсутствуют; так, слово "библиография" вообще не упоминается в тексте on-line help.

В Microsoft Word для Win32 задача переразбиения текста на страницы решается фоновой нитью, которая работает параллельно с пользовательским интерфейсом. При нормальной работе пользователь этого вообще не замечает (впрочем, внимательный пользователь может увидеть, что при редактировании в нижней панели возникает анимационная иконка в виде перелистываемой книги, на которой что-то пишет карандаш). Фактически, именно введение фонового разбиения на страницы в версии Office 95 сделало Word более или менее пригодным для редактирования больших документов.

Однако при сохранении файла Word вынужден завершать разбиение на страницы. Поэтому при сохранении — в том числе при автосохранении — пользователь может оказаться вынужден ждать фоновой нити. При автосохранении это выглядит как внезапная блокировка интерфейса текстового редактора (этую ситуацию можно считать примером инверсии приоритета, которая рассматривается далее в этой главе).

При редактировании небольших файлов паузы при автосохранении почти не заметны, но при редактировании большого файла с таблицами и рисунками (например, главы книги) они превращаются в серьезную проблему. Так, при редактировании главы 9 второго издания этой книги на Р IV/2.8 ГГц с 512 Мбайт памяти паузы достигали нескольких секунд. При редактировании этой же главы на ноутбуке Р II/600 с 196 Мбайт памяти Word проводил в этих паузах десятки секунд и даже минуты, порой большую часть интервала времени между автосохранениями, так что работать было практически невозможно. Выключать же автосохранение, особенно при работе с большим документом, также нежелательно — Word, даже самых последних версий, известен своей "падучестью".

В Word 2000 переразбиение на страницы можно выключить, выбрав режим просмотра документа Web. К сожалению — как я смог убедиться при работе над вторым изданием данной книги — это создает гораздо более серьезную проблему. Автосохранение в этом режиме происходит не просто без разбиения на страницы, а с некорректным разбиением (во всяком случае в Word 2000 SP 1), так что после нескольких сеансов редактирования файл оказывается невозможен переключить в нормальный режим просмотра. Точнее, переключить его можно, но текст главы на экране при этом исчезает, остаются только первые две-три страницы. В файле, к счастью, текст остается — в этом можно убедиться, вернувшись к режиму просмотра Web. Но объяснить текстовому процессору, что ему нужно просто переразбить текст на страницы "с нуля", невозможно — даже специалисты издательства БХВ, посмотрев на мой файл, не смогли

предложить ничего лучше экспорта файла в RTF и обратно (это привело к потере всей истории редактирования).

Возможно, в более современных версиях Word, например в Word 2003, этот файл можно было бы починить средствами восстановления поврежденных документов.

Таким образом, чтобы удовлетворить пользователя, разработчик интерактивного приложения должен обеспечить быструю — меньше кванта времени планировщика — обработку тех действий, на которые пользователь ожидает быстрой реакции, и обеспечить адекватную обратную связь для действий, которые физически невозможно реализовать за такое время. В многопоточных приложениях такие действия можно вынести в фоновый поток (впрочем, как мы видели ранее, при этом можно столкнуться со своеобразными проблемами).

Легко понять, что работа динамического приоритизатора систем разделенного времени состоит именно в том, чтобы повысить приоритет задачам и нитям, непосредственно обслуживающим интерактивные функции за счет тех задач и нитей, завершения которых пользователь готов подождать.

Большинство старых ОС, такие, как OS/390, VAX/VMS, Unix, не делают различия между интерактивными и просто ориентированными на ввод/вывод задачами и повышают приоритет всем задачам, которые блокируются на операциях ввода/вывода и межпроцессного взаимодействия.

В старых версиях VAX/VMS это приводит к своеобразной проблеме, которую можно даже использовать для атаки отказа сервиса. Система повышает приоритет задачам, которые обмениваются сообщениями через mailbox. Запустив пару задач, которые обмениваются через mailbox коротким сообщением и больше ничего не делают, пользователь может парализовать всю интерактивную работу, потому что эти задачи быстро получат приоритет выше, чем у интерактивных командных процессоров. Во всяком случае, в VAX/VMS 4.x это действительно приводило к блокировке интерактивных задач, так что администратору системы стоило больших усилий "убить" эту пару задач.

В Unix-системах данная проблема решается очень просто — система никогда не повышает приоритет задачи выше некоторого базового уровня. На практике, в Unix-системах даже не говорят о динамическом повышении приоритета. Обычно работу динамической приоритизации в Unix описывают следующим образом. Приоритет процесса складывается из статического приоритета (nice level) и штрафа за использование процессора (CPU penalty). Оба эти значения представляют собой положительные целые числа; чем больше их сумма, тем ниже приоритет задачи. Nice level наследуется от родительского процесса или, если это необходимо, повышается запуском задачи с использованием команды shell, которая называется nice. Если это совсем уж необходимо, задача

может повысить свой nice level с помощью системного вызова, который также называется nice. Простой пользователь может только повышать nice level своих задач, а суперпользователь (администратор системы) может также и понижать.

Штраф начисляется, когда система отнимает у задачи управление по исчерпанию кванта времени, и сбрасывается, когда задача сама уступает процессор.

VMS повышает приоритет также и тем задачам, которые остановились в ожидании подкачки страницы. Это сделано потому, что если программа несколько раз выскочила за пределы своего рабочего набора (т. е. потребовала еще страницу, когда ее рабочий набор весь занят), то она, скорее всего, будет делать это и далее, а процессор она освобождает на время подкачки.

Отдельные современные ОС, среди которых нельзя не упомянуть Windows NT/2000/XP и OS/2 (начиная с версии 4.0), дополнительно повышают приоритет интерактивным задачам или, точнее, задачам, которые связаны с тем окном графического пользовательского интерфейса, которое сейчас имеет фокус ввода. В Windows это поведение по умолчанию включено у клиентских версий системы и выключено у серверных; оно может контролироваться флагом **Application Response** в свойствах объекта **System** и соответствующим ключом реестра. При работе простых однопоточных приложений это действительно приводит к повышению наблюдаемой скорости реакции, но при работе многослойных приложений, часть функциональности которых реализуется фоновыми процессами, последствия не столь однозначны.

Нужно отметить, что процесс разделенного времени может повысить свой приоритет до максимального в классе разделения времени, но никогда не сможет стать процессом реального времени. А для процессов реального времени динамическое изменение приоритетов обычно не применяется.

В Unix System V Release 4 процессы могут принадлежать к разным классам планирования. Класс планирования определяет диапазон приоритетов, квант времени и политику динамической приоритизации. В старых версиях системы поддерживалось три класса планирования: реального времени (RT), системный и разделенного времени (TS). При этом приоритет процессов RT был больше системных, а системных — больше TS. Для процессов реального времени используется фиксированная приоритизация, для TS — динамическая приоритизация с приоритетом, складывающимся из nice level и CPU penalty.

Пользовательские процессы могли исполняться в классах приоритетов RT и TS. Для запуска процессов или, точнее, LWP с классом планирования RT необходимы административные привилегии.

В современных версиях Solaris были добавлены интерактивный (IA), честный (FSS — Fair Share Scheduling) и фиксированный (FX) классы планирования. TS и новые классы планирования разделяют одни и те же приоритеты, но используют разные кванты времени и разную политику приоритизации. Не рекомендуется совмещать на одном процессоре динамические (TS/IA), честные (FSS) и статические (FX) классы планирования, однако на многопроцессорной машине можно настроить исполнение процессов со статическими приоритетами на одних процессорах, а с динамическими — на других.

Управление приоритетами в OS-9

Любопытно реализовано динамическое изменение приоритета в OS-9. В этой ОС каждый процесс имеет статически определенный приоритет и *возраст* (age) — количество просмотров очереди с того момента, когда этот процесс в последний раз получал управление. Обе эти характеристики представлены 16-разрядными беззнаковыми числами. Процесс может повысить или понизить свой приоритет, исполнив соответствующий системный вызов, но система по собственной инициативе никогда не меняет его. При этом управление каждый раз получает процесс с наибольшей суммой статического приоритета и динамически изменяющегося возраста (рис. 8.1 — в изображенной на рисунке ситуации будет выбран процесс 12). Если у двух процессов такие суммы равны, то берется процесс с большим приоритетом. Если у них равны и приоритеты, то берется тот, который оказался ближе к началу очереди.

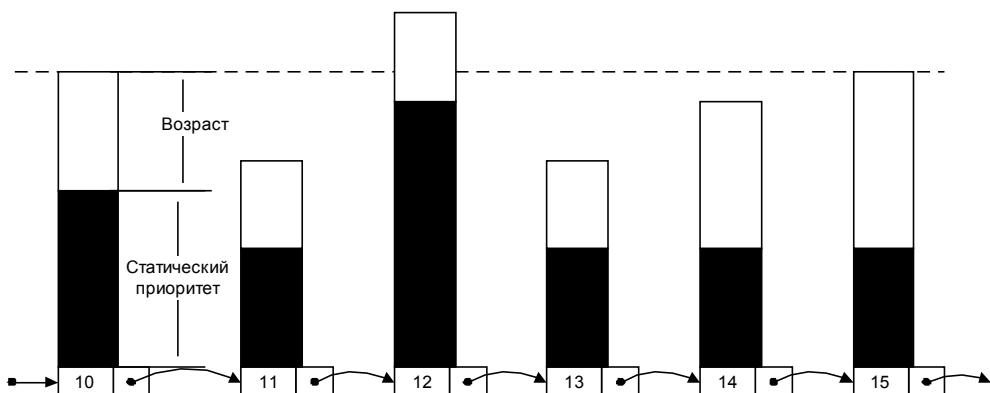


Рис. 8.1. Приоритеты и возраст в OS/9

Этот алгоритм гарантирует, что любой низкоприоритетный процесс рано или поздно получит управление. Если же нам нужно, чтобы он получал управление раньше, то мы должны просто повысить его приоритет.

Кроме того, можно запретить исполнение процессов со статическим приоритетом ниже заданного. Это может уменьшить загрузку процессора и, например, позволит высокоприоритетным процессам обработать увеличившийся поток внешних событий. Понятно, что такой запрет можно вводить только на небольшое время, чтобы не нарушить справедливое распределение процессора.

Возможна и более тонкая регулировка — системный вызов, который запрещает увеличивать возраст процесса больше заданного значения. То есть процесс, стоя в очереди, может достичь этого максимального возраста, после чего он по-прежнему остается в очереди, но его возраст уже не увеличивается.

Получающаяся в результате схема распределения времени процессора отчасти похожа на двухслойную организацию VAX/VMS, когда исполнение процессов со статическим приоритетом, превышающим границу, не может быть прервано низкоприоритетным процессом.

8.2.2. Честное планирование

Зачем меня ты надинамил;
Неужли ты забыл о том,
Как мы с тобой в помойной яме
Одним делились косяком?
Зачем меня ты кинул круто,
Зачем порушил мой ништяк?

А. Гуницкий

Честное планирование (Fair Share Scheduling, FSS) — относительно новая концепция, интерес к которой появился из-за развития вычислительных систем публичного доступа: сервисов приложений (Application Service Provider, ASP) и, в первую очередь, веб-хостинга (web hosting). Приобретая долю ресурсов вычислительной системы, пользователь, разумеется, имеет право требовать гарантий, что эти ресурсы ему будут предоставлены.

В 60-е и 70-е годы XX века пользователи обычно приобретали машинное время для запуска вычислительных задач. Например, пользователь хочет запустить задачу численного моделирования, время работы которой измеряется двадцатью часами. Он приобретает двадцать часов процессорного времени и запускает свою задачу. Разумеется, поскольку система многозадачная, задача может исполняться существенно больше двадцати астрономических часов, но пользователя это, в общем, устраивает. Задача гарантии требуемой доли ресурсов при этом может решаться простыми средствами, не имеющими прямого отношения к системе приоритизации процессов, такими, как квоты процессорного времени.

Но если пользователь покупает ресурсы системы для запуска интерактивных приложений (веб-сервисы и серверы баз данных в этом смысле являются интерактивными приложениями), ему недостаточно гарантии, что ему выделят двадцать часов процессорного времени в течение неопределенного интервала астрономического времени. Как правило, такому пользователю требуется гарантия, что его задачи будут получать определенную (часто не очень большую, но это, разумеется, в конечном итоге определяется задачей) долю процессорного времени в течение каждой секунды или даже в течение каж-

дой десятой доли секунды. Только при этих условиях можно обеспечить приемлемое время реакции системы для конечного пользователя.

Легко убедиться, что классические планировщики разделенного времени, описанные в предыдущем разделе, не способны выполнить это требование. Они обеспечивают определенную справедливость планирования для каждого отдельного процесса, но, просто запустив много процессов, пользователь может употребить непропорциональную долю системных ресурсов. В определенных пределах это может контролироваться за счет квот количества процессов на пользователя, но даже при самых небольших значениях такой квоты, пользователи с многопоточными приложениями будут получать преимущество перед пользователями, чьи задачи не могут быть реализованы в качестве многопоточных.

Первые попытки решить эту проблему относятся к концу 80-х годов прошлого столетия. В работе [Kay/Lauder 1988] описан Share — экспериментальный планировщик для BSD Unix, который использовал динамическое честное планирование. В Share, как и в простом планировщике Unix, приоритет процесса складывался из базового приоритета (nice level) и штрафа за потребление ресурсов. Однако штраф за потребление ресурсов начислялся всем процессам того же пользователя. Таким образом, у пользователя, задачи которого потребляли непропорционально большую долю ресурсов (не только процессорных), снижался реальный приоритет всех процессов.

В реальном Share использовалась весьма сложная схема начисления штрафа. Так, штраф за процессорное время, поглощаемое высокоприоритетными и низкоприоритетными задачами, был различен — за задачи, выполняющиеся с высоким nice level, штраф уменьшался. Таким образом, пользователь был заинтересован в использовании nice для неинтерактивных задач. Штрафы начислялись также за сам факт создания дополнительных процессов (далее в этом разделе мы увидим, почему это важно).

По утверждению разработчиков Share, пользователи положительно реагировали на использование такой системы. Действительно, система воспринималась как "честная", а ее поведение было хотя и не полностью детерминистическим, но интуитивно понятным. Впрочем, эта система обладала важным недостатком, ограничившим ее практическое применение.

Действительно, важным преимуществом классического планировщика разделенного времени является его локальность. Изменения приоритета задач состоят в перестановке задач из одной приоритетной очереди в другую, а при самом планировании нам достаточно взять первую задачу из самой приоритетной очереди (в многопроцессорном планировщике это не совсем так, но возникающие там трудности реализуются за счет усложнения структуры очередей, а не за счет просмотра этих очередей). Все решения о том, следует ли

нам изменить приоритет процесса, и если следует, то на сколько, принимаются на основе значения системного таймера и информации, размещенной в дескрипторе процесса. Так, для того, чтобы поднять приоритет процессу, который долго спал на семафоре, нам достаточно знать, когда он заснул на семафоре и сколько времени сейчас. Таким образом, все подзадачи динамического планировщика в однопроцессорной системе решаются за константное время.

Напротив, планировщик Share при каждом акте потребления ресурсов, вообще говоря, должен пересчитывать приоритеты всех остальных задач этого пользователя. То есть накладные расходы такого планировщика, вообще говоря, растут вместе с количеством процессов в системе.

Другим недостатком Share была его недетерминированность: система обеспечивала распределение ресурсов, которое в среднем воспринимается как честное, но на коротких интервалах времени неравномерности в распределении процессорного времени могли быть значительными. В условиях, когда пользователи оплачивают определенную долю ресурсов, это неприемлемо. Поэтому интерес к планировщикам такого типа долгое время оставался чисто академическим.

В второй половине 90-х годов были реализованы несколько более совершенные планировщики, обеспечивающие гарантии качества обслуживания. Как и Share, эти планировщики основаны на признании того факта, что структура распределения ресурсов в системе должна быть иерархической. Вместо того чтобы отслеживать потребление ресурсов каждой отдельной задачей, система должна объединять задачи в группы и отслеживать потребление ресурсов группой в целом.

В планировщике FSS (Fair Share Scheduler), который поставлялся в качестве отдельного продукта для Sun Solaris 8 и был включен в основную поставку Solaris 9, для группировки процессов используется понятие проекта (project). Каждому проекту выделяется относительная доля процессорного времени, измеряемая неотрицательным целым числом (share — доля).

Реальная доля процессорного времени, получаемая задачами проекта, определяется по довольно простому принципу. Система суммирует значения share для всех проектов, в рамках которых есть хотя бы одна задача, и делит share каждого из проектов на полученное число. Таким образом, если у нас есть два проекта, у одного из которых share равно 6, а у другого — 2, то первый проект получит $6/8$ процессорного времени, а второй — $2/8$.

В пределах каждого из проектов действует традиционная динамическая приоритизация Unix. Таким образом, планируя задачу, система ищет самую приоритетную из задач среди проектов, которые еще не выбрали свою долю про-

цессорного времени. Если все проекты с ненулевыми значениями share выбрали свою долю, то система исполняет самую приоритетную из всех задач, в том числе принадлежащих к проектам с нулевым значением share.

Видно, что время работы такого планировщика зависит от количества активных проектов в системе, поэтому такие планировщики находят применение только в условиях, когда они действительно необходимы. Однако по мере развития бизнеса поставщиков приложений потребность в таких планировщиках растет.

8.2.3. Инверсия приоритета

Гэндалльф: Это Балрог! Бежим!

Гимли: Но мы все равно не можем бежать быстрее демона!

Леголас: Нам достаточно бежать быстрее тебя!

Анонимная пародия

В системах с приоритетным планированием при взаимодействии процессов с разными приоритетами возникает ряд специфических проблем, объединяемых названием *инверсия приоритета* (*priority inversion*).

Один из наглядных примеров этой проблемы мы видели в предыдущем разделе, когда обсуждали фоновое разбиение текста на страницы в текстовом процессоре Microsoft Word. Действительно, при сохранении документа Word должен завершить разбиение на страницы, а для этого он вынужден остановить редактирование, иначе есть риск, что переразбиение на страницы никогда не завершится. Таким образом, при редактировании большого документа с включенным автосохранением пользователь большую часть времени взаимодействует с высокоприоритетным и действительно быстро реагирующим потоком, отвечающим за редактирование, — но иногда оказывается вынужден ждать завершения работы низкоприоритетного фонового потока. Это выглядит как внезапное "зависание" Word. В добавок ко всему, этот фоновый поток вынужден делиться ресурсами с не менее жадным до процессорного времени потоком, который занимается фоновой проверкой орфографии...

Если в интерактивных приложениях инверсия приоритета только раздражает, в системах реального времени она может приводить к действительно серьезным проблемам.

При расчете времени реакции на событие разработчик системы реального времени должен принимать во внимание не только время исполнения кода, непосредственно обрабатывающего это событие, но и времена работы всех критических секций во всех нитях, которые могут удерживать мутексы, необходимые обработчику, — ведь обработчик не сможет продолжить выполнение, пока не захватит эти мутексы, а произвольно снимать их нельзя, потому

что они сигнализируют, что защищаемый ими разделяемый ресурс находится в несогласованном состоянии.

Если высокоприоритетная нить пытается захватить мутекс, занятый низкоприоритетной нитью, то в определенном смысле получится, что эта нить будет работать со скоростью низкоприоритетного процесса.

В условиях, когда планировщик не обеспечивает справедливого распределения времени центрального процессора, а в системе наравне с высокоприоритетной нитью (работа которой нас интересует) и низкоприоритетной (которая держит мутекс) существуют еще среднеприоритетные нити, может оказаться, что низкоприоритетная нить не будет получать управления в течение значительного времени. На это же время будет заблокирована и высокоприоритетная нить.

Особенно серьезна эта проблема, когда высоко- и низкоприоритетная нити относятся к разным классам планирования — а в системах реального времени так оно и есть.

Аналогичная проблема может возникать также при работе с синхронными примитивами гармонического взаимодействия — линками транспьютера, randеву языка Ада и т. д.

Инверсия приоритета в Mars Pathfinder

При разработке ПО для бортового компьютера космического аппарата Mars Pathfinder была принята архитектура, которую сами разработчики называли "общей шиной" — глобальная разделяемая область памяти, которую все процессы в системе использовали для коммуникации. Доступ к этой области защищался мутексом.

В системе существовала высокоприоритетная нить, занимавшаяся управлением шиной, которая периодически запускалась по сигналам таймера и собирала из "шины" некоторую важную для нее информацию. Эта же нить должна была сбрасывать сторожевой таймер, подтверждая, что система функционирует нормально.

Кроме того, в системе существовало еще несколько нитей, осуществлявших доступ к шине, в частности нить сбора метеорологических данных. Эта нить также запускалась по расписанию и копировала данные в буфер "шины", разумеется, захватывая при этом мутекс.

Таким образом, если нить управления "шиной" запускалась во время работы "метеорологической" нити, то она должна была ждать некоторое дополнительное время. В моменты, когда в системе не было других активных нитей, это не приводило к проблемам. Однако когда накладывалось исполнение трех нитей, интервал ожидания сильно увеличивался, и это приводило к срабатыванию сторожевого таймера и сбросу бортового компьютера — в данном случае, ложному, потому что система все-таки оставалась работоспособной.

При тестировании на земле сбросы такого рода несколько раз происходили, но разработчики не смогли воспроизвести условия их возникновения (срабатывание ошибки требует наложения трех нитей, каждая из которых сама по себе

большую часть времени заблокирована) и запустили аппарат с недиагностированной проблемой. При эксплуатации на Марсе непредсказуемые и происходящие в среднем раз в несколько дней сбросы бортового компьютера доставляли много неудобств центру управления полетом; к тому же, специалисты центра вынуждены были объяснять происходящее журналистам. На копии системы на Земле сбросы долгое время не удавалось воспроизвести; наконец, под утро, когда в лаборатории оставался только один разработчик, сброс все-таки произошел и по анализу отладочного дампа системы проблему удалось диагностировать (по материалам [risks 1997]).

Основным средством борьбы с инверсией приоритета является *наследование приоритета* (*priority inheritance*). Обычно наследование контролируется флагом в параметрах мутекса или в параметрах системного вызова, захватывающего мутекс. Если высокоприоритетная нить пытается захватить мутекс с таким флагом, то приоритет нити, удерживающей этот мутекс, приравнивается приоритету нашей нити. Таким образом, в каждый момент времени реальный приоритет нити, удерживающей мутекс, равен наивысшему из приоритетов нитей, ожидающих этого мутекса.

Более радикальное решение называется *потолком приоритета* (*priority ceiling*). Оно состоит в том, что приоритет нити, удерживающей мутекс, приравнивается наивысшему из приоритетов нитей, которые могут захватить этот мутекс. Разумеется, во время исполнения определить потолок приоритета невозможно, он должен устанавливаться программистом как параметр мутекса.

Легко понять, что во время работы в критических секциях, защищенных такими мутексами, задачи — даже низкоприоритетные — должны подчиняться всем ограничениям, которые может накладывать исполнение в режиме реального времени. Если мы в каком-то смысле не доверяем низкоприоритетной нити (в частности, не можем оценить время, в течение которого она будет удерживать мутекс), нам следует отказаться от использования разделяемой памяти для взаимодействия с этой нитью и перейти к каким-либо буферизованным средствам обмена данными. При этом достаточно обеспечить гарантированное время передачи данных в буфер; для того, чтобы избежать переполнения буфера, достаточно того, чтобы средний темп генерации данных высокоприоритетной нитью был ниже среднего же возможного темпа их обработки низкоприоритетным потребителем. Впрочем, если темп такой обработки подвержен большим флуктуациям (например, из-за наличия в системе других процессов), может оказаться необходимо предусмотреть возможность сброса части данных для борьбы с переполнением буфера — именно так борются с перегрузками сетевые маршрутизаторы и коммутаторы.

Управление наследованием или потолком приоритета для мутексов в обязательном порядке предоставляется системами реального времени и некоторы-

ми многопоточными API общего назначения, например POSIX Thread library (POSIX 1003c).

Необходимо отметить, что и наследование, и потолок приоритета применимы только к мутексам и другим примитивам синхронизации, для которых можно указать, какая именно нить их удерживает. Точнее говоря, для борьбы с инверсией приоритета необходимо, чтобы приоритет повышался для нити, которая будет освобождать семафор. Для мутексов это всегда та же нить, которая его захватывала, но для семафоров-счетчиков это, вообще говоря, неверно. Так, если бы в примере 7.7 на семафоре sem1 был бы установлен потолок приоритета, система должна была бы установить нити "производитель" высокий приоритет в момент захвата семафора. Но в коде этой нити нет освобождения этого семафора, так что нить сохраняла бы высокий приоритет вплоть до своего завершения!

Поэтому даже те системы, которые предоставляют наследование или потолки приоритетов для мутексов, не предоставляют аналогичных средств для семафоров-счетчиков. В книге [QNX 2004] приводятся результаты экспериментов над системой реального времени QNX, из которых видно, что эта ОС реализует наследование приоритетов на мутексах, но при использовании семафоров-счетчиков можно получить классический случай инверсии приоритета.

8.3. Пользовательские и ядерные нити

В предыдущих разделах мы избегали обсуждать вопрос о том, как в системах с виртуальной памятью нити связаны с процессами. В соответствии с принятой в современных системах семейства Unix и в Win32 терминологией, процесс — это то, что в старых системах называлось задачей, т. е. сущность, которой выделено собственное адресное пространство. Очевидно, что нить обязательно исполняется в каком-то адресном пространстве. Впрочем, в разные моменты времени одна и та же нить может исполняться в разных адресных пространствах.

В старых системах семейства Unix с каждым процессом была связана единственная нить исполнения. Легко понять, что планирование процессов требует переключения регистров диспетчера памяти, поэтому оно может осуществляться только ядром.

В большинстве современных и не столь современных ОС — в том числе, в современных ОС семейства Unix, Win32, OS/390, OS/2, VAX/VMS и во многих других — в рамках одного процесса может исполняться несколько нитей. На первый взгляд, было бы логично, если бы планирование этих нитей также осуществлялось ядром. Впрочем, можно привести аргументы и в поддержку другого решения.

Действительно, если в рамках процесса исполняется несколько нитей, то переключение между ними не требует перезагрузки регистров диспетчера памяти. То есть такие нити могут иметь урезанный (по сравнению с процессами) контекст и могли бы переключаться планировщиком, исполняющимся в пользовательском адресном пространстве. Это сократило бы количество переключений в режим ядра и могло бы повысить производительность.

Волокна Win32

В Win32 обычные нити (thread) планируются на уровне ядра. Однако по просьбе разработчиков приложений в Win32 API предусмотрены также пользовательские нити, называемые волокнами (fiber). Переключение между этими нитями осуществляется планировщиком, исполняющимся в пользовательском адресном пространстве. Планировщик фактически кооперативный — волокна переключаются при обращениях к примитивам синхронизации. Волокно всегда принадлежит к определенной нити. Важным недостатком волокон является тот факт, что исполнение большинства блокирующихся системных вызовов приводит к блокировке нити как целого, — пока система не вернет управление родительской нити, ни одно из волокон этой нити исполняться не будет.

Пользовательские нити и LWP в Unix System V Release 4

В Unix SVR4 и некоторых других системах семейства Unix реализована гибридная многопоточность. В рамках процесса исполняется одна или несколько единиц планирования уровня ядра — LWP (LightWeight Process — легковесный процесс). LWP одного процесса могут принадлежать разным классам планирования; например, некоторые из LWP могут исполняться с классом планирования реального времени, а другие — нет.

Реализации стандарта POSIX threads и некоторые другие реализации многопоточности (например, Solaris threads) надстраивают над LWP слой пользовательских нитей.

Планировщик POSIX thread library в Solaris вплоть до версии 8 работает в пользовательском контексте и активизируется при обращении активной нити к примитивам синхронизации (мутексам, семафорам и т. д.). Нить по умолчанию не привязана к определенному LWP, так что исполнение блокирующегося системного вызова в одном из LWP не обязательно приводит к блокировке процесса в целом. Если все LWP процесса исполняют блокирующиеся системные вызовы, планировщик пробуждается сигналом SIGWAITING и может создать дополнительный LWP.

Пользователь в определенных пределах может управлять привязкой нитей к LWP. Поэтому на практике нити делятся на непривязанные (unbound), которые планируются на любом свободном LWP, и привязанные (bound).

В Solaris 9 разработчики отказались от этой системы и перешли к одному LWP на каждую пользовательскую нить.

8.4. Монолитные системы и системы с микроядром

Не бывает монолитных программ, бывают плохо структурированные.

Реплика с семинара по ООП

Ходовая часть танка работает в экстремальных условиях и с трудом поддается модернизации.

Реплика в эхоконференции RU.WEAPON сети ФИДО

Исполняя системный вызов, пользовательская программа передает управление ядру. С понятием *ядра* — комплекса программ, исполняющихся в привилегированном (системном) режиме процессора, — мы уже сталкивались в разд. 4.5 и главе 5. Практически во всех современных системах ядро представляет собой единый процесс — единое адресное пространство. Но организация взаимодействия между нитями этого процесса в различных системах устроена по-разному.

Три основные группы нитей, исполняющихся в режиме ядра, — это, во-первых, обработчики прерываний, во-вторых, — обработчики системных вызовов со стороны пользовательских процессов и, в-третьих, — нити, исполняющие различные внутренние по отношению к системе работы, например, управление дисковым кэшем. В документации фирмы Sun нити этих трех групп в ядре Solaris называются, соответственно, исполняющимися в *контексте прерывания* (*interrupt context*), в *пользовательском контексте* (*user context*) и в *контексте ядра* (*kernel context*, или в контексте системы) [docs.sun.com 805-7378-10].

Далее мы будем называть последние две категории нитей, соответственно, пользовательскими и системными, хотя и те, и другие исполняют системный код в системном адресном пространстве.

Обработчики прерываний всегда представляют собой особую статью — система практически никогда не имеет контроля над тем, когда возникают внешние события, зато практически всегда обязана обрабатывать эти события по мере их появления. Поэтому обработчики прерываний получают управление по необходимости.

В то же время, порядок получения управления пользовательскими и системными нитями в ряде ситуаций находится под контролем системы. Планировщик является частью ядра и модули системы вольны включать его и выключать по мере необходимости. Практически всегда его выключают на время работы обработчика прерывания, но некоторые системы делают это и при активизации других нитей ядра.

Полное выключение планировщика на время работы ядра фактически означает, что система не реализует внутри ядра вытесняющей многозадачности. Системные и пользовательские нити в этом случае являются сопрограммами, а не нитями в полном смысле этого слова.

Эта архитектура, называемая *монолитным ядром*, привлекательна примерно тем же, чем привлекательна кооперативная многозадачность на пользовательском уровне, — любой модуль ядра может потерять управление лишь в двух случаях: при исполнении обработчика прерывания или по собственной инициативе. Благодаря этому разработчик модуля может не беспокоиться о критических секциях и прочих малоприятных аспектах доступа к разделяемым данным (кроме, разумеется, случаев, когда разделяет данные с обработчиком прерывания).

Платить за эти преимущества приходится значительными сложностями при переносе системы на многопроцессорные машины и невозможностью реализовать режим реального времени. Действительно, код ядра, написанный в расчете на кооперативную многозадачность, не может быть реентерабельным, поэтому такая система в то время, когда исполняется какая-то из ее нитей, не может обрабатывать системные вызовы. Следовательно, она не может иметь пользовательские процессы с приоритетом выше, чем у нитей ядра, — а именно такие процессы нужны для поддержки приложений реального времени.

Примечание

Те из читателей, кто когда-либо пытался реализовать обработку аппаратных прерываний под MS/DR DOS, должны быть хорошо знакомы с этой проблемой. Вопреки хакерскому фольклору, нереентерабельность ядра DOS связана вовсе не с переустановкой указателя стека при входе в обработчик прерывания 21h, а именно с тем, что ядро работает с разделяемыми данными, но не предоставляет собственных средств для взаимоисключения доступа к ним.

Одно из решений этой проблемы состоит в переходе к асинхронным системным вызовам. Именно так были устроены системы РВ первых поколений, такие, как DEC RT-11. Эти ОС имели кооперативное событийно-ориентированное ядро, а большинство системных вызовов, особенно запросы ввода-вывода, были организованы как асинхронные: в описании системного вызова указывалось, что к моменту возврата из него операция, как правило, еще не завершена; во многих случаях система даже не приступала к копированию данных из пользовательского буфера в свои внутренние кэши. При этом с каждым запросом ассоциировалась запись запроса (IRP, Input/output Request Packet — пакет запроса ввода/вывода) и локальный флаг события (род двоичного семафора). Просматривая IRP, прикладная программа могла отслеживать состояние своего запроса, а ожидая семафора — дождаться его завершения.

ния или хотя бы момента, когда она сможет использовать буфер с данными для других целей.

Синхронный и асинхронный ввод/вывод в RT-11, RSX-11 и VMS

Системный вызов ввода/вывода в этих ОС называется QIO (Queue Input/Output [Request] — [установить в] очередь запрос ввода/вывода) и имеет две формы: асинхронную QIO и синхронную QIOW (Queue Input/Output and Wait — установить запрос и ждать [завершения]). С точки зрения подсистемы ввода/вывода эти вызовы ничем не отличаются, просто при запросе QIO ожидание конца запроса выполняется пользовательской программой "вручную", а при QIOW выделение флага события и ожидание его установки делается системными процедурами пред- и постобработки.

Поскольку запрос QIO состоит только в постановке запроса в очередь, для его реализации необходимо защитить спинлоками только сам заголовок очереди (на VAX не требуется даже этого, потому что данный процессор имеет команду установки записи в очередь, которая исполняется в режиме монопольного захвата шины). В однопроцессорных системах (RT-11 поддерживала только такие машины) на время исполнения этой операции можно просто запретить прерывания. Благодаря этому, в RT-11 запросы на ввод/вывод можно было исполнять даже из обработчиков прерываний!

Впрочем, если нам не требуется реальное время и перед нами не стоит задача обеспечить равномерное распределение системных нитей между процессорами симметрично многопроцессорной машины, монолитное ядро вполне приемлемо. Большинство систем семейств Unix (фактически, все широко распространенные системы этого семейства, кроме System V R4) и Win32, OS/2 и ряд других систем общего назначения более или менее успешно ее используют.

Альтернативной монолитным ядрам является *микроядро*. Микроядерные системы реализуют вытесняющую многозадачность не только между пользовательскими процессами, но и между нитями ядра.

Микроядро QNX

Классическая реализация микроядра, QNX, состоит из вытесняющего планировщика и примитивов гармонического межпоточного взаимодействия, средств для обмена сообщениями *send* и *receive*. Эти примитивы, конечно же, сами по себе не могут быть реализованы реентерабельным образом, однако они крайне просты, содержат очень мало кода, исполняются быстро, и на время их исполнения система просто запрещает прерывания.

Все остальные модули системы с точки зрения микроядра представляют собой полностью равноценные нити. То, что некоторые из этих нитей исполняются в пользовательском, а другие — в привилегированном режиме доступа, микроядру совершенно неинтересно и не влияет на их приоритет и класс планирования.

QNX разрабатывался для приложений реального времени, в том числе и для использования во встраиваемых микропроцессорных системах; но, благодаря компактности и фантастической производительности, эта ОС иногда заменяет

системы общего назначения. Микроядро QNX действительно заслуживает названия микро, поскольку занимает всего 12 Кбайт кода и полностью входит в кэш первого уровня даже старых процессоров архитектуры x86. Все остальные модули ядра — драйверы внешних устройств, файловых систем, сетевых протоколов, имитация API систем семейства Unix — динамически загружаются и выгружаются и не являются обязательными (если, конечно, приложение не требует сервисов, предоставляемых этими модулями), благодаря чему ОС может использоваться во встраиваемых приложениях с весьма небольшим объемом ПЗУ.

Микроядро транспьютера

Другим примером классического микроядра является транспьютер. Микропрограммно реализованное микроядро транспьютера содержит планировщик с двумя уровнями приоритета и средства для передачи данных по линкам.

Микроядро Unix SVR4

Другие системы микроядерной архитектуры, например Unix System V Release 4.x (на этом ядре построены такие ОС, как Sun Solaris, SCO UnixWare, SGI Irix), предоставляют нитям ядра гораздо больше примитивов межпроцессного взаимодействия — в частности, мутексы. Ядро у этих систем в результате получается не таким уж "микро", но нашему определению это никоим образом не противоречит.

Важно подчеркнуть, что приведенное определение не имеет отношения к ряду других критерии, которые иногда (например, в дискуссиях в публичных компьютерных сетях, а нередко и в публикациях в более или менее серьезных журналах) ошибочно принимают за обязательные признаки микроядерной архитектуры. Отчасти эта путаница, возможно, создавалась целенаправленно, потому что в середине 90-х "микроядро" стало популярным маркетинговым слоганом, и поставщикам многих ОС монолитной или эклектичной архитектуры захотелось получить свою долю выгоды от возникшей шумихи.

Способ сборки ядра (динамическое или статическое связывание ядра с дополнительными модулями) и возможность динамической загрузки и выгрузки модулей без перезагрузки системы к микроядерности не имеют никакого отношения. Вполне микроядерная SCO UnixWare по умолчанию предлагает собирать ядро в единый загрузочный файл /stand/unix (хотя, впрочем, и поддерживает динамическую загрузку модулей). Напротив, не то, что монолитная, а кооперативно многозадачная Novell Netware замечательным образом умеет загружать и выгружать на ходу любые модули, в том числе и драйверы устройств (выгружать, разумеется, лишь при условии, что модуль никем не используется).

Один из корней этих заблуждений состоит в том, что в большинстве других контекстов антонимом "монолитной" архитектуры считается архитектура модульная. Таким образом, любой признак, свидетельствующий о том, что ядро ОС имеет модульную структуру, считается признаком микроядерности. В данном случае, однако, дилемма "монолитность/микроядерность" говорит не о наличии или отсутствии в ядре более или менее автономных модулей или подсистем, а о принципах взаимодействия между этими модулями или подсистемами или, точнее, об одном аспекте этого взаимодействия: о том, что в монолитных ядрах взаимодействие происходит синхронно или преимущественно синхронно, а в микроядре — асинхронно.

Совсем уж наивно было бы отказывать Solaris в праве называться микроядерным на том основании, что файл /kernel/genunix у этой системы имеет размер около 900 Кбайт. Ведь кроме собственно микроядра — планировщика нитей и примитивов взаимодействия между ними — этот файл содержит также диспетчер системных вызовов, систему динамической подгрузки других модулей ядра (см. разд. 3.11) и ряд других обязательных подсистем.

Микроядро концептуально очень привлекательно, но предъявляет к разработчикам модулей ядра известные требования. Например, в документе [docs.sun.com 805-7378-10] основное из этих требований — не забывать о том, что ядро Solaris многопоточное, и любая из нитей ядра может быть в любой момент вытеснена [практически любой] другой нитью — высказывается на второй странице, а выводам, которые из этого следуют, посвящена целая глава.

При разработке системы с нуля это само по себе не представляет проблемы, но если мы хотели бы обеспечить совместимость с драйверами внешних устройств и другими модулями ядра предыдущих версий ОС...

Из материала предыдущей главы легко понять, что код, рассчитанный на работу в однопоточной среде или среде кооперативной многозадачности, при переносе в многопоточную среду нуждается в значительной переработке, а нередко и в перепроектировании. Таким образом, сделать из монолитной (пусть даже модульной) системы микроядерную практически невозможно.

Следует учитывать, что требование поддержки многопроцессорных машин или приложений реального времени часто предъявляется к разработчикам через много лет после того, как были приняты архитектурные решения. В этой ситуации разработчики часто не переходят на микроядерную архитектуру полностью, а создают архитектуру гибридную (или, если применить более эстетский термин, эклектичную).

Действительно, как говорилось ранее, в чистом микроядре взаимодействия происходят асинхронно, а в чистом монолитном ядре — синхронно. Если некоторые из взаимодействий происходят асинхронно (что неизбежно, например, в многопроцессорной машине), то мы можем сказать, что система частично микроядерная. Если же некоторые из взаимодействий обязательно синхронны, мы, наверное, вынуждены будем признать, что наша система частично монолитная, как бы странно это ни звучало.

В зависимости от того, какого рода взаимодействия преобладают, мы можем выстроить целый спектр более или менее монолитных (и, напротив, более или менее микроядерных) архитектур. На практике большинство современных ОС общего назначения имеют гибридную архитектуру, которая не является микроядерной и в то же время не может быть классифицирована как монолитная. Многие из архитектур и, во всяком случае, многие из ключевых

принципов взаимодействия между модулями современных операционных систем были разработаны еще до того, как появилось само слово "микроядро". При этом некоторые из этих взаимодействий имеют синхронный, а некоторые — особенно взаимодействие с драйверами внешних устройств — асинхронный характер.

Во многих современных ОС широко применяется взаимодействие драйверов с остальной системой посредством очередей запросов (или событий). Такое взаимодействие отчасти стирает различия между синхронным и полностью асинхронным межмодульным взаимодействием.

Впрочем, при реализации системы РВ на основе микроядра мы не решаем проблему полностью. Действительно, если ядро реализовано в расчете на вытесняющую многозадачность, то мы можем исполнять системные вызовы из процессов, работающих с приоритетом более высоким, чем приоритет ядра. Однако если мы при этом исполним блокирующийся системный вызов, то это приведет к инверсии приоритета: запрос нити реального времени будет исполняться с системным приоритетом, который, по условию, ниже приоритета РВ. Поскольку время работы запросов ввода/вывода определяется временем реакции соответствующих внешних устройств, наследованием приоритета устраниТЬ эту проблему не получится, единственным выходом является переход от блокирующихся синхронных вызовов к очередям запросов. Поэтому даже микроядерные системы РВ вынуждены реализовать асинхронные системные вызовы ввода/вывода.

Вопросы для самопроверки

1. Сформулируйте основное отличие кооперативной многозадачности от вытесняющей. Почему в форумах и рекламных статьях вытесняющую многозадачность часто называют "настоящей"? Что "ненастоящего" в кооперативной многозадачности?
2. Какие проблемы затрудняют перенос программ, рассчитанных на кооперативную многозадачность, в среду с вытесняющей многозадачностью?
3. Какие проблемы возникают при переносе программ в обратном направлении?
4. Почему при кооперативном переключении нитей не обязательно сохранять все регистры центрального процессора?
5. Почему в системах разделенного времени повышают приоритет задачам, ориентированным на ввод/вывод? Постарайтесь развернуть ответ

на этот вопрос с учетом материала предыдущих глав, вплоть до определения систем разделенного времени.

6. Почему инверсия приоритета особенно опасна для приложений реального времени?
7. При использовании каких примитивов межпоточного взаимодействия возможна инверсия приоритетов? Постарайтесь сформулировать критерий, которому должны соответствовать эти средства, а не просто перечислить известные вам типы примитивов.
8. Почему средства борьбы с инверсией приоритета реализуют для мутексов, но не для семафоров?
9. Почему ядро ОС обычно нереентерабельно? Что можно сделать, чтобы оно стало реентерабельным?
10. Верно ли, что система реального времени должна быть микроядерной? Приведите примеры ОС реального времени с другими архитектурами ядра.



ГЛАВА 9

Внешние устройства

Митрофан. Эта? Прилагательна.

Правдни. Почему ж?

Митрофан. Потому что она приложена к своему месту. Вон у чулана шеста неделя дверь стоит еще не навешена: так та покамест существительна.

Д. Фонвизин

Все без исключения приложения вычислительных систем так или иначе связаны с использованием *внешних*, или *периферийных устройств*. Даже чисто вычислительные задачи нуждаются в устройствах для ввода исходных данных и вывода результата. Без преувеличения можно сказать, что процессор, не имеющий никаких внешних устройств, абсолютно бесполезен.

У вычислительных систем первых поколений набор периферийных устройств часто исчерпывался упомянутыми устройствами для ввода исходных данных и вывода результата вычислений, поэтому до сих пор модули ОС, работающие с периферией, называют *подсистемой ввода/вывода (input/output subsystem)*. У большинства современных компьютеров набор внешних устройств весьма обширен, и функции многих из них не могут или лишь с определенной натяжкой могут быть описаны как ввод и вывод.

С функциональной точки зрения внешние устройства, подключаемые к современным компьютерам, можно разделить на следующие категории (приведенную классификацию вряд ли можно считать исчерпывающей, а порядок перечисления не является критерием важности данного типа устройств):

- Устройства внешней памяти, которые в свою очередь, можно разделить на два класса:
 - устройства памяти с произвольным доступом, главным образом магнитные диски. К этому же классу относятся дискеты, магнитооптические и оптические диски, практически не применяемые в настоящее время магнитные барабаны и накопители на основе флэш-памяти (ЭСППЗУ с возможностью блочного перепрограммирования). Удачным

универсальным обозначением для этого класса устройств является принятое в документации фирмы IBM сокращение *DASD* (Direct Access Storage Device — запоминающее устройство прямого доступа);

- устройства памяти с последовательным доступом. В основном, это лентопротяжные устройства (стриммеры и др.);
- сетевые и телекоммуникационные устройства;
- устройства алфавитно-цифрового ввода/вывода: печатающие устройства, телетайпы, текстовые терминалы;
- устройства звукового ввода/вывода;
- устройства графического ввода/вывода: сканеры или видеодекодеры (ввод), графические дисплеи, плоттеры, графические принтеры или видеокодеры (вывод);
- позиционные устройства ввода: мыши, планшеты-дигитайзеры, световые перья и т. д.;
- сенсорные и исполнительные устройства управляющих систем.

Например, у бортового компьютера самолета сенсорными устройствами могут являться гироскопы или другие датчики ориентации, трубка Пито (датчик, определяющий скорость самолета относительно воздуха), радар и терминал глобальной системы позиционирования, а исполнительными устройствами — шаговые электромоторы, управляющие рулевыми плоскостями, топливные насосы двигателей и т. д.

Все перечисленные устройства либо передают информацию центральному процессору (и, таким образом, могут быть объявлены *устройствами ввода*), либо получают информацию от него (*устройства вывода*), либо могут как передавать, так и принимать информацию (*устройства ввода/вывода*). Эта классификация может показаться неестественной, потому что в соответствии с ней в одну категорию попадают столь функционально неродственные устройства, как сетевой адаптер и жесткий диск (*устройства ввода/вывода*), или печатающее устройство и рулевая машинка летательного аппарата (*устройства вывода*), однако разработчику операционной системы во многих случаях этой классификации оказывается достаточно.

Нередко, впрочем, в эту классификацию вводят еще один уровень: устройства ввода делят на пассивные (выдающие данные только в ответ на явные запросы центрального процессора) и активные, или *генераторы событий*, которые могут порождать данные тогда, когда их об этом явно не просили. Ко второй категории относятся интерактивные устройства ввода (клавиатура, мышь), сетевые адAPTERы, таймеры различного рода, а также многие датчики управляющих систем.

9.1. Доступ к внешним устройствам

С точки зрения центрального процессора и исполняющейся на нем программы, внешние устройства представляют собой наборы специализированных ячеек памяти или, если угодно, регистров. У микроконтроллеров эти ячейки памяти представляют собой регистры центрального процессора.

У процессоров общего назначения регистры устройств обычно подключаются к шинам адреса и данных ЦПУ. Устройство имеет адресный дешифратор. Если выставленный на шине адрес соответствует адресу одного из регистров устройства, дешифратор подключает соответствующий регистр к шине данных (рис. 9.1). Таким образом, регистры устройства получают адреса в физическом адресном пространстве процессора.

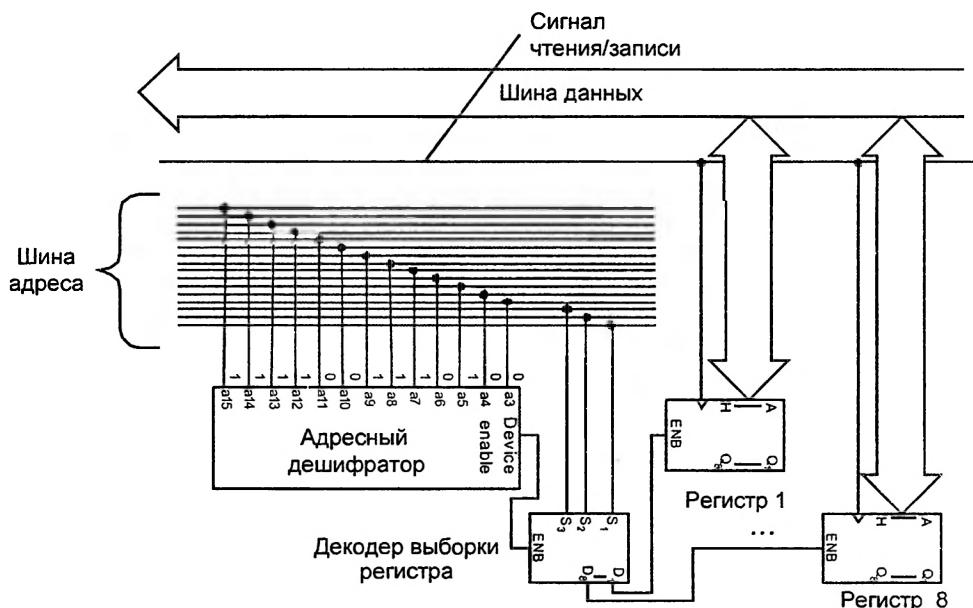


Рис. 9.1. Подключение внешнего устройства с восемью регистрами к шине

Два основных подхода к адресации этих регистров — это отдельное *адресное пространство ввода/вывода и отображенный в память ввод/вывод (memory-mapped I/O)*, когда память и регистры внешних устройств размещаются в одном адресном пространстве.

В первом случае для обращения к регистрам устройств применяются специальные команды IN и OUT; используемые в этих командах адреса могут совпадать с адресами ячеек памяти, но указывают на объекты в собственном ад-

ресном пространстве. Например, процессоры x86 имеют 45-разрядный виртуальный адрес, 32-разрядный физический адрес памяти (36-разрядный у Р III и более новых процессоров) и 16-разрядное адресное пространство ввода/вывода.

Во втором случае могут использоваться любые команды, способные работать с операндами в памяти, но регистры ввода/вывода занимают адреса, которые могли бы быть использованы для адресации ОЗУ. Так, PDP-11 имеет 16-разрядный адрес и может, таким образом, адресовать не более 64 Кбайт ОЗУ (у поздних моделей, таких как PDP-11/20, физическое адресное пространство расширено до 2 Мбайт, но каждому процессу все равно доступно не более 64 Кбайт). В компьютерах с шиной Unibus и Qbus последние 4 Кбайт из этих адресов заняты под страницу ввода/вывода, так что под оперативную память остается всего 60 Кбайт.

В принципе, никто не запрещает использовать отображеный на память ввод/вывод в системах на основе процессоров, реализующих отдельное адресное пространство. Так, в IBM PC-совместимых компьютерах большая часть управляющих регистров внешних устройств расположена в адресном пространстве ввода/вывода, но видеобуфер графического адаптера отображен на память. В оригинальном IBM PC предполагалась установка не более 640 Кбайт ОЗУ, а оставшиеся 384 Кбайт адресного пространства 8088 резервировались для отображенных на память устройств и ПЗУ. 128 Кбайт занимал видеобуфер, который отображался двумя разными способами: на адреса 0xA0000—0xFFFF в текстовом режиме и на адреса 0xB0000—0xBFFFF в графическом. Ряд плат расширения для шины ISA (сетевые платы NE2000, некоторые SCSI HBA и звуковые карты, платы расширения памяти LIM XMS) также использовали отображенные на память буферы для обмена данными.

В PC-совместимых компьютерах с шиной PCI четвертый гигабайт адресного пространства отведен под отображенные на память периферийные устройства. В первую очередь это, разумеется, видеобуфера графических адаптеров, но отображение на память используется и другими важными устройствами. Так, в современных PC-совместимых машинах используются контроллеры прерываний, совместимые с Intel APIC. Управляющие регистры этих контроллеров отображены на адреса ОЗУ — 0fec00000 для локальных контроллеров процессора и 0xfc0xy00 для IOAPIC в многопроцессорных материнских платах.

Как правило, даже в случае раздельных адресных пространств, для обмена данными с памятью и внешними устройствами процессор использует одни и те же шины адреса и данных, но имеет дополнительный сигнал адресной шины, указывающий, какое из адресных пространств используется в данном конкретном цикле.

Любопытный гибридный подход, сочетающий преимущества обоих вышеназванных, предоставляют микропроцессоры с системой команд SPARC v9.

У этих процессоров команды Load и Store имеют поле, служащее селектором адресного пространства. Этот селектор, в частности, может использоваться для выбора адресного пространства памяти или ввода/вывода. Благодаря этому, с одной стороны, можно применять для работы с регистрами портов любые команды работы с памятью, как при отображенном в память вводе/выводе, и в то же время полностью задействовать адресное пространство для работы с ОЗУ.

Два основных подхода к выделению адресов внешним устройствам — это *фиксированная адресация*, когда одно и то же устройство всегда имеет одни и те же адреса регистров, и *географическая адресация*, когда каждому разъему периферийной (или системной, если внешние устройства подключаются непосредственно к ней) шины соответствует свой диапазон адресов (рис. 9.2). Географически можно распределять не только адреса регистров, но и другие ресурсы — линии запроса прерывания, каналы ПДП.

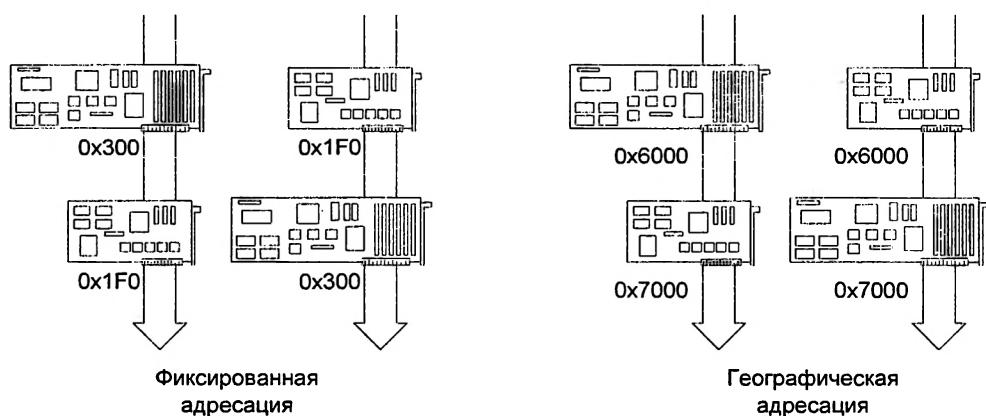


Рис. 9.2. Фиксированная и географическая адресация

Географическая адресация обладает свойством, которое на первый взгляд кажется противоестественным: перемещение платы устройства в другой разъем приводит к необходимости переконфигурации ОС (а в некоторых случаях, например, если перемещенная плата была контроллером загрузочного диска, а вторичный загрузчик или процедура инициализации ядра недостаточно сообразительны, может даже привести к ошибкам при загрузке). Однако этот способ распределения адресного пространства удобен тем, что исключает возможность конфликта адресов между устройствами разных производителей или между двумя однотипными устройствами (с этой проблемой должен

быть знаком каждый, кто пытался одновременно установить в компьютер сетевую и звуковую карты конструктива ISA). Большинство *периферийных шин* современных мини- и микрокомпьютеров, такие как PCI, S-Bus и др., в той или иной форме реализуют географическую адресацию.

Многие современные конструктивы, например уже упоминавшаяся шина PCI, требуют, чтобы кроме регистров управления и данных устройства имели также конфигурационные регистры, через обращение к которым ОС может получить информацию об устройстве: фирму-изготовителя, модель, версию, количество регистров и т. д. Наличие таких регистров позволяет ОС без вмешательства (или с минимальным вмешательством) со стороны администратора определить установленное в системе оборудование и автоматически подгрузить соответствующие управляющие модули.

9.2. Простые внешние устройства

У кошки четыре ноги —
Вход, выход, земля и питание.

По-видимому, самым простым из мыслимых (а также и из используемых) внешних устройств является *порт вывода*. Такие устройства являются стандартным компонентом большинства микропроцессорных систем. У микропроцессоров первых поколений порты реализовались отдельными микросхемами, у современных микроконтроллеров они обычно интегрированы в один кристалл с процессором.

Порт вывода представляет собой регистр и несколько выходных контактов, называемых на жаргоне микроэлектронщиков "ногами". В литературе входные и выходные контакты микросхем обычно называют просто входами и выходами. Количество выходов порта, как правило, соответствует и никогда не превосходит количества бит в регистре. Если в бит регистра записан ноль, напряжение на выходе порта будет низким, а если единица, то, соответственно, высоким. Большинство современных микропроцессорных комплектов используют так называемые *ТТЛ-совместимые напряжения*, когда нулю соответствует напряжение 0 В, а единице — 5 В или, при работе от источника питания с более низким напряжением, напряжение этого самого источника (рис. 9.3).

Поскольку и внутри процессора для представления нулей и единиц также используется высокое и низкое напряжение, порт полезен прежде всего тем, что имеет регистр — чтобы удержать напряжение на выходе, процессору достаточно один раз установить значение регистра.

Кроме того, польза от порта состоит в том, что поддержание напряжения на выходе может потребовать пропускания через этот выход тока. Порты совре-

менных микроконтроллеров имеют внутреннее сопротивление около 200 Ом и способны без вреда для себя пропустить через вывод ток до 25 мА. Выводы шин адреса и данных микропроцессоров, как правило, рассчитаны на гораздо меньшие токи. Кроме того, порты часто имеют встроенные механизмы защиты от короткого замыкания, статического электричества и т. д.

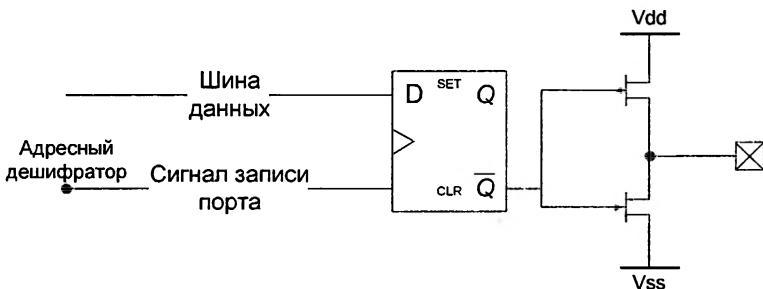


Рис. 9.3. Принципиальная схема ТТЛ-совместимых выходных каскадов порта вывода (здесь и далее Vss — высокое напряжение питания, Vdd — ноль)

Применения порта вывода многообразны. Например, к нему можно присоединить светодиод и получить лампочку, миганием которой можно программно управлять (такие диоды часто используются при отладке программ для микроконтроллеров — вместо диагностической печати).

Присоединив к выводу порта динамик, можно издавать различные звуки — впрочем, чтобы получился именно звук, а не одиночный щелчок, центральный процессор должен периодически записывать в соответствующий бит порта то ноль, то единицу. Применяя широтно-импульсную модуляцию (т. е. манипулируя относительными продолжительностями периодов высокого и низкого напряжений), таким способом можно генерировать не только прямоугольный меандр, но и сигналы более сложной формы и, соответственно, звуки различного тембра. Используя переменный период, можно генерировать сигналы с частотой, не кратной тактовой частоте центрального процессора или таймера, применяемого для синхронизации (пример 6.1). Изменяя период сигнала с помощью достаточно сложных алгоритмов, можно даже получать отношение тактовой частоты и частоты сигнала, равное иррациональному числу.

Впрочем, сигналы, порожденные с помощью вышеперечисленных приемов, будут состоять не только из сигнала заданного тембра и частоты, но и из гармоник тактовой частоты. Если последняя частота достаточно высока, с этим можно смириться в надежде на то, что динамик не сможет ее воспроизвести, а слушатель — уловить.

Кроме того, к выводу порта можно присоединить внешнюю цифровую, аналоговую или электромеханическую схему (например, шаговый электродвигатель), которая будет выполнять какую-то полезную работу. Таким образом, порт вывода, как правило, не является внешним устройством сам по себе, а служит интерфейсом между микропроцессором и собственно внешним устройством.

Другое столь же простое устройство — это *порт ввода*. Порт ввода также состоит из регистра и нескольких входных линий, соответствующих битам регистра (рис. 9.4). Бит регистра имеет значение 0, если на вход подано низкое напряжение (точнее говоря, если во время последнего цикла тактового генератора порта на вход было подано низкое напряжение), и, наоборот, единицу — если высокое. Понятно, что напряжение практически никогда не соответствует в точности 0 или 5 В, поэтому в спецификациях портов ввода всегда указывают диапазон напряжений, которые считаются нулем (например, от 0 до 0.2 В) и единицей (например, от 4.5 до 5 В), для промежуточных же напряжений значение соответствующего бита не определено (а на практике определяется случайными факторами). Регистр порта ввода часто называют *регистром-защелкой* (latch register), потому что основная его функция — зафиксировать напряжения на входах в определенный момент времени и передать их центральному процессору в виде однозначно (пусть и негарантированно правильно) определенных значений.

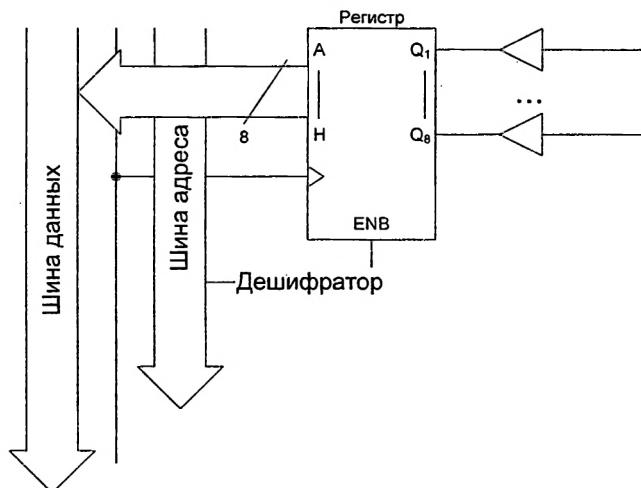


Рис. 9.4. Порт ввода

Разработчики микросхем часто совмещают входы портов ввода и выходы портов вывода, создавая таким образом комбинированное устройство —

порт ввода/вывода. Такое устройство должно быть существенно сложнее, чем простая комбинация порта ввода и порта вывода. Если порт вывода пытается установить на определенном выходе высокое напряжение, а другое устройство пытается установить низкое, в соответствии с законом Ома это приведет к возникновению электрического тока. Внутреннее сопротивление типичного микроконтроллерного порта составляет около 200 Ом. При разности напряжений в 5 В это соответствует упоминавшемуся ранее предельно допустимому току в 25 мА: таким образом, устройство с нулевым внутренним сопротивлением все-таки может выставить ноль, но попытка сделать это на нескольких линиях одновременно приведет к перегрузке схем питания контроллера. В любом случае, большие токи приводят к потерям энергии и разогреву схемы, поэтому без крайней необходимости их лучше избегать. Если мы хотим использовать одни и те же "ноги" микросхемы как для ввода, так и для вывода, мы должны иметь возможность контролируемого отключения выходных каскадов порта вывода.

Отключаемые выходные каскады называются тристабильными, а третье, отключенное, состояние выхода — высокоимпедансным. Тристабильные выходы используются не только для реализации двунаправленных контактов микросхем, но и для подключения устройств к шине: устройство, перешедшее в третье состояние, освобождает шину и позволяет какому-то другому задатчику выставлять на ней свои данные. Для управления переводом выходов в высокоимпедансное состояние порт ввода/вывода должен иметь еще один регистр, называемый регистром или маской направления данных (data direction register) (рис. 9.5).

Единица в разряде этого регистра обычно соответствует переводу соответствующего вывода в третье состояние, дающее возможность использовать этот контакт для ввода. Таким образом, даже такое простое устройство, как порт ввода/вывода, имеет целых три регистра: два регистра данных (многие реализации портов размещают эти два регистра по одному адресу: при чтении обращение происходит к защелке порта ввода, а при записи — к регистру порта вывода) и один управляющий. Более сложные устройства обычно также имеют один или несколько регистров данных и один или несколько управляющих регистров. Устройства, передающие и принимающие большие объемы данных (контроллеры жестких дисков, сетевые интерфейсы, видеoadаптеры), часто вместо одного регистра данных снабжаются буфером памяти, отображенном на адреса памяти процессора.

Вместо управляющих регистров у некоторых сложных устройств есть командный регистр. Центральный процессор передает через этот регистр последовательность команд, а устройство их исполняет и, возможно, передает последовательность ответов.

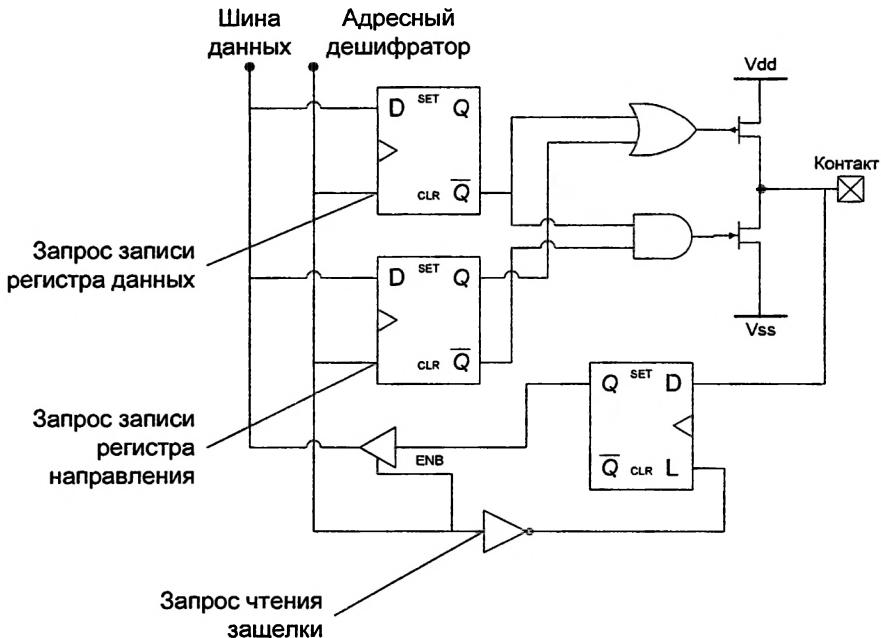


Рис. 9.5. Принципиальная схема порта ввода/вывода

9.3. Порты передачи данных

Порты ввода/вывода преимущественно используются для управления простыми внешними устройствами: если бит установлен, мотор крутится (заслонка открыта, нагреватель включен и т. д.), и наоборот. Если же устройство более сложное, и работа с ним предполагает обмен последовательностями команд и ответов, или просто большими объемами данных, простой порт оказывается не очень удобен.

Основная проблема при использовании простого порта в качестве средства обмена данными состоит в том, что принимающему устройству необходимо знать, выставило ли передающее устройство на своих выходах новую порцию данных, или еще нет. Три основных подхода к решению этой проблемы называются синхронной, асинхронной и изохронной передачами данных.

При *синхронной передаче* мы либо предоставляем дополнительный сигнал, *строб* (рис. 9.6), либо тем или иным способом передаем синхросигналы по тем же проводам, что и данные. Например, можно установить, что каждая следующая порция данных должна хотя бы одним битом отличаться от предыдущей. При этом необходимо предусмотреть протокол, посредством кото-

рого передатчик будет кодировать, а приемник декодировать повторяющиеся последовательности символов. Например, второй символ из пары одинаковых последовательных символов можно заменять на специальный символ повторения; впрочем, в этом случае нам необходимо предусмотреть и способ кодирования символа, совпадающего по значению с символом повторения. Реальные способы совмещения кодирующих и синхронизующих сигналов в одном проводе относительно сложны и их детальное обсуждение было бы более уместно в книге, посвященной сетевым технологиям. Некоторые простые способы кодирования с таким совмещением мы рассмотрим в разд. 9.6.

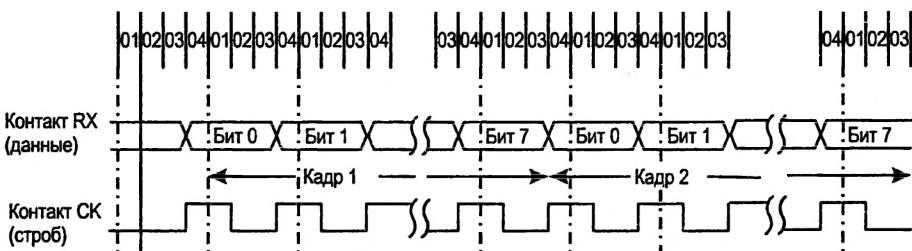


Рис. 9.6. Временная диаграмма стробируемого порта

Передача стробирующего сигнала требует прокладки дополнительных проводов, но с реализацией точки зрения гораздо проще совмещения синхро-сигнала и данных и поэтому широко применяется в самых разнообразных ситуациях. Большинство стробируемых портов асимметричны: одно из устройств, *ведущий* (*master* — хозяин), генерирует стробовый сигнал, а второе, *ведомый* (*slave* — раб), пользуется этим сигналом для приема или передачи. Нередко вместо одиночного строба используется несколько различных сигналов — например, один сигнал выставляется передатчиком и сообщает, что следующая порция данных готова, а второй сигнал — приемником и сообщает, что приемник принял эти данные и готов принять следующие. Дополнительные сигналы могут также решать вопрос о том, какое из устройств в данном цикле будет приемником, а какое — передатчиком.

При *асинхронном обмене* данными передающее устройство посылает специальный стартовый символ, сигнализирующий о том, что сейчас пойдут данные, и с фиксированным интервалом выставляет на своих выходах символы данных. Передаваемый за один прием блок данных обычно невелик по объему — ведь необходимо считаться с опасностью того, что часы приемника и передатчика, посредством которых они отмеряют интервал между последовательными порциями данных, недостаточно точны и могут разойтись. Обычно блок данных состоит из фиксированного количества символов и называется *кадром* или *фреймом* (*frame*). Кадр обычно завершается одним или несколь-

кими стоповыми символами. Не обнаружив этих символов (или обнаружив на месте этих символов неверные значения), приемник может понять, что его часы все-таки разошлись с часами передатчика.

Асинхронная передача позволяет сэкономить на проводах (не требуется стробирующих сигналов) и при этом избежать сложных способов кодирования, характерных для совмещенной синхронной передачи, однако стартовые и стоповые символы составляют значительную часть потока передаваемых данных и создают ощутимые накладные расходы.

Кроме того, при передаче большого объема данных в виде плотно следующих друг за другом кадров велика опасность, что приемник потеряет заголовок очередного кадра и не сможет восстановить структуру потока, поэтому многие асинхронные протоколы требуют паузы между последовательными кадрами.

Асинхронная передача данных удобна в ситуациях, когда объем передаваемых данных невелик, а потребность в их передаче возникает в труднопредсказуемые моменты. Обычно асинхронные порты работают на небольших скоростях, не более нескольких килобит в секунду.

Изохронная передача данных по идеи напоминает асинхронную, с тем лишь отличием, что при обмене данными приемник и передатчик пользуются высокостабильными, но независимыми — при использовании одного тактового генератора получится синхронная передача — тактовыми генераторами, и благодаря этому могут обмениваться кадрами большого размера. В идеале, изохронная передача сочетает преимущества синхронной и асинхронной, но на практике сложности обеспечения стабильности и калибровки тактовых генераторов достаточно велики и в чистом виде изохронная передача используется очень редко.

Во многих документах под словами "изохронная передача" подразумевают протокол более высокого уровня, обеспечивающий передачу данных с сохранением временных соотношений. Такие протоколы могут быть полезны при передаче аудио- и видеоданных в реальном времени.

Как синхронные, так и асинхронные порты бывают следующих типов (рис. 9.7):

- симплексные* (*simplex* — передавать данные может только одно устройство);
- полудуплексные* (*half-duplex* — оба устройства могут принимать и передавать данные, но не способны делать это одновременно, например, потому, что прием и передача идут по одному проводу);
- полнодуплексные* (*full-duplex*) или просто дуплексные (оба устройства способны одновременно передавать и принимать данные, чаще всего — по различным проводам).

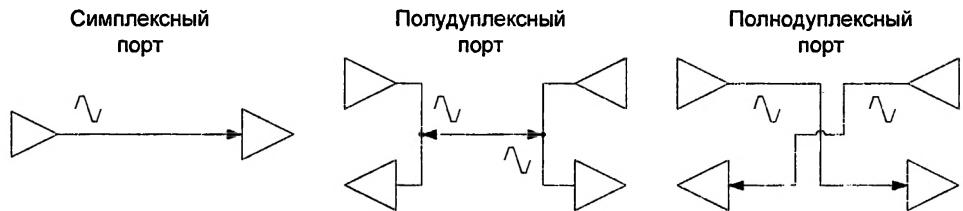


Рис. 9.7. Симплексные, полудуплексные и полнодуплексные порты

Еще одна практически важная классификация портов передачи данных — это деление их на последовательные и параллельные порты (рис. 9.8).

Последовательный порт состоит из одного провода, по которому, как следует из названия, последовательно передаются биты данных, а также, возможно,

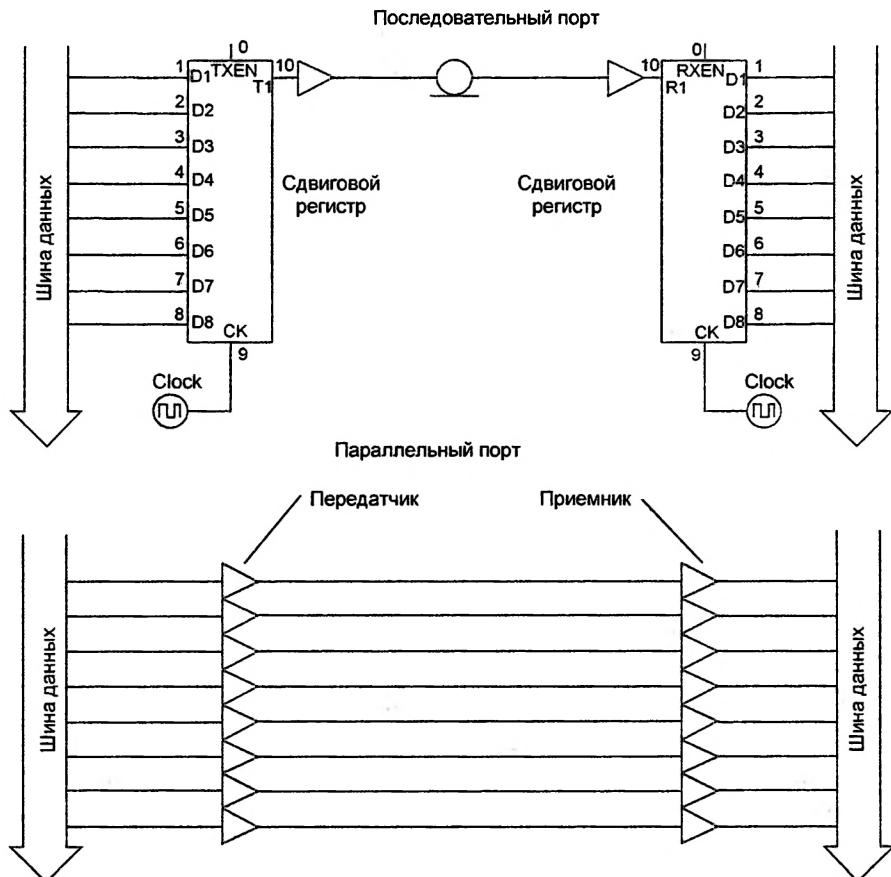


Рис. 9.8. Последовательные и параллельные порты

синхронизационные или стартовые и стоповые биты. Параллельный порт имеет несколько линий передачи данных, обычно 8 (чтобы можно было передать за один прием один байт), а иногда и больше.

Как правило, последовательные порты — асинхронные или синхронные с совмещенной передачей синхросигнала, но исключения из этого правила — стробируемые последовательные порты — также нередки. Напротив, если уж мы проложили восемь проводов для данных, то экономить на девятом — синхронизирующем проводе — было бы совсем уж глупо, поэтому подавляющее большинство практически используемых параллельных портов — синхронные стробируемые.

Теоретически, параллельные порты обеспечивают более высокую скорость передачи данных, чем реализованные по той же технологии последовательные. На практике, при действительно высоких скоростях передачи, разработчики устройств с последовательными портами сталкиваются с проблемой битового перекоса (*bit skew*). Эта проблема состоит в том, что если длина проводов различается, то сигналы на разных линиях порта будут прибывать к приемнику в разное время, т. е. будут рассинхронизованы друг с другом и со стробирующим сигналом. Наибольшие сложности это создает при разработке печатных плат.

Из-за этого в последние годы наметилась тенденция к отказу от параллельных портов и шин и к их замене высокоскоростными последовательными портами. Примерами такой замены являются обсуждаемые далее в этой главе Parallel SCSI и SAS (Serially Attached SCSI), Parallel ATA и SATA, PCI и PCI Express.

Порт RS232

Протокол последовательного асинхронного обмена RS232/CCITT V24 широко применяется для подключения к компьютеру алфавитно-цифровых терминалов, низкоскоростных печатающих устройств, позиционных устройств ввода (мышей, планшетов), низкоскоростного телекоммуникационного оборудования и т. д., а иногда и для соединения компьютеров между собой, например, если более скоростное сетевое оборудование отсутствует или не может быть использовано. В настоящее время этот интерфейс быстро вытесняется последовательной шиной USB.

Такие порты используются для передачи данных за пределы корпуса компьютера, поэтому кроме линии передачи данных предусмотрена также провод, передающий опорное нулевое напряжение. Вместо TTL-совместимых напряжений, RS232 использует в качестве 1 напряжения в диапазоне от -25 до -3 В, а в качестве 0 — соответственно, в диапазоне от +3 до +25 В (рис. 9.9).

RS232 предполагает двусторонний обмен данными. Для этой цели предусмотрено две линии данных — для приема и для передачи, обозначаемые TX и RX. В соответствии со стандартом, устройства делятся на два типа: "компьютеры" и "терминалы". Различие между ними состоит в том, что "компьютер" передает данные по линии TX, а получает по линии RX, а "терминал" — наоборот. Для

соединения двух "компьютеров" необходим специальный, так называемый нуль-модемный кабель, в котором провода TX и RX перекрещены. Любопытно, что стандартные порты IBM PC являются "терминалами", а не "компьютерами".

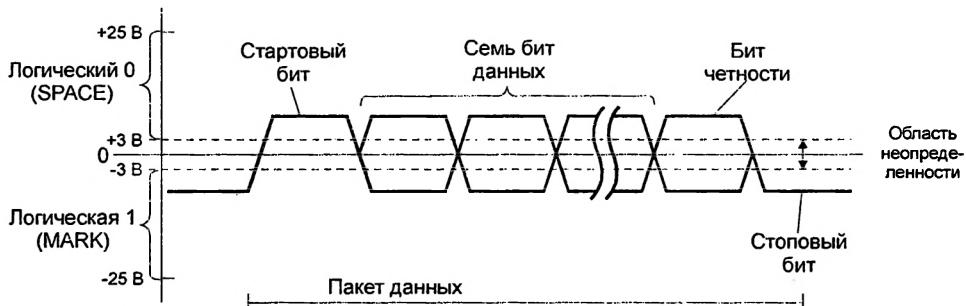


Рис. 9.9. Диаграмма напряжений RS232

Обмен данными осуществляется кадрами, состоящими из стартового бита, семи или восьми битов данных (младший бит передается первым), возможно — контрольного бита четности (см. разд. 1.7), и одного или двух стоповых битов. Игнорируя ошибки четности или вообще не проверяя четность, можно использовать этот бит для передачи данных и получить, таким образом, девять битов данных в одном кадре.

Минимальная скорость передачи составляет 300 бит/с, последующие допустимые скорости получаются удвоением предыдущей — 600, 1200, 2400 бит/с и т. д. Современные реализации RS232 поддерживают скорости 115 200 бит/с и более. Скорость и вариации формата кадра определяются настройками приемника и передатчика. Необходимо, чтобы у соединенных портом устройств эти настройки совпадали, однако протокол сам по себе не предоставляет средств для их согласования.

Кроме линий приема, передачи, нуля и питания спецификация RS232 предусматривает ряд дополнительных сигналов, в просторечии называемых *модемными линиями* — признак несущей, разрешение передачи данных (очисткой этого сигнала приемник может сигнализировать передатчику, что он не успевает обрабатывать поступающие данные) и др. Эти сигналы не должны обязательно поддерживаться всеми устройствами и применяются, главным образом, акустическими модемами, откуда и происходит название. Полная спецификация при использовании 25-контактного разъема предусматривает также возможность синхронной передачи данных с отдельными стробирующими сигналами, но основная масса реализаций RS232 этого не поддерживает.

Протокол RS232 весьма прост и легко может быть реализован программными средствами с использованием двух битов порта ввода/вывода (пример 9.1) — впрочем, в этом случае потребуется еще внешняя микросхема приемопередатчика, преобразующая TTL-совместимые напряжения в диапазон напряжений RS232. Однако порты, использующие этот протокол, применяются очень широко, а многие модели микроконтроллеров и практически все комплекты вспомогательных микросхем для микропроцессоров предлагают аппаратные реализации этого стандарта.

Пример 9.1. Программная имитация RS232-совместимого последовательного порта, цит. по [www.atmel.com AVR305]

```
;***** APPLICATION NOTE AVR305 *****
;*
;* Название      : Полудуплексный программный UART
;* Версия       : 1.20
;* Последнее обновление : 97.08.27
;* Целевое устройство : Все микроконтроллеры AVR
;*
;* Адрес поддержки : avr@atmel.com
;*
;* Размер кода    : 32 слова
;* Мин. Регистров : 0
;* Макс. Регистров : 4
;* Прерывания   : Не используются
;*
;* Описание
;* Этот пример содержит эффективный с точки зрения объема кода
программный UART.
;* Программа-пример получает один символ и передает его назад.
;***** .include "1200def.inc"
;***** Определения контактов
.equ RxD =0 ;Контакт приема PD0
.equ TxD =1 ;Контакт передачи PD1
;***** Глобальные регистровые переменные
.def bitcnt =R16 ;счетчик битов
.def temp =R17 ;промежуточный регистр
.def Txbyte =R18 ;Передаваемые данные
.def Rxbyte =R19 ;Полученные данные
.cseg
.org 0
```

```
;*****
;*
;* "putchar"
;*
;* Эта подпрограмма передает байт из регистра "Txbyte"
;* Количество стоповых битов определяется константой sb
;*
;* Количество слов :14 включая возврат
;* Количество циклов :Зависит от частоты передачи
;* Мин. Регистров :Нет
;* Макс. Регистров :2 (bitcnt,Txbyte)
;* Указатели :Не используются
;*
;*****
.equ sb =1 ;Кол-во стоповых битов (1, 2, ...)

putchar: ldi bitcnt,9+sb ;1+8+sb
    com Txbyte ;Инвертировать все
    sec ;Стартовый бит

putchar0: brcc putchar1 ;Если перенос установлен
    cbi PORTD,TxD ;передать '0'
    rjmp putchar2 ;иначе

putchar1: sbi PORTD,TxD ; передать '1'
    nop

putchar2: rcall UART_delay ;Задержка в один бит
    rcall UART_delay

    lsr Txbyte ;Получить следующий бит
    dec bitcnt ;Если не все биты переданы
    brne putchar0 ; послать следующий
        ;иначе
    ret ; возврат

;*****
;*
;* "getchar"
;*
```

```
;* Эта подпрограмма получает один байт и возвращает его в "Rxbyte"
;*
;* Кол-во слов :14 включая возврат
;* Кол-во циклов :Зависит от скорости приема
;* Мин. Регистров :Нет
;* Макс. Регистров :2 (bitcnt,Rxbyte)
;* Указатели :Не используются
;*
;*****
```

getchar: ldi bitcnt,9 ;8 бит данных + 1 стоповый

getchar1: sbic PIND,RxD ;Ждать стартового бита
rjmp getchar1

rcall UART_delay ;задержка в 0.5 бита

getchar2: rcall UART_delay ;задержка в один бит
rcall UART_delay

clc ;очистить перенос
sbic PIND,RxD ;если вход RX = 1
sec

dec bitcnt ;Если бит стоповый
breq getchar3 ; возврат
;иначе
ror Rxbyte ; сдвинуть бит в Rxbyte
rjmp getchar2 ; получить следующий

getchar3: ret

```
;*****
```

;*

;* "UART_delay"

;*

;* Эта подпрограмма задержки генерирует требуемую задержку между битами
;* при передаче и приеме байтов. Полное время исполнения определяется
;* константой "b":
;*

;*
;*3*b + 7 cycles (включая rcall и ret)

```
/*
;* Кол-во слов :4 включая возврат
;* Мин. Регистров :Нет
;* Макс. Регистров :1 (temp)
;* Указатели :Не используются
;*
;***** Допустимые значения b:
;
; 1 MHz crystal:
; 9600 bps - b=14
; 19200 bps - b=5
; 28800 bps - b=2
;
; 2 MHz crystal:
; 19200 bps - b=14
; 28800 bps - b=8
; 57600 bps - b=2
;
; 4 MHz crystal:
; 19200 bps - b=31
; 28800 bps - b=19
; 57600 bps - b=8
; 115200 bps - b=2

.equ b =31 ;19200 bps @ 4 MHz crystal

UART_delay: ldi temp,b
UART_delay1: dec temp
    Brne UART_delay1

    ret

;***** Исполнение программы начинается здесь

;***** Тестовая программа

reset: sbi PORTD,TxD ;Установить контакты порта
      sbi DDRD,TxD
```

```

ldi Txbyte,12 ;Очистить терминал
rjmp putchar

forever: rcall getchar
        mov Txbyte,Rxbyte
        rcall putchar ;Воспроизвести полученный символ
        rjmp forever

```

USART микроконтроллера PIC

В качестве примера аппаратной реализации последовательного порта, которую, помимо других вариантов использования, можно применить и для реализации RS232, давайте рассмотрим устройство USART (Universal Synchronous Asynchronous Receiver Transmitter), включенное в состав ряда моделей микроконтроллеров PIC фирмы Microchip [www.microchip.com/PICMicro].

USART может использоваться либо как полнодуплексный асинхронный порт с форматом кадра, совместимым с RS232, либо как полуудуплексный синхронный стробируемый порт. Устройство состоит из приемника (receiver), передатчика (transmitter) и генератора тактовой частоты (baud rate generator) (рис. 9.10).

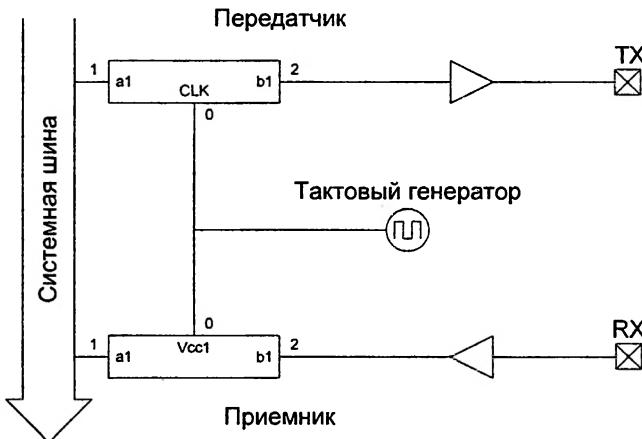


Рис. 9.10. Структура USART микроконтроллера PIC

Генератор тактовой частоты представляет собой двоичный счетчик, уменьшающий на единицу на каждом такте центрального процессора. Когда счетчик доходит до нуля, генерируется один такт и счетчик инициализируется исходным значением. Фактически, генератор представляет собой делитель тактовой частоты процессора в заданном отношении. Документ [www.microchip.com/PICMicro] содержит таблицу, по которой можно рассчитать коэффициент деления, чтобы из типичных тактовых частот процессора получить (иногда с определенной ошибкой) стандартные частоты RS232.

Счетчик генератора тактовой частоты программно недоступен, а начальное значение этого счетчика (оно же коэффициент деления) задается 8-разрядным регистром SPBRG. USART использует одну и ту же тактовую частоту и для приемника, и для передатчика. При синхронной работе в режиме ведомого встроенный генератор тактовой частоты не используется вообще.

Приемник и передатчик во многом аналогичны по структуре и имеют по два программно-доступных 8-разрядных регистра. Первый регистр называется регистром управления и статуса (status and control register), а второй — регистром данных. Значения битов управляющих регистров приемника и передатчика приведены в табл. 9.1 и 9.2. И приемник, и передатчик имеют также программно-недоступный сдвиговый регистр, называемый TSR (Transmit Shift Register) у передатчика и RSR (Receive Shift Register) у приемника.

Таблица 9.1. Описание битов управляющего регистра передатчика USART

Бит	Описание
7	CSRC: Clock SouRCe — выбор источника тактовой частоты при асинхронном режиме игнорируется, при синхронном режиме: <ul style="list-style-type: none"> ▪ 1 = режим ведущего (USART генерирует стробовый сигнал); ▪ 0 = режим ведомого (стробовый сигнал генерирует другое устройство)
6	TX9: передавать 9 бит <ul style="list-style-type: none"> ▪ 1 = передавать 9 бит; ▪ 0 = передавать 8 бит
5	TXEN: Transmission ENabled — разрешить передачу <ul style="list-style-type: none"> ▪ 1 = передача разрешена; ▪ 0 = передача запрещена
4	SYNC: бит выбора режима USART <ul style="list-style-type: none"> ▪ 1 = синхронный режим; ▪ 0 = асинхронный режим
3	Не используется. Читается как 0
2	BRGH: Baud Rate Generator High — переключение режимов работы тактового генератора
1	TRMT: бит состояния сдвигового регистра передатчика <ul style="list-style-type: none"> ▪ 1 = TSR пуст; ▪ 0 = TSR содержит данные (идет передача)
0	TX9D: девятый бит передаваемых данных (может использоваться как бит четности)

Таблица 9.2. Описание битов управляющего регистра приемника USART

Бит	Описание
7	SPEN: Serial Port Enable — включить последовательный порт <ul style="list-style-type: none"> ▪ 1 = последовательный порт включен; ▪ 0 = последовательный порт выключен
6	RX9: разрешение приема 9 бит <ul style="list-style-type: none"> ▪ 1 = выбран прием 9 бит; ▪ 0 = выбран прием 8 бит
5	SREN: Single Receive ENable — бит одиночного приема Асинхронный режим — не используется. Синхронный режим (ведущий): <ul style="list-style-type: none"> ▪ 1 = одиночный прием разрешен; ▪ 0 = одиночный прием запрещен. Этот бит очищается после завершения приема. Синхронный режим (ведомый) — не используется
4	CREN: Continuous Receive ENable — бит непрерывного приема. Асинхронный режим: <ul style="list-style-type: none"> ▪ 1 = прием разрешен; ▪ 0 = прием запрещен. Синхронный режим: <ul style="list-style-type: none"> ▪ 1 = непрерывный прием разрешен; ▪ 0 = непрерывный прием запрещен
3	Не используется. Читается как 0
2	FERR: Framing ERRor — ошибка кадра <ul style="list-style-type: none"> ▪ 1 = ошибка кадра; ▪ 0 = нет ошибки кадра
1	OERR: Overrun ERRor — ошибка переполнения буфера <ul style="list-style-type: none"> 1 = переполнение буфера приема; 0 = нет ошибки переполнения
0	RX9D: девятый бит принятых данных. Может использоваться как бит четности

При передаче значение регистра данных передатчика помещается в сдвиговый регистр и, возможно, расширяется девятым битом, который берется из бита TX9D управляющего регистра. Значение младшего бита выставляется на выходе TX микроконтроллера. Затем на каждом такте генератора регистр сдвигается на один бит — в результате получается последовательная передача битов кадра с заданной скоростью (рис. 9.11). Когда биты кончаются, передатчик устанавливает бит TRMT управляющего регистра и, если это требуется, генерирует прерывание по завершении передачи.

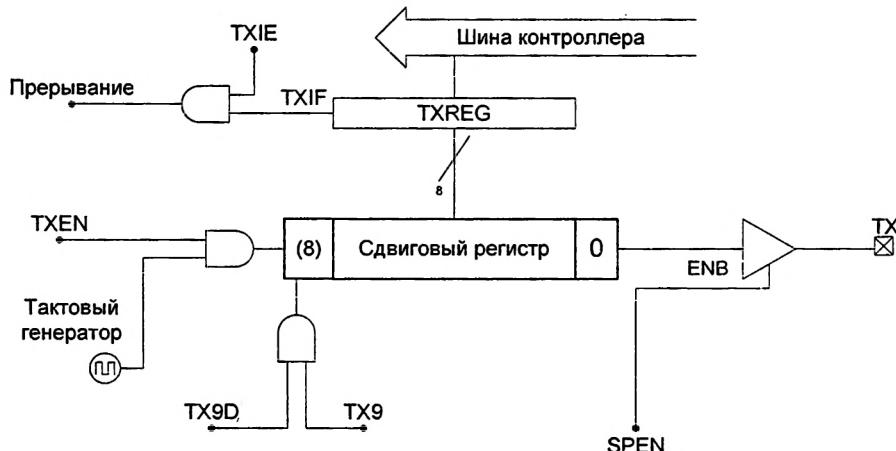


Рис. 9.11. Принципиальная схема передатчика USART

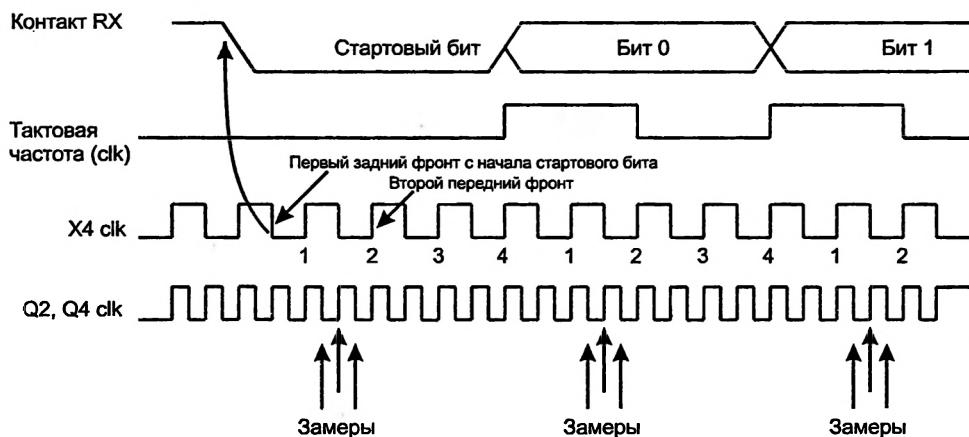


Рис. 9.12. Временная диаграмма работы приемника USART

Приемник имеет несколько более сложное устройство. Значение входа RX анализируется не один раз за каждый такт генератора, а 16 раз, и усредняется — благодаря этому значительно увеличивается помехоустойчивость (рис. 9.12).

Кроме того, приемник USART имеет скрытый буферный регистр, в который помещается значение принятого байта, если регистр данных приемника еще не был прочитан с момента последнего приема (рис. 9.13).

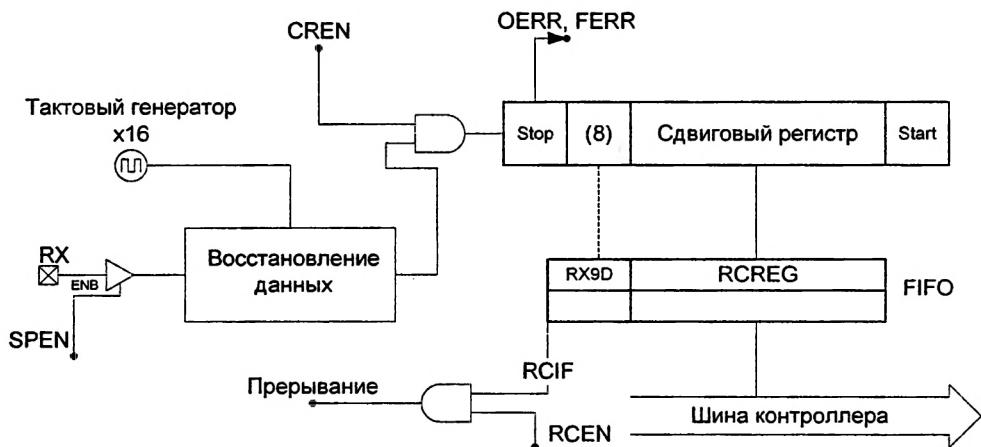


Рис. 9.13. Принципиальная схема приемника USART

Приемники большинства современных аппаратных реализаций RS232 имеют буферные регистры. Так, последовательные порты подавляющего большинства современных IBM PC-совместимых компьютеров основаны на микросхеме National Semiconductor PC16552 [NS PC16552D] и имеют буфера объемом 16 байт (у оригинальной IBM PC были небуферизованные порты).

Помимо освобождения центрального процессора от исполнения работы сдвиговых регистров, аппаратная реализация RS232 позволяет заменить ожидание данных в режиме опроса на работу по прерываниям, что во многих случаях тоже весьма полезно. Многопортовые адаптеры, используемые, например, для организации модемных пуллов интернет-провайдеров, часто могут передавать данные в режиме ПДП.

9.4. Шины

Описанные в предыдущем разделе порты передачи данных соединяют друг с другом два устройства. Однако часто оказывается целесообразно подключить к одному порту передачи данных несколько устройств, причем необязательно однотипных. Каждая передаваемая по такому порту порция данных обязана сопровождаться указанием, какому из подключенных устройств она предназначена — адресом или селектором устройства. Такие многоточечные порты называются *шинами (bus)* (рис. 9.14). Как и двухточечные порты передачи данных, шины бывают синхронные и асинхронные, а также последовательные и параллельные. При описании шин термины "синхронный" и "асинхронный" используют в ином значении, чем при описании портов. Асин-

хронными называют шины, в которых ведомое устройство не выставляет (или не обязано выставлять) сигнал завершения операции, а синхронными, соответственно, шины, где ведомый обязан это делать.

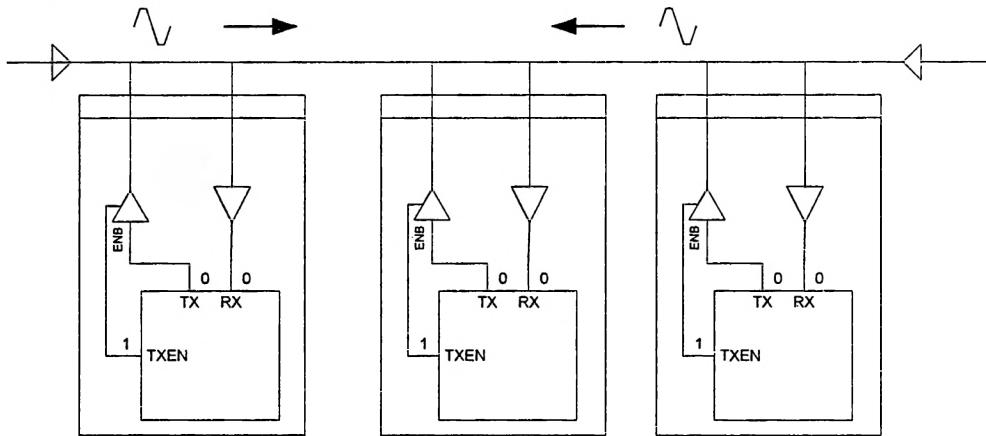


Рис. 9.14. Шина

Подключение N устройств двухточечными портами требует, чтобы центральный процессор имел N приемопередатчиков; использование же многоточечного порта позволяет обойтись одним. В частности, за счет этого удается уменьшить количество выводов микросхемы процессора или периферийного контроллера. Кроме того, при удачном размещении устройств можно получить значительный выигрыш в общей длине проводов и, таким образом, например, уменьшить количество проводников на печатной плате. Во многих случаях это приводит к столь значительному снижению общей стоимости системы, что оказывается целесообразным смириться с усложнением протокола передачи данных и другими недостатками, присущими шинной архитектуре.

Последовательная шина USB

Последовательная шина USB (Universal Serial Bus — универсальная последовательная шина) обеспечивает подключение до 127 устройств при радиусе сети (максимальном расстоянии по кабелю от компьютера до устройства) не более 25 метров. Шина обеспечивает скорости 1,5, 12 и 480 Мбит/с и допускает подключение широкого спектра устройств, начиная от низкоскоростных, таких как мыши, клавиатуры и модемы, и заканчивая дисковыми накопителями и цифровыми видеокамерами. Теоретически она могла бы обеспечивать передачу неупакованного видеопотока с разрешением до 1024×768. На практике, однако, контроллеры USB могут обеспечивать достаточно высокую для этого пиковую скорость передачи, но не могут гарантировать такую скорость, поэтому все камеры, подключаемые по USB, передают упакованный с потерями видеопоток, а

для передачи неупакованных видеоданных применяют интерфейс IEEE 1342 (FireWire).

Топология сети многоуровневая звездообразная; при этом устройства всегда являются оконечными узлами сети. В качестве разветвителей используются активные повторители, так называемые пассивные хабы (натоки сетевых технологий должны отметить, что терминология USB отличается от терминологии, принятой в локальных сетях, где пассивными хабами называют простые резисторные согласователи, а повторители называют активными хабами), либо коммутаторы (активные хабы).

Шина обеспечивает "горячее" — т. е. без выключения компьютера или хаба — включение и выключение устройств. Для обеспечения этого используется ряд приемов, в том числе требования к гальванической развязке портов и высокое внутреннее сопротивление у неактивных приемопередатчиков. Предусмотрены провода для подачи питания; устройство может получать с шины до 500 мА при напряжении 5 В, т. е. до 2,5 Вт. Допускается питание от шины не только оконечных устройств, но и хабов; разумеется, к таким хабам можно подключать только маломощные устройства с потреблением не более 100 мА. Впрочем, этого тока вполне достаточно, чтобы "накормить" мышь или даже клавиатуру. Предусмотрены также механизмы перевода устройств (как пытающихся от шины, так и имеющих собственные источники питания) в режим пониженного энергопотребления.

Микросхемы, реализующие протокол USB (оконечные устройства, хабы и хабы, совмещенные с оконечным устройством), изготавливаются крупными сериями и поэтому достаточно дешевы; многие современные микроконтроллеры имеют встроенные порты USB. Благодаря этому USB быстро вытесняет RS232 и родственные ему протоколы как из вычислительных систем, так и из встраиваемых приложений. USB-устройства практически вытеснили параллельные порты Centronics с позиций основного интерфейса подключения печатающих устройств, и сильно потеснили SCSI и IEEE 1394 при подключении сканнеров и внешних запоминающих устройств. Большинство современных компьютеров общего назначения имеют встроенные контроллеры USB, нередко с большим количеством портов.

Настоящую революцию произвело появление на рынке карманных запоминающих устройств на основе флэш-памяти, которые до сих пор не имеют общепринятого русскоязычного названия, кроме жаргонного "флэшка". По-английски они называются *flash drive*, т. е. флэш-накопитель; по-русски обычно используют кальку "флэш-диск", что, вообще говоря, неправильно, ведь никакого диска в этом устройстве нет. Большинство способов использования этих устройств были бы невозможны без такого интерфейса, как USB — достаточно распространенного, достаточно скоростного и обеспечивающего питание от шины и "горячее" подключение и отключение.

Шина должна иметь выделенный узел, называемый хостом (*host* — англ. хозяин) или корневым хабом (*root hub*), который управляет шиной. Чем именно этот узел отличается от остальных, мы поймем далее в этом разделе; однако выделение его начинается с того, что все разъемы в USB несимметричны: они делятся на разъемы типа А (которыми кабель подключается к корневому хабу или к нисходящему порту хаба) и разъемы типа В (которыми кабель подключается к устройству или восходящему порту хаба). Благодаря этому, используя стан-

дартные кабели, невозможно смонтировать сеть с несколькими корневыми хабами. На практике это, в частности, означает, что невозможно использовать USB для прямого соединения двух компьютеров; впрочем, на рынке доступно много коммуникационных адаптеров USB, начиная от преобразователей USB-RS232 и заканчивая приемопередатчиками WiFi (IEEE 802.11).

Первоначально предполагалось, что корневым хабом всегда будет компьютер. Однако появление многочисленных портативных устройств с достаточно высоким уровнем собственного интеллекта (КПК, цифровых фотоаппаратов и т. д.) создало потребность в более гибкой схеме построения сетей USB — например, пользователь мог бы использовать КПК в качестве ведомого устройства при обмене данными с настольным компьютером и в качестве ведущего — при копировании данных с КПК на флэш-накопитель или цифровой фотоаппарат, или при печати с КПК на принтер. В 2001 году были опубликованы спецификации USB OTG (On The Go — "на ходу"). Кроме ведущих устройств с разъемами типа A и ведомых с разъемами типа B эта спецификация допускает двухролевые устройства со специальными разъемами типа AB, которые допускают подключение двух разных типов кабелей. В зависимости от того, какой кабель подключен, двухролевое устройство работает либо как корневой хаб, либо как ведомое окончное устройство. При соединении двух двухролевых устройств допускается даже динамическая смена ролей, т. е. передача роли корневого хаба от одного устройства к другому.

USB использует четырехжильные кабели на основе экранированной витой пары. По одной паре передается сигнал, по другой — питание. Длина кабеля составляет не более 5 м; если нужен более длинный кабель, отрезки необходимо соединять повторителем (т. е. хабом). Документы стандарта прямо запрещают производство кабелей-“удлинителей”, на обоих концах которых находятся разъемы типа A, потому что, соединяя такие кабели, можно создать сеть, нарушающую спецификации. Однако на рынке такие “удлинители” вполне можно приобрести; они особенно удобны для подключения устройств, не имеющих собственного кабеля, таких, как флэш-накопители, к разъемам USB, находящимся на задней панели компьютера.

Устройства делятся на низкоскоростные (1,5 Мбит/с), полноскоростные (12 Мбит/с) и высокоскоростные (480 Мбит/с). Скорость определяется в момент конфигурации устройства по сопротивлениям между линиями данных и землей и, дополнительно, по запросам конфигурации. Низко- и полноскоростные устройства могут включаться в одну шину; при этом хаб должен определить скорость устройства в момент подключения и, с одной стороны, изолировать низкоскоростное устройство от высокоскоростных сигналов, а с другой стороны — сообщить корневому хабу скорость вновь подключенного порта; для совместного использования полно- и высокоскоростных устройств необходимо использовать коммутаторы (активные хабы), иначе высокоскоростное устройство окажется вынуждено работать на скорости 12 Мбит/с.

Для передачи сигнала используется дифференциальное кодирование сシンхронизацией NRZI (Non Return to Zero Inverted — без возврата к нулю с инверсией): нулевой бит кодируется переходом напряжения от высокого к низкому и обратно, а единица — отсутствием такого перехода. При передаче длинной последовательности единиц приемник может потерять синхронизацию; чтобы этого не происходило, используется битовое заполнение (bit stuffing) — после каждого шести единиц всегда вставляется 0. При передаче двоичных данных приемник ориентируется на разность напряжений между проводами линии данных,

которые обозначаются D+ и D–; "высоким" напряжением (или, что то же самое, "дифференциальной единицей") считается состояние, когда напряжение на линии D+ выше на 200 мВ напряжения на линии D–, "низким" ("дифференциальным нулем") — когда соотношение между напряжениями обратное.

Когда напряжения на проводах равны, это называется "линейным нулем" (SSE, Single Ended Zero). Удержание шины в этом состоянии в течение различных интервалов времени, возможно, в сочетании с дифференциальными нулями и единицами, используется для кодирования дополнительных сигналов — отметки начала и конца кадра, подключения и отключения устройства, сброса и некоторых других.

Каждое устройство на шине может содержать несколько логических устройств, имеющих собственные адреса в диапазоне от 1 до 127. Устройства делятся на хабы и оконечные устройства, так называемые функции (function). Адреса устройств распределяются хостом и назначаются устройствам по мере их подключения.

Каждое логическое устройство может иметь до 16 портов обмена данными с номерами в диапазоне от 0 до 15. Каждый порт характеризуется типом передаваемых данных и направлением передачи (IN — от порта к корневому хабу, OUT — от корневого хаба к порту). Порт 0 используется для конфигурационных запросов.

Определены следующие типы передаваемых данных:

- управляющие передачи (control) — используются хостом для конфигурации устройства и запросов статуса во время работы;
- передачи массивов данных (bulk data) — передача больших объемов данных с гарантией доставки (пакеты, на которые не поступило подтверждения, передаются повторно), но без гарантий времени доставки. Таким образом осуществляется обмен данными с дисками и флэш-накопителями, передача данных на принтеры и т. д.;
- прерывания (interrupt) — сообщения об одиночных событиях, требующих немедленного внимания хоста, например, о движении или нажатии клавиш мыши, нажатии клавиш на клавиатуре и др.;
- изохронные передачи (isochronous) — передача данных реального времени, прежде всего аудио- и видеопотоков. Изохронные передачи используют оговоренную полосу пропускания, т. е. им гарантируется возможность передачи определенного количества байт в течение каждой миллисекунды. Предоставляется несколько механизмов синхронизации часов хоста и устройства, чтобы разрешить возможные недоразумения при определении того, что именно они подразумевают под миллисекундой. Повтора потерянных или поврежденных пакетов не предусмотрено.

Все обмены данными по шине инициируются хостом (корневым хабом). Единичная операция передачи данных называется транзакцией и состоит в обмене тремя пакетами. Транзакции бывают трех типов: SETUP (конфигурация устройства), OUT (передача данных от хоста устройству) и IN (прием хостом данных от устройства). Передача данных между устройствами без их буферизации хостом невозможна.

Первый пакет называется маркером (token), он содержит адрес устройства и порта и указывает тип транзакции. Он всегда передается хостом.

Второй пакет содержит данные. При транзакциях SETUP и OUT он передается хостом, при транзакции IN — устройством. Размер пакета данных различен для разных типов передачи и составляет от 8 до 1023 байт. Пакеты данных защищены CRC16.

Третий пакет — подтверждение приема данных, называется рукопожатием (handshake). При транзакциях SETUP и OUT он передается устройством, при транзакции IN — хостом. При изохронной передаче пакет подтверждения не посыпается. При остальных типах передачи пакет рукопожатия содержит информацию о результате исполнения операции. Определены три кода результатов: ACK (acknowledgement, подтверждение), NAK (Negative Acknowledgement — отрицательное подтверждение) и STALL (сбой). ACK обозначает нормальный прием данных или завершение конфигурационного запроса. NAK обозначает, что устройство занято и не способно выполнить запрос, но, скорее всего, сможет выполнить тот же самый запрос через некоторое время. STALL обозначает ошибку конечной точки, которая требует вмешательства хоста (а, скорее всего, драйвера этого устройства на хосте).

В USB 2.0 была добавлена вырожденная транзакция PING, используемая для проверки хостом готовности устройства к работе. Кроме того, был добавлен код результата NYET (Not YET — нет еще), обозначающий, что устройство выполнило эту транзакцию, но, по-видимому, не сможет выполнить следующую. Обычно это происходит при заполнении внутренних буферов устройства.

Из этого описания можно понять, что соблюдение временных интервалов и гарантий полосы пропускания при изохронной передаче должно обеспечиваться хостом. Сами по себе протоколы USB не содержат никаких механизмов, с помощью которых оконечные устройства или хабы могли бы защититься от недобросовестного хоста.

Можно понять также, что "прерывания" USB реализуются посредством циклического опроса устройств, которые могли бы породить прерывание. Частота этого опроса соответствует "задержке прерывания"; желаемая частота указывается устройством при конфигурации, а соблюдение этой частоты полностью лежит на совести хоста.

Детальное обсуждение принципов работы шины USB, в частности, механизмов синхронизации часов при изохронных передачах или совместной работы разноскоростных устройств можно найти в описании стандарта [www.usb.org USB2] или в посвященной этому вопросу литературе, например в книгах [Агуров 2006, Гук 2005].

Заслуживает обсуждения вопрос о том, как происходит обнаружение новых устройств нашине, особенно когда шина содержит несколько хабов. Благодаря звездообразной топологии сети и конструкции разъемов, новые устройства могут подключаться только к нисходящим портам хабов. Каждый хаб является устройством USB, имеет адрес (в этом смысле нельзя не отметить, что обещание возможности подключения к USB 127 устройств несколько завышено — ведь в числе этих устройств необходимо считать и хабы) и список атрибутов, в числе которых количество портов хаба и состояние каждого порта. Для каждого порта хаб помнит, подключено ли к нему вообще что-то, и если подключено, то с какой скоростью (низкой, полной или высокой). Хост-контроллер периодически опрашивает все свои хабы и выясняет, не изменились ли эти списки.

При подключении нового устройства, хаб обнаруживает это по изменению сопротивления между линиями D+, D- и землей; по этому же признаку он опреде-

ляет скорость устройства. Выждав определенное время, в течение которого устройство должно было инициализировать свой USB-интерфейс, хаб обновляет свои состояния портов. Когда хост в очередной раз запросит у хаба обновления, хаб сообщит о новом устройстве. Дальнейшее взаимодействие с устройством осуществляется хостом. Хост должен назначить устройству адрес (после включения в шину устройство получает адрес 0) и получить его атрибуты, в том числе идентификатор, по которому следует искать подходящий драйвер для устройства.

Рассмотрим устройство корневого хаба (хост-контроллера) UHCI (Universal Host Controller Interface). Спецификации UHCI были разработаны компанией Intel и описывают архитектуру двухпортового хост-контроллера USB 1.0 и интерфейс между хост-контроллером и компьютером. Контроллеры с таким интерфейсом обычно выполняются на одной микросхеме с контроллером системной и периферийной шин, так называемым чипсетом (chipset); UHCI-совместимые контроллеры выпускаются не только Intel, но и рядом других производителей чипсетов, причем не только для PC-совместимых компьютеров. Решение опубликовать и стандартизовать хост-контроллер, разумеется, сильно упростило жизнь поставщикам операционных систем и значительно облегчило продвижение USB на рынок. Действительно, поскольку контроллеры разных изготовителей полностью совместимы на уровне управляющих регистров и команд, для поддержки всех таких контроллеров достаточно одного драйвера. Большинство современных ОС в той или иной форме поддерживает UHCI-совместимые контроллеры.

Кроме интерфейса UHCI от Intel в свое время был опубликован также интерфейс OHCI (Open Host Controller Interface — открытый интерфейс хост-контроллера), разработанный компаниями VIA и IBM, но этот интерфейс имел несколько меньший успех на рынке, так что даже новые чипсеты VIA снабжены UHCI-совместимым контроллером.

После выхода спецификаций USB 2.0 были разработаны спецификации хост-контроллера EHCI (Enhanced Host Controller Interface). EHCI описывает контроллер, который может иметь до восьми портов USB 2.0 и объединяет до четырех логических контроллеров UHCI. При этом действует механизм передачи порта (port handoff): если к порту подключено устройство или хаб USB 1.0, то порт передается соответствующему логическому контроллеру UHCI. Также, если ОС не поддерживает EHCI, она может инициализировать все четыре логических контроллера как UHCI.

Контроллер UHCI представляет собой устройство PCI, имеющее восемь регистров различных размеров (табл. 9.3). Большинство регистров занимает два байта адресного пространства ввода/вывода; кроме того, есть один четырехбайтовый регистр и один однобайтовый. Расположение этих регистров в адресном пространстве ввода/вывода контролируется стандартными средствами PCI (периферийная шина PCI обсуждается далее в этом разделе) с помощью настройки базового адреса; благодаря этому компьютер может содержать несколько хост-контроллеров.

Значения битов регистров приводятся в документе [www.intel.com UHCI]. Детальное описание всех битов потребовало бы детального же описания процедуры инициализации контроллера и представляется мне недостаточно поучительным для изложения в этой книге.

Таблица 9.3. Порты контроллера UHCI, цит. по [www.intel.com UHCI]

Адрес порта	Назначение	Доступ
Base + (00–01h)	USBCMD USB Command	R/W
Base + (02–03h)	USBSTS USB Status	R/WC
Base + (04–05h)	USBINTR USB Interrupt Enable	R/W
Base + (06–07h)	FRNUM Frame Number	R/W
Base + (08–0Bh)	FRBASEADD Frame List Base Address	R/W
Base + 0Ch	SOFMOD Start Of Frame Modify	R/W
Base + (10–11h)	PORTRSC1 Port 1 Status/Control	R/WC
Base + (12–13h)	PORTRSC2 Port 2 Status/Control	R/WC

R — регистр доступен для чтения.

W — регистр доступен для записи.

WC — запись с очисткой (при записи 1 соответствующий бит очищается, запись 0 игнорируется).

Кроме этих регистров, контроллер использует для общения с программным обеспечением структуры данных в ОЗУ, которые он считывает и модифицирует в режиме прямого доступа к памяти. Указатель на эти структуры хранится в четырехбайтовом регистре FRBASEADDR. Структура называется списком кадров (frame list); она всегда выровнена на двойное слово, поэтому младшие четыре бита регистра используются для других целей.

Список кадров представляет собой массив из 1024 элементов. Регистр FRBASEADDR указывает на его начало; младшие десять бит регистра FRNUM используются в качестве индекса. Каждый кадр соответствует интервалу времени в 1 мс и содержит список транзакций, которые необходимо совершить в течение этого интервала. В соответствии со спецификациями USB, список должен начинаться с изохронных транзакций, затем должны идти транзакции опроса источников прерываний и только затем могут следовать запросы на передачи массивов данных. Соблюдение этого требования, впрочем, целиком возлагается на драйвер хост-контроллера.

Каждый запрос на транзакцию описывается структурой TD (Transfer Descriptor). Эта структура описывает тип транзакции, адрес устройства, с которым ее надлежит осуществить, и ряд других параметров, например, следует ли генерировать прерывание по завершению этой транзакции, и если да, при каких именно условиях. Кроме того, она содержит указатель на буфер, в котором лежат предназначенные к передаче данные (для транзакций OUT и SETUP) или в который следует поместить принятые данные (для транзакций IN). Содержится также размер этого буфера. После завершения транзакции контроллер помещает в ее TD код результата и, если этого требуют параметры, может генерировать прерывание.

Список транзакций имеет сложную структуру и кроме TD может содержать записи типа QH (Queue Head — заголовок очереди). Эти записи используются,

главным образом, для описания транзакций блочной передачи массивов данных, когда массивы слишком велики и не могут быть переданы одной транзакцией. Из-за того, что для транзакций этого типа предусмотрен повтор неудачных передач, невозможно предсказать полное время, в течение которого будет происходить передача всего блока и, в частности, невозможно гарантировать, что данная последовательность транзакций будет исполнена в течение одного кадра. Поэтому рекомендуемая структура очереди выглядит так, как изображено на рис. 9.15.

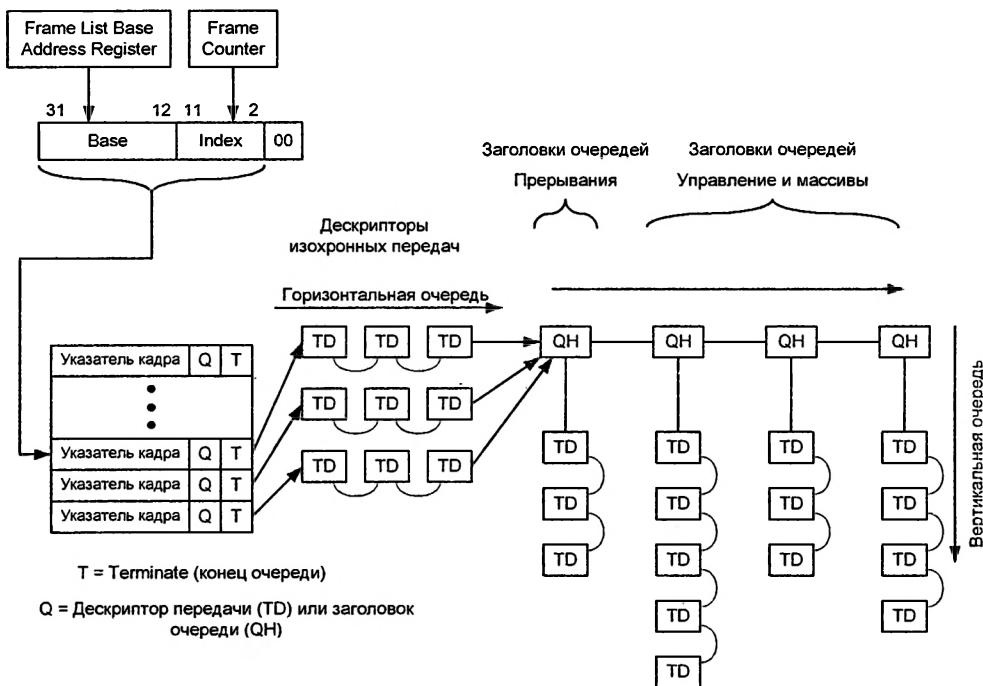


Рис. 9.15. Типичная структура расписания транзакций для контроллера UHCI

Успешно завершив блочную транзакцию, контроллер передвигает указатель в заголовке очереди. Если транзакция завершилась ответом NAK или какой-то другой ошибкой (например, не сошлась контрольная сумма у принятого пакета или пакет рукопожатия не был получен в течение заданного времени), контроллер не будет передвигать указатель и сделает повторную попытку в следующем кадре. TD содержит двухбитный счетчик, который указывает количество допустимых повторных попыток. Этот счетчик уменьшается при каждой ошибке, и когда он доходит до нуля, контроллер генерирует прерывание, оповещая драйвер, что с устройством, похоже, что-то не в порядке.

Таким образом, при своей работе контроллер осуществляет прямой доступ к памяти. Однако, в отличие от простых контроллеров ПДП, подобных описанному в разд. 6.2, контроллер UHCI может работать со списками буферов, расположенных в несмежных областях ОЗУ. Такой режим ПДП называется разбра-

сыванием-сборкой (scatter-gather) и поддерживается многими современными дисковыми контроллерами, сетевыми адаптерами и адаптерами периферийных шин, таких как SCSI.

Основной недостаток шин состоит в том, что в каждый момент времени только одно устройство нашине может передавать данные. Если у двухточечных портов часто оказывается целесообразным реализовать полнодуплексный обмен данными посредством двух комплектов линий (один на прием, другой на передачу, как в описанном ранее RS232), то в случае шинной топологии это невозможно. Поэтому шины бывают только полудуплексные или симплексные.

Невозможность параллельно осуществлять обмен с двумя устройствами может привести к падению производительности по сравнению с собственным портом обмена данными у каждого устройства. Если устройства не занимают пропускную способность каналов передачи полностью, проигрыш оказывается не так уж велик. Благодаря этому шины широко используются даже в ситуациях, когда только одно устройство имеет возможность инициировать обмены данными.

Если передачу данных могут инициировать несколько устройств (как, например, в случае системной шины с несколькими процессорами и/или контроллерами ПДП), шинная топология оказывается наиболее естественным выбором. Такая конфигурация требует решения еще одной проблемы: обеспечения арбитража доступа кшине со стороны возможных инициаторов обмена — задатчиков шины. Методы арбитража отличаются большим разнообразием.

В системных и периферийных шинах вычислительных систем применяются две основные стратегии.

- Центральный арбитр. Каждый потенциальный задатчик шины имеет собственную линию запроса на передачу и собственную же линию разрешения передачи (рис. 9.16). Арбитр может реализовать различные стратегии выбора между конфликтующими задатчиками — приоритетные, "справедливые" и т. д.
- Разного рода "выборы" (election). Все задатчики шины выставляют нашину свои запросы; затем задатчик с наивысшим приоритетом (способ определения приоритета зависит от способа выставления запроса) считается выигравшим арбитраж и начинает передачу.

Арбитраж нашине SCSI

Периферийная шина SCSI (подробнее рассматривается далее в этом разделе) использует для передачи данных параллельнуюшину с восемью битами данных (более современный Wide SCSI — с шестнадцатью) и несколькими линиями служебных сигналов. Кшине может подключаться до восьми (наWide

SCSI — до 16) устройств с адресами в диапазоне от 0 до 8 (соответственно, от 0 до 15 на Wide SCSI). Адрес называется SCSI ID, устанавливается на устройстве перемычками перед подключением к шине. Инициатором передачи может быть любое из устройств.

Желая начать передачу, устройство выставляет на шине сигнал BSY (занято) и устанавливает бит данных, соответствующий своему SCSI ID. Поэтому максимальное число устройств на шине соответствует разрядности шины данных. Устройство с наибольшим ID выигрывает арбитраж; остальные устройства подтверждают это, снимая свои запросы.

Спецификации SCSI, предназначенные для работы по последовательным портам (IEEE 1394 и FC/AL), используют маркерный доступ. IEEE 1394 представляет собой маркерную шину с центральным устройством, которое периодически рассыпает маркер всем устройствам, передавая им право захватить шину на некоторое время (USB тоже представляет собой маркерную шину). FC/AL — маркерное кольцо, в котором все устройства объединены кабелями в кольцо и последовательно передают маркер друг другу. Передача маркера редко используется в системных шинах, этот прием более характерен для сетевых протоколов [Иртегов 2004].

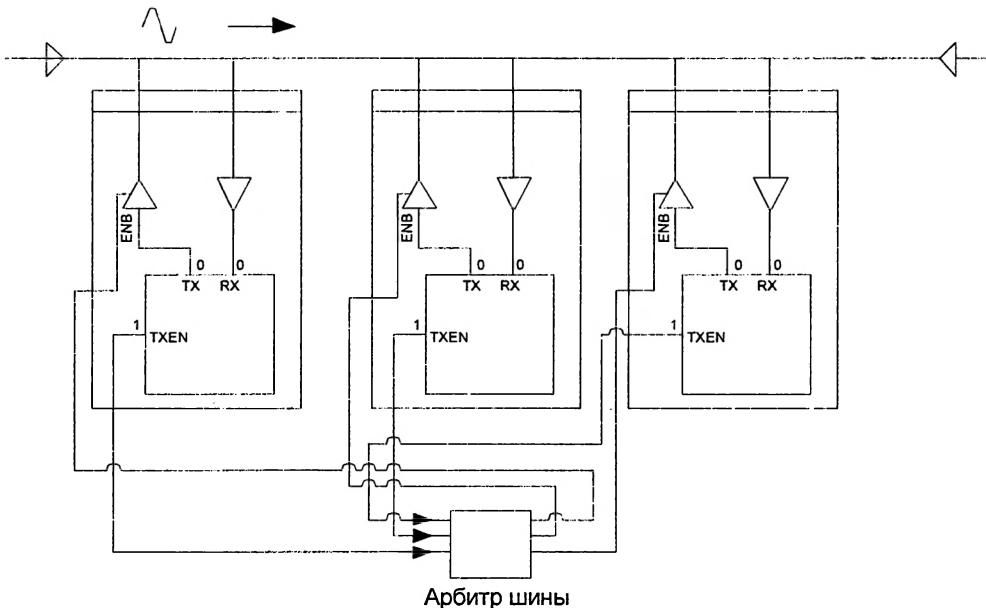


Рис. 9.16. Шина с несколькими задатчиками

Необходимость обнаружения коллизий и арбитража накладывает ограничения на физический размер шины: сигналы распространяются по шине с конечной скоростью, а любые два конфликтующих устройства должны узнать о коллизии за время, меньшее чем минимальный цикл передачи данных (цикл

шины). Кроме того, большое количество коллизий само по себе снижает производительность системы.

Один из основных путей решения этой проблемы упоминался в разд. 6.5: замена шины центральным коммутатором либо более или менее сложной сетью коммутаторов. Устройства соединяются с ближайшим коммутатором полнодуплексными двухточечными каналами, такие же каналы используются для соединения коммутаторов друг с другом, а все коллизии возникают и разрешаются только внутри коммутаторов.

Внутренняя шина коммутатора, как правило, имеет большую пропускную способность, чем внешние соединения — у многих практически используемых коммутаторов несколько внутренних шин, поэтому разрешение коллизии внутри коммутатора часто состоит в отправке данных другим путем (рис. 9.17).

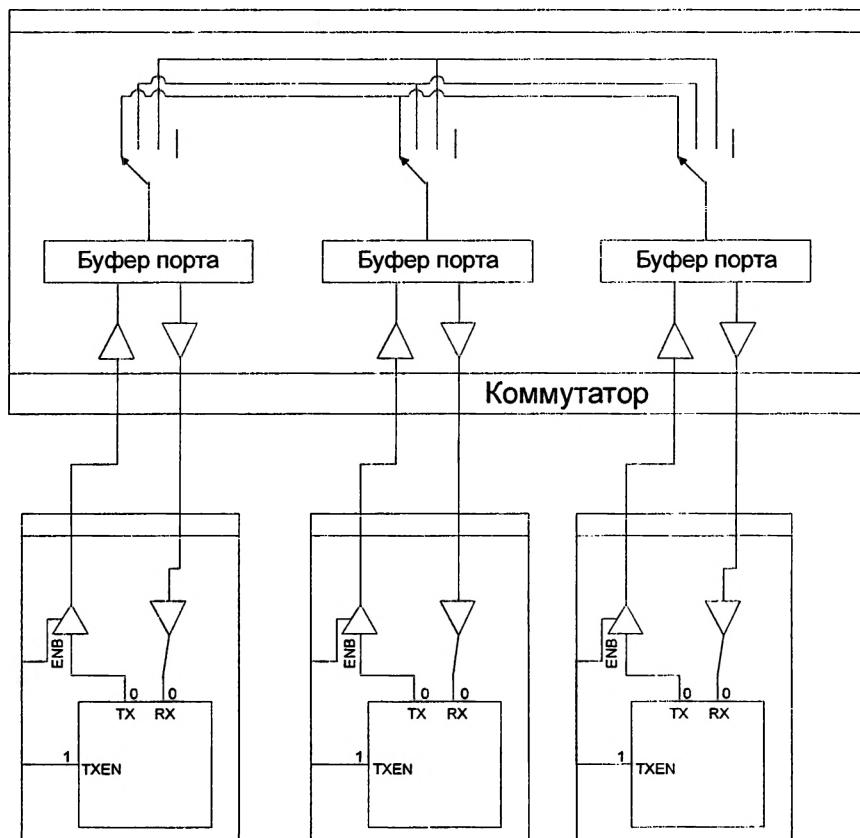


Рис. 9.17. Коммутатор с несколькими внутренними шинами

Таким образом, коммутируемая магистраль позволяет уменьшить потери производительности, порождаемые коллизиями, и повысить реальную пропускную способность магистрали. Однако коммутируемая магистраль значительно повышает общую стоимость системы и далеко не всегда оправдана с этой точки зрения.

Применяются также гибридные топологии, когда все или некоторые устройства подключаются к коммутатору полудуплексным портом или коротким участком шины — при этом возможны локальные коллизии, но их гораздо меньше, чем при единойшине. Гибридная топология позволяет использовать устройства с одними и теми же приемопередатчиками как в обычной шинной, так и в коммутируемой топологии (рис. 9.18). В частности, именно такую гибридную топологию имеют описанные в разд. 6.5 многопроцессорные системы IBM NUMA-Q и SGI Origin — процессоры и память подключаются к локальной шине процессорного модуля, а сами модули соединяются коммутируемой сетью.

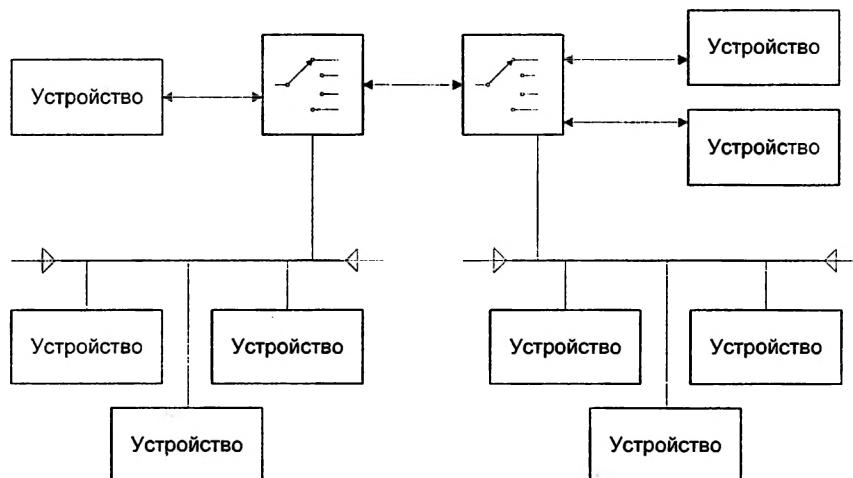


Рис. 9.18. Гибридная топология

Гибридная топология также позволяет снизить стоимость системы по сравнению с полностью коммутируемой сетью — коммутаторы можно устанавливать только в тех участках шины, где расстояния или большое количество коллизий становятся проблемой. В таких сетях часто используются двухпортовые коммутаторы, называемые мостами (bridge). Так же называются и адаптеры, используемые для соединения двух шин с разными протоколами.

Во всех перечисленных случаях сохраняется логическая шинная топология: каждый потенциальный задатчик шины может обращаться по адресу к любо-

му другому устройству, поэтому разработчикам программного обеспечения не всегда интересны детали реализации физического подключения устройств. Шины (как физические, так и логические) находят широчайшее применение в вычислительной технике. В частности, один из важнейших сетевых протоколов канального уровня, Ethernet, начинал свою карьеру как последовательная шина с совмещением синхросигнала и данных, да и современные реализации этого протокола, хотя физически и представляют собой коммутируемую или, все реже и реже, гибридную сеть, с логической точки зрения по-прежнему являются шиной.

Связь центральных процессоров с банками ОЗУ и внешними устройствами в подавляющем большинстве современных компьютеров осуществляется параллельными стробируемыми шинами.

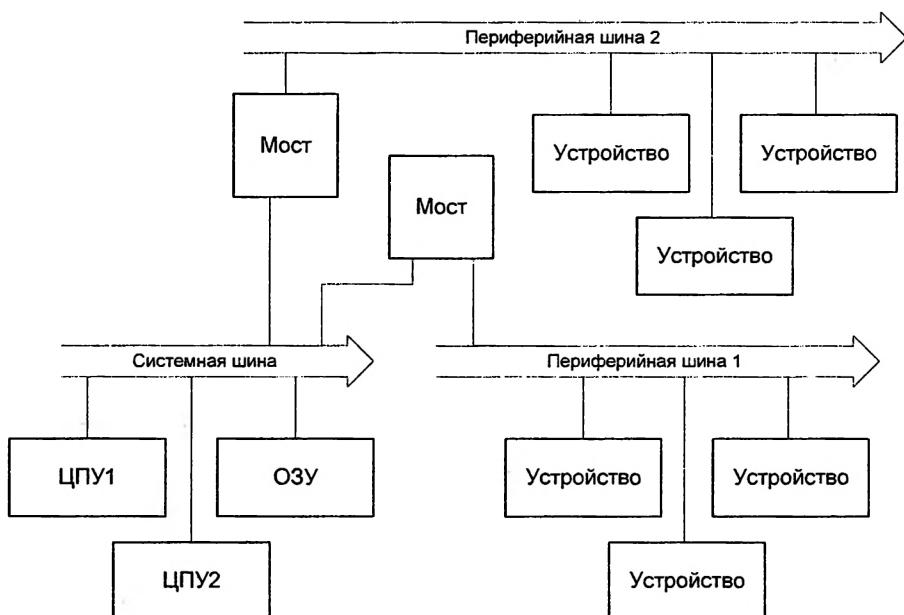


Рис. 9.19. Системная и периферийные шины

В вычислительных системах общего назначения чаще всего используется топология с двумя или более шинами. Процессоры и память связаны *системной шиной*. К этойшине также присоединены один или несколько адаптеров *периферийных шин*, к которым и подключаются внешние устройства (рис. 9.19). Основное преимущество такого решения состоит в том, что одни и те же периферийные устройства могут использоваться в вычислительных системах с различными системными шинами. Высокоскоростные устройства, например

дисковые адаптеры *FC-AL* у серверов или графические контроллеры у персональных компьютеров, иногда подсоединяются к системной шине непосредственно или, точнее, через собственный адаптер периферийной шины.

Шина PCI

Шина *PCI* (Peripheral Component Interconnect), разработанная фирмой Intel, в настоящее время используется в вычислительных системах разного уровня, начиная от персональных компьютеров с процессорами x86 и PowerMac и заканчивая старшими моделями серверов Sun Fire. В конструктиве PCI исполняется множество различных периферийных устройств — сетевые карты, адаптеры *SCSI*, графические и звуковые карты, платы видеоввода и видеовывода и др. [PC Guide PCI, Гук 2005]. С начала 90-х годов XX века до настоящего времени было опубликовано несколько версий спецификаций шины. Первая версия спецификаций, изданная в 1992 году, предполагала 32-разрядную совмещенную шину адреса и данных, работающую с тактовой частотой до 33 МГц. Она обеспечивала пиковую пропускную способность до 132 Мбайт/сек.

В более поздних версиях спецификаций шины адреса и данных были расширены до 64 бит; при этом 64-битные устройства могли включаться в 32-битные разъемы и работали как 32-битные, так что новые устройства можно было подключать к старым материнским платам. В 1999 году были опубликованы спецификации PCI-X, допускающие тактовые частоты до 133 МГц. В PCI-X 2.0 допускаются новые режимы синхронизации, обеспечивающие удвоенную и учетверенную по отношению к тактовой частоте шины частоту передачи данных, что обеспечивало пиковую пропускную способность до 4256 Мбайт/сек.

Шина PCI используется в нескольких существенно различающихся конструкциях. Наиболее известны конструктивы PCI и CardBus. Конструктив PCI с ножевыми разъемами используется в платах расширения настольных компьютеров и серверов. Некоторые серверные материнские платы поддерживают горячее подключение устройств, но далеко не все.

Конструктив CardBus используется в портативных компьютерах. По габаритам и разъему платы расширения CardBus похожи на платы PCMCIA, но отличаются от них выступами на разъеме ("ключом"). В CardBus поддержка горячего подключения является обязательной, так что пользователь компьютера может подключать и отключать устройства по своему желанию (разумеется, при условии, что ОС компьютера поддерживает добавление устройств на ходу).

Менее известные варианты конструктива — это miniPCI (также используется в портативных компьютерах, но разъемы и посадочные гнезда обычно скрыты внутри корпуса) и cPCI/PXI (используется в промышленных компьютерах).

К вариантам шины PCI нередко также относят конструктив AGP (Accelerated Graphic Port). AGP использует тактовую частоту 66 МГц, допускает подключение только одного устройства, конвееризацию обращений к памяти, некоторые дополнительные сигналы и увеличенную (двух-, четырех- и даже восьмикратную) частоту передачи данных. Пиковая пропускная способность порта составляет 2132 Мбайт/сек, что уступает спецификациям PCI-X533. Порт специально оптимизирован для работы с устройствами, имеющими собственные буферы ОЗУ большого объема, т. е. с графическими адаптерами.

Новая версия PCI — PCI Express, ранее известная как 3GIO (3 Generation Input/Output — ввод/вывод третьего поколения), — представляет собой не ши-

ну, а коммутируемую сеть древовидной топологии. Соединения между коммутаторами и между коммутатором и устройством представляют собой двухточечные полнодуплексные последовательные порты с пропускной способностью 2,5 Гбит/сек. Допускается агрегирование портов в группы из 2, 4, 8, 12, 16 и 32 линий, что обеспечивает пиковую скорость около 8 Гбайт/сек. Допускаются как электрические, так и оптоволоконные соединения. Подключение устройств PCI и PCI-X к шине PCI Express осуществляется через мосты.

Поскольку PCI Express обеспечивает более высокие скорости передачи, чем AGP, этот интерфейс используется главным образом для подключения видеокарт, и довольно быстро вытесняет AGP.

PCI не рассчитана на использование в качестве системной шины. Спецификации шины предполагают ее подключение к процессорам и памяти через специальное устройство, мост системной шины (host-to-PCI bridge) (рис. 9.20). Это позволяет использовать PCI в системах, основанных на различных процессорах, таких как x86, Power, SPARC.

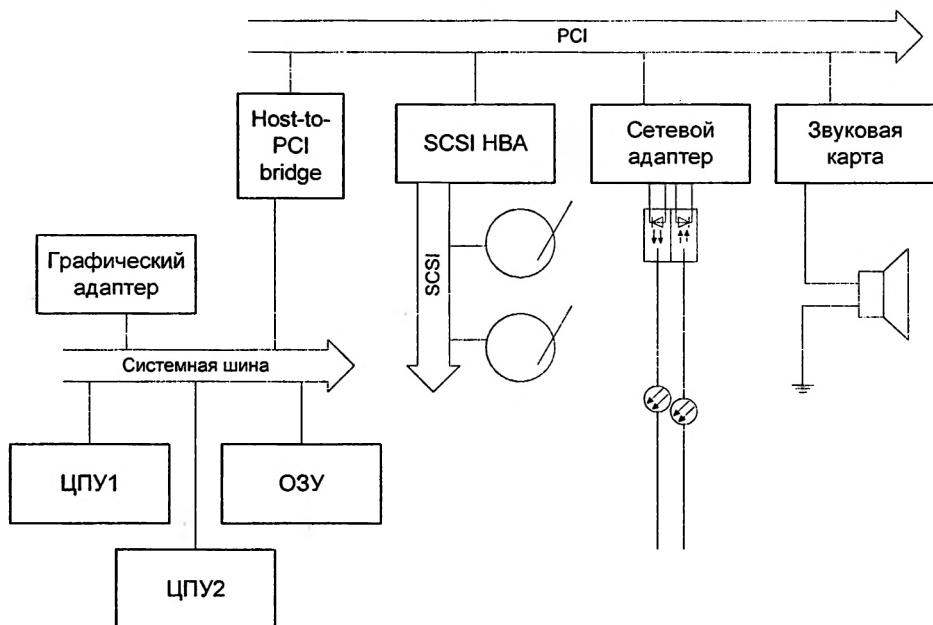


Рис. 9.20. Мост системной шины

PCI предусматривает как монтаж устройств на материнскую плату системы, так и их исполнение в виде плат расширения. Первые версии спецификаций шины допускали подключение не более четырех устройств, при этом только три из них могло подключаться через внешние разъемы (четвертое должно было монтироваться на материнской плате). Современные спецификации допускают до шести устройств нашине. Предусмотренный стандартом способ обхода указанного ограничения состоит в установке в системе нескольких адаптеров PCI, через собственные мосты системной шины или через PCI-to-PCI bridge. Допус-

тимы также мосты, подключающие другие периферийные шины, например ISA, USB и т. д. (рис. 9.21).

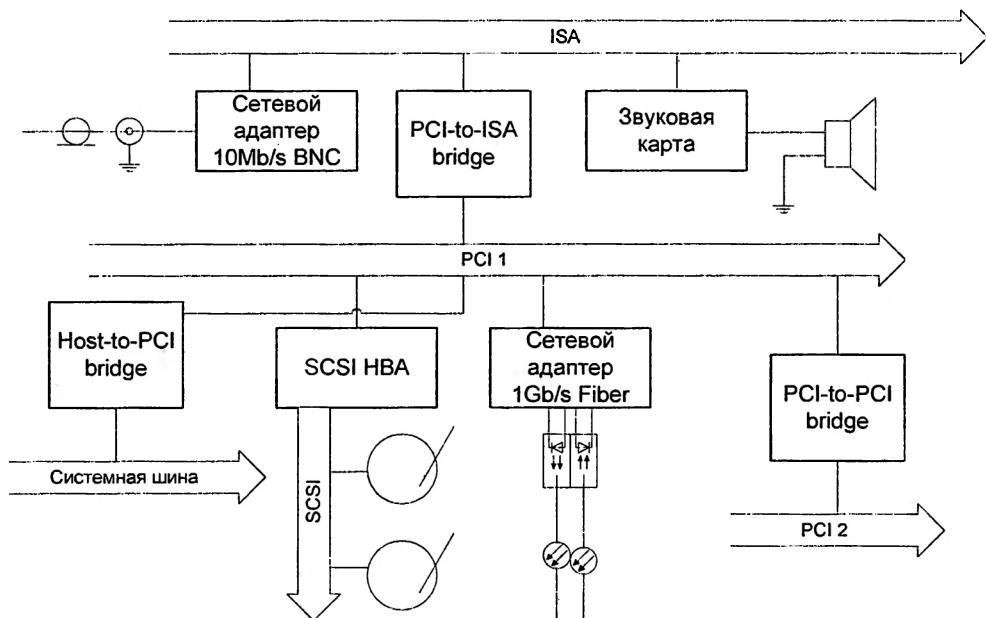


Рис. 9.21. Мости PCI-to-PCI и PCI-to-ISA

Количество разъемов PCI Express практически не ограничено спецификациями, но современные материнские платы для настольных компьютеров обычно имеют только один такой разъем, реже — два.

Таким образом, PCI может состоять из нескольких шин, соединенных между собой мостами. Мости представляют собой устройства PCI и должны объединяться в древовидную сеть. Корень этого дерева подключается к системнойшине и называется главным мостом (host bridge).

Порт AGP обычно подключается к системнойшине непосредственно, а не через мост PCI, но конфигурация устройства производится теми же средствами, что и у устройств PCI.

После включения питания или системного сброса устройства PCI (в том числе и все мости, кроме главного) реагируют только на конфигурационные запросы. Каждому устройству выделено 256 байт конфигурационного адресного пространства (у PCI-X эта область расширена до 4096 байт). Адрес конфигурационного регистра представляет собой 32-разрядное слово и состоит из нескольких битовых полей. Младшие восемь бит адреса — AD[7:0] — идентифицируют собственно конфигурационный регистр. Биты AD[10:8] адресуют функцию (function) — логическую подсистему устройства. Каждое устройство должно иметь функцию 0, в описании которой может содержаться указание на наличие других функций. Биты AD[15:11] представляют собой недекодированный номер устройства нашине. Для устройств, смонтированных на платах расшире-

ния, этот адрес порождается на основе номера разъема, в который оно вставлено. Таким образом, для конфигурационных регистров используется географическая адресация.

Биты AD[23:16] кодируют номер шины. В PCI-X биты AD[27:24] используются в качестве старших битов адреса регистра, благодаря чему количество регистров каждого устройства и увеличено в шестнадцать раз. Биты AD[31:28] зарезервированы.

Блок конфигурационных регистров функции начинается с заголовка, который содержит следующие поля:

1. Vendor ID — 16-битный идентификатор изготовителя устройства. Идентификатор назначается PCI SIG. Vendor ID компании Intel равен 0x8086.
2. Device ID — 16-битный идентификатор устройства, назначаемый производителем.
3. Revision ID — 16-битный идентификатор версии микросхемы, также назначаемый производителем.
4. Header Type — 8-битный описатель типа заголовка. Наиболее известны типы 0 (простые устройства), 1 (мосты PCI-PCI) и 2 (мосты PCI-CardBus).
5. Class Code — 24-битный код типа устройства. Классификация устройств иерархическая и довольно обширная. Эта классификация постоянно расширяется; текущая версия таблицы кодов может быть найдена на сайте [\[pcisig.com\]](http://pcisig.com). Старший байт кода обозначает класс устройства, например видеoadаптер, контроллеры устройства внешней памяти, сетевой контроллер и т. д. Второй байт обозначает подкласс, например, для сетевых устройств это могут быть контроллеры SCSI, ATA или гибких магнитных дисков. Наконец, младший байт кодирует стандарт, которому соответствует интерфейс устройства. Так, контроллер USB UHCI обозначается кодом 0C:03:00. Таким образом, операционная система может выбирать подходящий драйвер для устройства либо на основании пары Vendor ID/Device ID, либо на основании кода класса.
6. Base Address Register (BAR) — описатель области памяти. Функция может иметь до шести таких областей. Младший бит регистра BAR определяет, относится ли эта область к адресному пространству ОЗУ или портов ввода/вывода. Биты [2:1] кодируют тип адресации — 00 соответствует 32-битному адресу, 10 — 64-битному. Коды 01 и 11 зарезервированы, в старых версиях спецификаций код 01 обозначал память в пределах первого мегабайта, доступную из реального режима x86. Бит 3 обозначает, что эта область соответствует настоящей памяти, допускающей предварительную выборку. Старшие биты регистра могут модифицироваться. Благодаря этому BIOS или операционная система могут разместить блоки портов и банки памяти так, чтобы их адреса не конфликтовали с другими устройствами и основным ОЗУ.
7. Capability Pointer — указатель на цепочку регистров свойств функции. Свойства, которые могут описываться таким образом, отличаются большим разнообразием и включают в себя управление энергопотреблением, специальные возможности AGP и др. Некоторые устройства используют свойства для описания блоков управляющих регистров, для которых не хватило стандартных шести BAR.

8. Expansion ROM Base Address — адрес микросхемы ПЗУ, которая содержит модуль расширения BIOS, позволяющий BIOS материнской платы и вторичному загрузчику ОС (а иногда и самой ОС) управлять этим устройством. Такие микросхемы обычно устанавливаются на адAPTERы SCSI (обеспечивают доступ к дискам SCSI через сервисы BIOS Int 13 и их современные расширения), на сетевые адAPTERы (модуль, обеспечивающий загрузку по сети) и видеoadAPTERы (так называемые video BIOS).

Данное описание не является исчерпывающим, в него не включен ряд регистров, описание функций которых не существенно для понимания процесса автоконфигурации PCI. Итак, после сброса устройство реагирует только на циклы доступа к конфигурационным регистрам. BIOS материнской платы или загрузчик ОС сканирует конфигурационное адресное пространство, находит устройства и назначает им области адресов памяти, ввода/вывода и ПЗУ, а также другие ресурсы (например, линии запроса прерывания) в соответствии с их потребностями (см. пример 9.2, рис. 9.23).

Доступ к конфигурационным регистрам осуществляется с помощью главного моста PCI. В компьютерах на основе процессоров x86 используются мосты двух типов (оба эти типа описываются в спецификациях PCI). Мосты первого типа имеют два 32-разрядных регистра — CONFIG_ADDRESS (0x0CF8) и CONFIG_DATA (0x0CFA). Для обращения к конфигурационному регистру устройства процессор записывает в регистр CONFIG_ADDRESS адрес конфигурационного регистра. После этого мост отображает соответствующий регистр устройства на свой регистр CONFIG_DATA. Считывание или модификация значения CONFIG_DATA приводит к соответствующим операциям над регистром устройства. Записывая в регистр CONFIG_ADDRESS специальные значения, можно инициировать специальные циклы шины, которые вкратце описываются далее в этом разделе.

Мосты второго типа отображают конфигурационное адресное пространство шины на адреса 0xC000—0xFFFF. При этом регистры 0x0CF8 и 0x0CFA используются как своего рода селекторы банков. Меняя значения этих регистров, можно обращаться к различным шинам и к функциям отдельных устройств. Такие мосты сложнее программировать, чем мосты первого типа. Кроме того, диапазон адресов размером в 4096 байт недостаточен для устройств PCI-X, поэтому на современных материнских платах такие мосты не применяются.

Пример 9.2. Фрагмент выдачи утилиты сканирования шины на компьютере автора

Craig Hart's PCI+AGP bus sniffer, version 0.48vKA, freeware made in 1996-2003.

```
PCI BIOS Version 2.10 found!
Number of PCI Busses : 3
PCI Characteristics : Config Mechanism 1
```

Searching for PCI Devices using the OEMHLP\$ driver

```
Vendor 8086h Intel Corporation
Device 2570h 82865G\PE\P Processor to I/O Controller
```

Command 0106h (Memory Access, BusMaster, System Errors)
Status 2090h (Has Capabilities List, Supports Back-To-Back Trans., Received Master Abort, Fast Timing)
Revision 02h, Header Type 00h, Bus Latency 00h
Self test 00h (Self test not supported)
PCI Class Bridge, type PCI to HOST
Subsystem ID 25708086h Unknown (Generic ID)
Subsystem Vendor 8086h Intel Corporation
Address 0 is a Memory Address (anywhere in 0-4Gb, Prefetchable) : F8000000h
New Capabilities List Present:
 Vendor-Dependant Capability
 AGP Capability, Version 3.0 (AGP 8x and below, core register support)
 AGP Speed(s) Supported : 4x 8x
 FW Transfers Supported : Yes
 >4Gb Address Space Supported : No
 Sideband Addressing Supported : Yes
 Maximum Command Queue Length : 32 bytes
 AGP Speed Selected : 2x
 FW Transfers Enabled : No
 >4Gb Address Space Enabled : No
 AGP Enabled : Yes
 Sideband Addressing Enabled : Yes
 Current Command Queue Length : 1 byte

 Vendor 8086h Intel Corporation
 Device 2571h 82865G\PE\P Processor to AGP Controller
 Command 0107h (I/O Access, Memory Access, BusMaster, System Errors)
 Status 00A0h (Supports 66MHz, Supports Back-To-Back Trans., Fast Timing)
 Revision 02h, Header Type 01h, Bus Latency 20h
 Self test 00h (Self test not supported)
 PCI Class Bridge, type PCI to PCI
 PCI Bridge Information:
 Primary Bus Number 0, Secondary Bus Number 1, Subordinate bus number 1
 Secondary Bus Command 0008h (VGA mapping)
 Secondary Bus Status 22A0h (Supports 66MHz, Supports Back-To-Back Trans., Received Master Abort, Medium Timing)
 Secondary Bus Latency 20h
 I/O Port Range Passed to Secondary Bus : A000h to AFFFh (16-bit)
 Memory Range Passed to Secondary Bus : FF800000h to FF8FFFFFh
 Prefetchable Memory Range Passed to Secondary Bus : D6B00000h to F6AFFFFFh

Vendor 8086h Intel Corporation
Device 24D2h 82801EB/ER (ICH5/ICH5R) USB UHCI Controller #1
Command 0005h (I/O Access, BusMaster)
Status 0280h (Supports Back-To-Back Trans., Medium Timing)
Revision 02h, Header Type 80h, Bus Latency 00h
Self test 00h (Self test not supported)
PCI Class Serial, type USB (UHCI)
Subsystem ID 524C8086h Unknown
Subsystem Vendor 8086h Intel Corporation
Address 4 is an I/O Port : 0000CC00h
System IRQ 11, INT# A

Vendor 8086h Intel Corporation
Device 24D4h 82801EB/ER (ICH5/ICH5R) USB UHCI Controller #2
Command 0005h (I/O Access, BusMaster)
Status 0280h (Supports Back-To-Back Trans., Medium Timing)
Revision 02h, Header Type 00h, Bus Latency 00h
Self test 00h (Self test not supported)
PCI Class Serial, type USB (UHCI)
Subsystem ID 524C8086h Unknown
Subsystem Vendor 8086h Intel Corporation
Address 4 is an I/O Port : 0000D000h
System IRQ 5, INT# B

Vendor 8086h Intel Corporation
Device 24D7h 82801EB/ER (ICH5/ICH5R) USB UHCI Controller #3
Command 0005h (I/O Access, BusMaster)
Status 0280h (Supports Back-To-Back Trans., Medium Timing)
Revision 02h, Header Type 00h, Bus Latency 00h
Self test 00h (Self test not supported)
PCI Class Serial, type USB (UHCI)
Subsystem ID 524C8086h Unknown
Subsystem Vendor 8086h Intel Corporation
Address 4 is an I/O Port : 0000D400h
System IRQ 10, INT# C

Vendor 8086h Intel Corporation
Device 24DEh 82801EB/ER (ICH5/ICH5R) USB UHCI Controller #4
Command 0005h (I/O Access, BusMaster)
Status 0280h (Supports Back-To-Back Trans., Medium Timing)

Revision 02h, Header Type 00h, Bus Latency 00h
Self test 00h (Self test not supported)
PCI Class Serial, type USB (UHCI)
Subsystem ID 524C8086h Unknown
Subsystem Vendor 8086h Intel Corporation
Address 4 is an I/O Port : 0000D800h
System IRQ 11, INT# A

Vendor 8086h Intel Corporation
Device 24DDh 82801EB/ER (ICH4/ICH5R) USB EHCI Controller
Command 0106h (Memory Access, BusMaster, System Errors)
Status 0290h (Has Capabilities List, Supports Back-To-Back Trans., Medium Timing)
Revision 02h, Header Type 00h, Bus Latency 00h
Self test 00h (Self test not supported)
PCI Class Serial, type USB (EHCI)
Subsystem ID 524C8086h Unknown
Subsystem Vendor 8086h Intel Corporation
Address 0 is a Memory Address (anywhere in 0-4Gb) : FFAFFC00h
System IRQ 9, INT# D
New Capabilities List Present:
Power Management Capability
Supports power state D1
Current Power State : D0 (Device operational, no power saving)
Debug Port Capability

Vendor 8086h Intel Corporation
Device 244Eh 82801EB/ER Hub Interface to PCI Bridge (ICH5/ICH5R A2/A3 step)
Command 0107h (I/O Access, Memory Access, BusMaster, System Errors)
Status 0080h (Supports Back-To-Back Trans., Fast Timing)
Revision C2h, Header Type 01h, Bus Latency 00h
Self test 00h (Self test not supported)
PCI Class Bridge, type PCI to PCI
PCI Bridge Information:
Primary Bus Number 0, Secondary Bus Number 2, Subordinate bus number 2
Secondary Bus Command 0202h ()
Secondary Bus Status 2280h (Supports Back-To-Back Trans., Received Master Abort, Medium Timing)
Secondary Bus Latency 20h
I/O Port Range Passed to Secondary Bus : B000h to BFFFh (16-bit)
Memory Range Passed to Secondary Bus : FF900000h to FF9FFFFFh

Vendor 8086h Intel Corporation
Device 24D0h 82801EB/ER (ICH5/ICH5R) LPC Interface Bridge
Command 010Fh (I/O Access, Memory Access, BusMaster, Special Cycles, System Errors)
Status 0280h (Supports Back-To-Back Trans., Medium Timing)
Revision 02h, Header Type 80h, Bus Latency 00h
Self test 00h (Self test not supported)
PCI Class Bridge, type PCI to ISA

Vendor 8086h Intel Corporation
Device 24DBh 82801EB/ER (ICH5/ICH5R) EIDE Controller
Command 0007h (I/O Access, Memory Access, BusMaster)
Status 0288h (Signalled Interrupt, Supports Back-To-Back Trans., Medium Timing)

Revision 02h, Header Type 00h, Bus Latency 00h

Self test 00h (Self test not supported)

PCI Class Storage, type IDE

PCI EIDE Controller Features :

BusMaster EIDE is supported

Primary Channel is at I/O Port 01F0h and IRQ 14

Secondary Channel is at I/O Port 0170h and IRQ 15

Subsystem ID 524C8086h Unknown

Subsystem Vendor 8086h Intel Corporation

Address 0 is an I/O Port : 00000000h

Address 1 is an I/O Port : 00000000h

Address 2 is an I/O Port : 00000000h

Address 3 is an I/O Port : 00000000h

Address 4 is an I/O Port : 0000FFA0h

Vendor 8086h Intel Corporation
Device 24D1h 82801EB (ICH5) SATA Controller
Command 0005h (I/O Access, BusMaster)
Status 02A0h (Supports 66MHz, Supports Back-To-Back Trans., Medium Timing)

Revision 02h, Header Type 00h, Bus Latency 00h

Self test 00h (Self test not supported)

PCI Class Storage, type IDE

PCI EIDE Controller Features :

BusMaster EIDE is supported

Primary Channel is in native mode at Addresses 0 & 1

Secondary Channel is in native mode at Addresses 2 & 3

Subsystem ID 524C8086h Unknown
Subsystem Vendor 8086h Intel Corporation
Address 0 is an I/O Port : 0000EC00h
Address 1 is an I/O Port : 0000E800h
Address 2 is an I/O Port : 0000E400h
Address 3 is an I/O Port : 0000E000h
Address 4 is an I/O Port : 0000DC00h
System IRQ 10, INT# A

Vendor 8086h Intel Corporation
Device 24D3h 82801EB/ER (ICH5/ICH5R) SMBus Controller
Command 0001h (I/O Access)
Status 0280h (Supports Back-To-Back Trans., Medium Timing)
Revision 02h, Header Type 00h, Bus Latency 00h
Self test 00h (Self test not supported)
PCI Class Serial, type SMBus Controller
Subsystem ID 524C8086h Unknown
Subsystem Vendor 8086h Intel Corporation
Address 4 is an I/O Port : 0000C800h
System IRQ 3, INT# B

Vendor 1002h ATI Technologies
Device 5961h Unknown
Command 0107h (I/O Access, Memory Access, BusMaster, System Errors)
Status 02B0h (Has Capabilities List, Supports 66MHz, Supports
Back-To-Back Trans., Medium Timing)
Revision 01h, Header Type 80h, Bus Latency 20h
Self test 00h (Self test not supported)
Cache line size 64 Bytes (16 DWords)
PCI Class Display, type VGA
Subsystem ID 7C13174Bh Unknown
Subsystem Vendor 174Bh PC Partner Ltd
Address 0 is a Memory Address (anywhere in 0-4Gb, Prefetchable) :
E8000000h
Address 1 is an I/O Port : 0000A800h
Address 2 is a Memory Address (anywhere in 0-4Gb) : FF8F0000h
System IRQ 11, INT# A
Expansion ROM of 128Kb decoded by this card
New Capabilities List Present:
AGP Capability, Version 3.0 (AGP 8x and below, core register support)
AGP Speed(s) Supported : 4x 8x

FW Transfers Supported : Yes
>4Gb Address Space Supported : No
Sideband Addressing Supported : Yes
Maximum Command Queue Length : 256 bytes
AGP Speed Selected : 2x
FW Transfers Enabled : No
>4Gb Address Space Enabled : No
AGP Enabled : Yes
Sideband Addressing Enabled : Yes
Current Command Queue Length : 1 byte
Power Management Capability
Supports power state D1
Current Power State : D0 (Device operational, no power saving)

Vendor 1002h ATI Technologies
Device 5941h Unknown
Command 0007h (I/O Access, Memory Access, BusMaster)
Status 02B0h (Has Capabilities List, Supports 66MHz, Supports Back-To-Back Trans., Medium Timing)
Revision 01h, Header Type 00h, Bus Latency 20h
Self test 00h (Self test not supported)
Cache line size 64 Bytes (16 DWords)
PCI Class Display, type Other
Subsystem ID 7C12174Bh Unknown
Subsystem Vendor 174Bh PC Partner Ltd
Address 0 is a Memory Address (anywhere in 0-4Gb, Prefetchable) : E0000000h
Address 1 is a Memory Address (anywhere in 0-4Gb) : FF8E0000h
New Capabilities List Present:
Power Management Capability
Supports power state D1
Current Power State : D0 (Device operational, no power saving)

Vendor 1516h Myson Technology Inc
Device 0803h 100/10M PCI Fast Ethernet Adapter
Command 0007h (I/O Access, Memory Access, BusMaster)
Status 0290h (Has Capabilities List, Supports Back-To-Back Trans., Medium Timing)
Revision 00h, Header Type 00h, Bus Latency 20h
Self test 00h (Self test not supported)

Cache line size 64 Bytes (16 DWords)
PCI Class Network, type Ethernet
Subsystem ID 132010BDh EP-320X-S Fast Ethernet Adapter
Subsystem Vendor 10BDh Surecom Technology
Address 0 is an I/O Port : 0000B800h
Address 1 is a Memory Address (anywhere in 0-4Gb) : FF9FFC00h
System IRQ 7, INT# A
Expansion ROM of 64Kb decoded by this card

New Capabilities List Present:

Power Management Capability
Supports power state D1
Current Power State : D0 (Device operational, no power saving)

Vendor 1274h Creative (Was: Ensoniq)
Device 1371h ES1371, ES1373 AudioPCI
Command 0105h (I/O Access, BusMaster, System Errors)
Status 3410h (Has Capabilities List, Received Target Abort, Received Master Abort, Slow Timing)
Revision 08h, Header Type 00h, Bus Latency 20h
Self test 00h (Self test not supported)
PCI Class Multimedia, type Audio
Subsystem ID 13711274h Creative Sound Blaster AudioPCI64V, AudioPCI 128 (Generic ID)

Subsystem Vendor 1274h Creative (Was: Ensoniq)
Address 0 is an I/O Port : 0000BC00h
System IRQ 10, INT# A

New Capabilities List Present:

Power Management Capability
Current Power State : D0 (Device operational, no power saving)

Vendor 11C1h Lucent/Agere Systems (Was: AT&T MicroElectronics)
Device 5811h FW322/323 IEEE1394 OHCI FireWire Controller
Command 0116h (Memory Access, BusMaster, MemWrite+Invalidate, System Errors)
Status 0290h (Has Capabilities List, Supports Back-To-Back Trans., Medium Timing)

Revision 61h, Header Type 00h, Bus Latency 20h
Self test 00h (Self test not supported)
PCI Class Serial, type OHCI FireWire
Subsystem ID 524C8086h Unknown
Subsystem Vendor 8086h Intel Corporation

Address 0 is a Memory Address (anywhere in 0-4Gb) : FF9FE000h
System IRQ 3, INT# A
New Capabilities List Present:
Power Management Capability
Supports power state D1
Current Power State : D0 (Device operational, no power saving)

Vendor 8086h Intel Corporation
Device 1050h PRO/100 VE Network Connection
Command 0117h (I/O Access, Memory Access, BusMaster,
MemWrite+Invalidate, System Errors)
Status 0290h (Has Capabilities List, Supports Back-To-Back Trans.,
Medium Timing)
Revision 01h, Header Type 00h, Bus Latency 20h
Self test 00h (Self test not supported)
Cache line size 64 Bytes (16 DWords)
PCI Class Network, type Ethernet
Subsystem ID 30208086h Unknown
Subsystem Vendor 8086h Intel Corporation
Address 0 is a Memory Address (anywhere in 0-4Gb) : FF9FD000h
Address 1 is an I/O Port : 0000B400h
System IRQ 9, INT# A
New Capabilities List Present:
Power Management Capability
Supports power state D1
Current Power State : D0 (Device operational, no power saving)

ROM PCI IRQ routing table Windows 9x Compatibility Tests....
ROM IRQ routing table found at F000h:3D20h
Table Version 1.0 - OK
Table size 224 bytes - OK
Table Checksum D0h - OK
IRQ's dedicated to PCI : None
The ROM PCI IRQ routing table appears to be OK.

IRQ Summary: IRQs 3,5,7,9,10,11,14,15 are used by PCI devices
Shared IRQs: IRQ 3 is shared by 2 PCI Devices
 IRQ 9 is shared by 2 PCI Devices
 IRQ 10 is shared by 3 PCI Devices
 IRQ 11 is shared by 3 PCI Devices

Таблица 9.4. Основные сигналы шины PCI

Сигнал	Назначение
AD[31:0]	Address/Data — совмещенная шина адреса/данных
C/BE[3:0]	Command/Byte Enable — команда/разрешение обращения к байтам
FRAME#	Кадр. Мастер шины удерживает этот сигнал на протяжении всей транзакции. Его снятие сигнализирует, что следующая порция данных будет последней в транзакции
DEVSEL#	Device Select — устройство выбрано
IRDY#	Initiator Ready — готовность ведущего устройства
TRDY#	Target Ready — готовность ведомого устройства
STOP#	Запрос на прекращение транзакции по инициативе ведомого устройства
LOCK#	Сигнал блокировки шины для целостного выполнения операции, состоящей из нескольких транзакций PCI
REQ#	Request — запрос ведущего устройства на захват шины.
GNT#	Grant — ответ арбитра шины
RST#	Reset — сброс шины
IDSEL	Initialization Device Select — выбор устройства в конфигурационных циклах
INT [A-D] #	Interrupt — линии запроса прерывания
CLK	Clock — тактовая частота шины

Наиболее важные из сигналов 32-разрядной шины PCI приведены в табл. 9.4. Большинство сигналов кодируется уровнем на время нарастающего фронта сигнала CLK; именно эти моменты и считаются тактами шины. Сигналы, обозначение которых завершается символом #, считаются установленными при низком напряжении на соответствующей линии и снятыми — при высоком напряжении.

В таблице не перечислен ряд сигналов, среди которых следует упомянуть линии, обеспечивающие обнаружение устройств и автоматическую конфигурацию шины. Так, сигнал M66EN должен быть заземлен у устройств, которые поддерживают тактовую частоту 33 МГц, и свободен — у устройств, способных работать на 66 МГц. Контроллер шины выбирает тактовую частоту, поддерживаемую всеми устройствами, — т. е. если он обнаруживает, что линия M66EN заземлена, то выбирает 33 МГц, а если свободна — то 66 МГц.

Обмен данными по шине осуществляется с помощью пакетов или, как это называется в спецификациях PCI, транзакций. В большинстве транзакций участвуют два устройства — ведущее (master) и ведомое (slave). Ведущее устройство также называют инициатором (initiator), а ведомое — целевым (target). Пре-

дусмотрены также специальные широковещательные транзакции. В таких транзакциях участвует один инициатор, а все остальные устройства шины выступают в качестве ведомых.

Транзакция начинается с того, что ведущее устройство запрашивает доступ к шине сигналом **REQ#**. Арбитр шины отвечает на этот запрос сигналом **GNT#**. Получив этот сигнал, устройство дожидается тишины на шине. Если до этого сигнал **GNT#** будет снят (как правило, это происходит потому, что поступил запрос от более приоритетного устройства), инициатор не имеет права начинать транзакцию. Более того, если сигнал **GNT#** будет снят во время транзакции, инициатор обязан прекратить ее.

Затем происходит выбор целевого устройства. Он начинается с того, что инициатор выставляет низкий уровень сигнала **FRAME#** и устанавливает на линиях **AD[31:0]** адрес целевого устройства, а на линиях **C/BE** код команды, который определяет тип транзакции. После одного такта **CLK**, инициатор освобождает линии адреса и выставляет сигнал **IRDY#**. За этим следует один или несколько пассивных тактов (так называемый *turnaround*), во время которых целевое устройство должно отреагировать и выставить на низкий уровень сигналы **TRDY#** и **DEVSEL#**. Затем происходит собственно обмен данными (рис. 9.22).

В зависимости от типа транзакции, передачу данных может осуществлять как ведущее, так и целевое устройство. Обмен данными может включать в себя одну или несколько фаз, в течение каждой из которых передаются 32 бита данных. Транзакции, в которых используется более одной фазы данных, называются пакетными (*packet*). Данные передаются по шине **AD[31:0]** при условии одновременного наличия сигналов **IRDY#** и **TRDY#**. Снимая свой сигнал, оба устройства (как ведущее, так и ведомое) могут сообщать партнеру о неготовности принимать данные с такой скоростью.

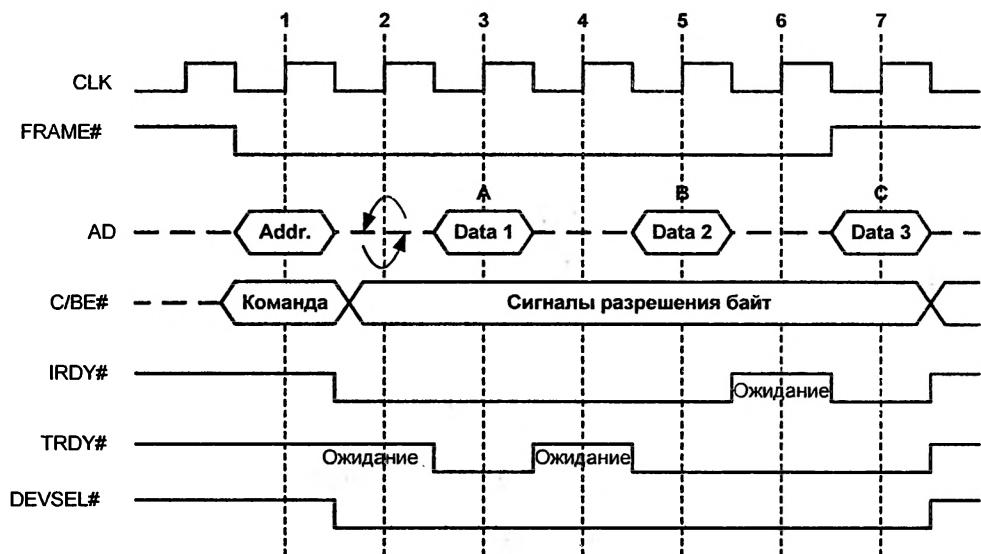


Рис. 9.22. Временная диаграмма сигналов шины PCI

Количество данных, передаваемых в рамках транзакции, определяется неявно, за счет того, что ведущее устройство снимает сигнал FRAME# после передачи предпоследнего пакета данных. После этого передается еще один пакет данных и транзакция завершается. Инициатор должен снять сигнал IRDY#, а ведомое устройство — TRDY# и DEVSEL#. Легко понять, что в транзакциях с одним пакетом данных сигнал FRAME# должен сниматься сразу после передачи адреса.

Допускаются быстрые смежные транзакции (fast back-to-back), при которых инициатор на одном такте снимает IRDY# и вновь устанавливает DEVSEL# — разумеется, при условии, что арбитр шины позволяет ему это.

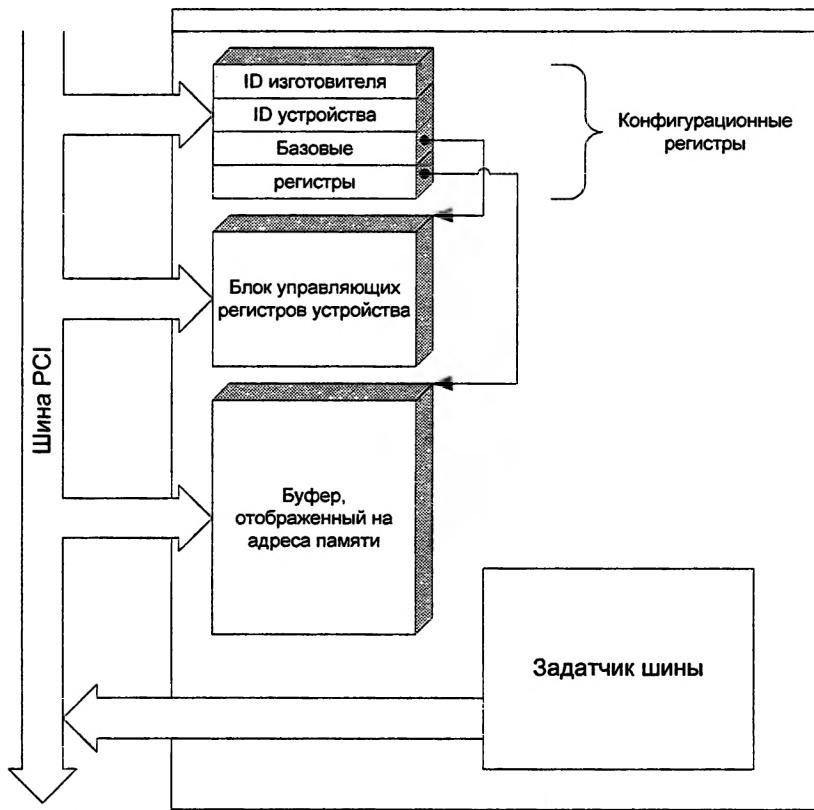


Рис. 9.23. Конфигурационные и рабочие регистры устройства PCI

Набор команд (типов транзакций) шины PCI включает следующие операции:

- I/O Read, I/O Write — команды чтения и записи портов ввода/вывода;
- Memory Read, Memory Write — команды чтения одиночных регистров, отображенных на память;
- Memory Read Line — чтение строки [кэша]. Такие транзакции используются для чтения оперативной памяти блоками, размер которых соответствует

размеру строки кэша ЦПУ. Поскольку у разных моделей процессоров размер строки кэша различается, устройства обязаны знать этот размер. Он задается при инициализации системы записью в специальный конфигурационный регистр, который должно иметь каждое устройство на шине;

- **Multiple Memory Read** — множественное чтение памяти, используется для транзакций, объем которых превосходит емкость строки кэш-памяти;
- **Memory Write and Invalidate (MVI)** — запись с инвалидацией строки кэша;
- **Configuration Read/Write** — команды чтения и записи конфигурационных регистров устройств;
- **Special Cycle** — широковещательные циклы, адресованные всем устройствам на шине. Инициатором таких циклов обычно является мост. Тип сообщения передается по линиям AD[15:0] в фазе, которая соответствует фазе адреса у остальных типов транзакций. Линии AD[31:16] при этом могут использоваться для передачи данных. В настоящее время определены три типа сообщений — *Shutdown* (отключение, код 0x0000), *Halt* (остановка, 0x0001) и трассировка (код 0x0003);
- **Interrupt Acknowledge** — подтверждение прерывания.

Из этого описания можно понять, что максимальная скорость работы шины достигима только в пакетных транзакциях, т. е. при обращениях к памяти — как к основной памяти системы, так и к отраженным на память буферам устройств. К сожалению, процессоры x86 не умеют генерировать пакетные транзакции, большие, чем одна линия кэша (64 бита у процессоров *Pentium*). Некоторые мосты PCI могут объединять последовательности операций записи в длинные пакетные транзакции, но для операций чтения это не получается. Поэтому близкие к теоретическому максимуму скорости передачи данных могут обеспечивать только периферийные устройства, обменивающиеся данными с ОЗУ в режиме ПДП.

В PCI-X логика управления сигналами шины в целом не изменена, но в каждой транзакции после фазы адреса введена дополнительная фаза, в течение которой инициатор сообщает свой идентификатор, состоящий из номера шины, номера устройства и номера функции на устройстве, а также 5-битный тег и 12-битный счетчик байтов. Идентификатор и тег совместно образуют серийный номер транзакции (*sequence*).

В этой версии спецификаций введено также понятие *расщепленной транзакции* (*Split transaction*). Если целевое устройство не способно завершить транзакцию немедленно, оно подает сигнал *Split* и освобождает шину. Когда устройство завершает операцию, оно инициирует транзакцию типа *Split Response* и передает данные или сообщение о завершении (*Split Complete Message*). В транзакции *Split Response* вместо адреса целевого устройства передается серийный номер исходной транзакции. Таким образом, каждый задатчик шины может одновременно вести до 32 незавершенных расщепленных транзакций.

Расщепленные транзакции повышают эффективность использования шины в системах с большим количеством устройств, работающих с разными скоростями.

В PCI-X введена также команда *DIM* (*Device ID Message*), которая позволяет отправлять сообщения устройствам, используя вместо адресов портов их идентификаторы, состоящие из номера шины, устройства и порта устройства.

Шина SCSI

Среди других практически важных применений шин необходимо упомянуть стандарт SCSI (Small Computer System Interface — системный интерфейс малых компьютеров), наиболее известный как интерфейс для подключения жестких дисков, но используемый для подключения широкого спектра высокоскоростных устройств (табл. 9.5), в том числе и не являющихся устройствами памяти, например сканеров [Гук 2000].

Современные спецификации SCSI по структуре аналогичны спецификациям сетевых протоколов и состоят из трех уровней: команд, протокола и соединений. Уровень команд определяет формат и семантику команд и ответов на них, т. е. приблизительно соответствует тому, что в сетевых протоколах называется прикладным уровнем. Уровень протокола определяет способ передачи команд и ответов и соответствует канальному уровню сетевых протоколов.

И, наконец, уровень соединений определяет физическую реализацию линий передачи данных (способ кодирования данных, допустимые токи и напряжения, конструкцию разъемов и т. д.), т. е. соответствует физическому уровню [www.t10.org architecture, Friedhelm/Shmidt 1997]. Сетевой (в данном случае описывающий способ адресации устройств) и транспортный уровни в спецификациях SCSI также присутствуют, хотя и вrudиментарном виде.

Таблица 9.5. Типы устройств SCSI, возвращаемые командой INQUIRY, цит. по [www.t10.org commands]

Код	Описание
00h	Устройство прямого доступа (например, магнитный диск)
01h	Устройство последовательного доступа (например, магнитная лента)
02h	Печатающее устройство
03h	Процессорное устройство
04h	Устройство однократной записи (например, некоторые оптические диски)
05h	CD-ROM
06h	Сканер
07h	Оптическое запоминающее устройство (например, некоторые оптические диски)
08h	Устройство с автоматической заменой носителя (например, jukebox)
09h	Коммуникационное устройство
0Ah—0Bh	Определено ASC IT8
0Ch	Контроллер массива запоминающих устройств (например, RAID)
0Dh	Устройство enclosure services (мост к шине другого типа)
0Eh—1Eh	Зарезервировано
1Fh	Неизвестный или неопределенный тип устройства

На физическом уровне SCSI представляет собой параллельную шину с 8 или 16 (так называемая Wide SCSI) линиями передачи данных. Современные реализации стандарта обеспечивают скорости передачи данных до 160 Мбит/с. Стандартом SCSI III предусмотрены также реализации в виде последовательной шины IEEE 1394 или волоконно-оптических колец FC-AL (Fiber Channel Arbitrated Loop — волоконно-оптический канал [в виде] арбитрируемого кольца). Fiber Channel допускает объединение нескольких колец коммутаторами, подключение до 127 устройств, общую длину сети, измеряемую несколькими сотнями метров, и скорости передачи до 800 Мбит/с.

Недавно появилась спецификация SAS (Serially Attached SCSI — последовательно присоединяемый SCSI), которая предусматривает звездообразную топологию подключения устройств. Каждое устройство подключается к HBA собственным кабелем, обеспечивающим последовательную передачу данных; это упрощает синхронизацию HBA и устройства и обеспечение работы различных устройств с разными скоростями. Тонкие последовательные кабели точка-точка гораздо проще разводить внутри корпуса компьютера, чем плоские кабели с множественными разъемами. Кроме того, поскольку кабели SAS имеют всего восемь проводов, на плате контроллера можно разместить гораздо больше разъемов, чем это было возможно при использовании параллельных кабелей. Переход к последовательным кабелям также упрощает разводку плат HBA и дисковых контроллеров — теперь разработчикам не надо добиваться совпадения длин разных проводников, чтобы избежать битового перекоса.

Обмен данными по шине происходит не байтами и даже не словами, а кадрами, которые могут содержать команды, информацию о состоянии устройства (*sense data*) или собственно данные. Формат кадров и способ разрешения коллизий зависит от используемого типа соединений — в этом смысле разделение уровня соединений и уровня протокола не реализовано в полной мере. Каждый кадр содержит адреса отправителя и получателя. Каждое устройство имеет собственный адрес, называемый SCSI target ID. У старых версий протокола этот адрес был 3-битным, допуская адресацию восьми устройств, у современной версии (Ultra-SCSI или SCSI III) адрес 4- или 5-битный. Адаптер, через который шина SCSI подключается к системной или периферийной шине компьютера — HBA (Host Bus Adapter — адаптер системной шины) — также считается устройством и обычно имеет ID, равный, в зависимости от разрядности адреса, 7, 15 или 127.

Устройство, имеющее один target ID, может содержать несколько логических устройств, идентифицируемых номерами логических устройств (Logical Unit Number, LUN). Спецификации протокола предусматривают два байта для кодирования LUN, но реальное число логических устройств, поддерживаемых на практике, обычно невелико.

Устройства SCSI делятся на два типа: *инициаторы (initiator)* и *целевые устройства (target)*. Инициатор посылает команды, а целевое устройство их исполняет. Спецификация допускает, что в разных актах обмена данными инициатор может становиться целевым устройством и наоборот, но в пределах одной операции эти роли заданы однозначно. Как правило, HBA может выступать только в роли инициатора, а периферийные устройства — только в роли целевых.

Блоки команд — CDB (Command Descriptor Block — блок описания команды) — невелики по размеру и могут содержать 6, 8, 10, 12 или 16 байт. Различие в длине обусловлено размером используемых в команде адресов: 6-байтовая

команда использует 21-битный адрес блока на устройстве и 8-битное поле для длины требуемого блока данных, 8-байтовая — 32-разрядный адрес и 16-битное поле длины, а более длинные команды — 32-разрядные адрес и длину. 16-байтова форма команды содержит, кроме 32-разрядных адреса и длины, также и 32-битное поле параметра. Кроме того, команда обязательно содержит код операции и поле управления. Под код операции отводится байт. Старшие три бита этого байта кодируют формат SCB. Некоторые команды используют только один формат SCB, другие допускают применение различных форматов (обычно с адресами и полем длины различной разрядности). Кроме перечисленных в табл. 9.6, различные типы устройств имеют собственные команды.

Блоки данных, в отличие от команд, имеют практически неограниченную длину. Многие команды интерпретируют поле длины CDB как указанное в блоках. При размере блока 512 байт это позволяет передать за одну операцию до 2 Тбайт (1 Тбайт = 1024 Гбайт) данных. На практике, пакет данных, передаваемых за одну операцию шины, ограничен размерами буферов устройства и НВА, поэтому запросы на передачу больших объемов данных обрабатываются в несколько приемов. Команда может либо принимать, либо передавать данные. В качестве данных могут передаваться либо блоки параметров и ответов, либо собственно данные. Существование команд, способных и передавать, и принимать данные, явно запрещено спецификациями, поэтому в CDB предусмотрена только одна группа полей описания буфера. Впрочем, операция может исполняться в виде последовательности команд и поэтому может предусматривать двусторонний обмен данными.

Большинство команд требует от устройства более или менее длительных подготовительных операций (перемещения головки, перемотки ленты), поэтому передача данных не может следовать немедленно за командой. По этой причине большинство современных устройств поддерживает очереди команд. Чтобы связать команду с пришедшим на нее ответом, команды могут снабжаться тегом задачи. Устройство способно параллельно выполнять несколько задач, а в рамках одной задачи может быть исполнена последовательность команд.

Таблица 9.6. Список команд SCSI, поддерживаемых всеми устройствами, цит. по [www.t10.org commands]

Имя команды	Код	Тип	Описание
CHANGE DEFINITION	40h	○	Изменить версию протокола, используемого устройством
COMPARE	39h	○	Сравнить данные
COPY	18h	○	Копировать данные внутри устройства или между устройствами
COPY AND VERIFY	3Ah	○	Копировать данные с проверкой
INQUIRY	12h	М	Проверить наличие устройства, его тип и поддерживаемые им операции
LOG SELECT	4Ch	○	Включить сбор статистики
LOG SENSE	4Dh	○	Считать собранную статистику

Таблица 9.6 (окончание)

Имя команды	Код	Тип	Описание
MODE SELECT(6)	15h	Z	Выбор режима работы устройства (6-байтовая команда)
MODE SELECT(10)	55h	Z	Выбор режима работы устройства (10-байтовая команда)
MODE SENSE(6)	1Ah	Z	Считывание режима работы (6-байтовая команда)
MODE SENSE(10)	5Ah	Z	Считывание режима работы (10-байтовая команда)
MOVE MEDIUM ATTACHED	A7h	Z	Для устройств с автоматической сменой носителя
PERSISTENT RESERVE IN	5Eh	Z	Управление режимом блокировки устройства
PERSISTENT RESERVE OUT	5Fh	Z	Управление режимом блокировки устройства
PREVENT ALLOW MEDIUM REMOVAL	1Eh	O	Заблокировать сменный носитель в устройстве
READ BUFFER	3Ch	O	Чтение данных
READ ELEMENT STATUS ATTACHED	B4h	Z	Для устройств с автоматической сменой носителя
RECEIVE DIAGNOSTIC RESULTS	1Ch	O	Получить результаты самотестирования устройства
RELEASE(6)	17h	Z	Разблокировка (освобождение) устройства (6-байтовая команда)
RELEASE(10)	57h	Z	Разблокировка (освобождение) устройства (10-байтовая команда)
REPORT LUNS	A0h	O	Считывание списка логических устройств
REQUEST SENSE	03h	M	Считывание статуса устройства
RESERVE(6)	16h	Z	Блокировка (захват) устройства (6-байтовая команда)
RESERVE(10)	56h	Z	Блокировка (захват) устройства (10-байтовая команда)
SEND DIAGNOSTIC	1Dh	M	Запрос самотестирования устройства
TEST UNIT READY	00h	M	Проверка готовности устройства
WRITE BUFFER	3Bh	O	Запись данных

М — реализация команды обязательна;

О — реализация команды возможна (может быть реализована, а может и не быть);

Z — реализация и семантика команды зависит от типа устройства.

9.5. Устройства графического вывода

Если устройства графического ввода (сканеры, видеограбберы, цифровые фотоаппараты) пока еще относительно редки (хотя, по мере развития цифровой фотографии, положение быстро меняется), то устройствами графического вывода снабжается каждый современный настольный и переносной компьютер. Многие встраиваемые приложения также имеют хотя бы небольшие, но дисплеи, чаще всего жидкокристаллические.

Два основных практически применяемых типа дисплейных устройств — это *электронно-лучевые трубы*, используемые в кинескопах телевизоров и мониторах настольных компьютеров, и уже упоминавшиеся *жидкокристаллические дисплеи*. Устройства других типов — матрицы светодиодов и газоразрядные ("плазменные") панели — пока что дороги в производстве либо не всегда приемлемы по качеству, поэтому — во всяком случае, на момент написания этой книги, — еще не находят массового применения.

Принцип действия ЭЛТ широко известен. Изображение в этих устройствах формируется катодным лучом — пучком электронов, испускаемых отрицательно заряженным электродом. Вспомогательные электроды и электромагниты фокусируют луч, а два набора управляемых электромагнитов — катушки горизонтальной и вертикальной развертки — отклоняют этот луч по вертикали и горизонтали (рис. 9.24). Попадая на лицевую поверхность трубы — экран — электроны заставляют светиться нанесенный на нее краситель-люминофор, формируя, таким образом, изображение. Изменяя напряжение на катоде, можно управлять яркостью луча и, соответственно, яркостью участка изображения. Люминофор светится еще какое-то время после ухода луча с него. За счет этого, а также за счет инерционности человеческого восприятия, изображение воспринимается как более или менее неподвижная двумерная картина, а вовсе не как быстро движущийся световой "зайчик".

Для формирования цветного изображения используются три фокусируемых со смещением катодных луча и маски, обеспечивающие попадание каждого из лучей на участки предназначенного для него цветного люминофора.

Чтобы изображение занимало весь экран, луч (или тройка лучей) проводится по сложной траектории, которая приведена на рис. 9.25. Генерация этой траектории достигается простой подачей пилообразного напряжения на катушки развертки. При этом частота "пилы" горизонтальной развертки называется *частотой строчной развертки*, а вертикальной, соответственно, *частотой кадровой развертки*. Соотношение кадровой и строчной частот равно количеству строк в кадре (несколько строк приходится на кадровый гасящий импульс, поэтому видимых строк на экране чуть меньше, чем следует из соотношения частот).

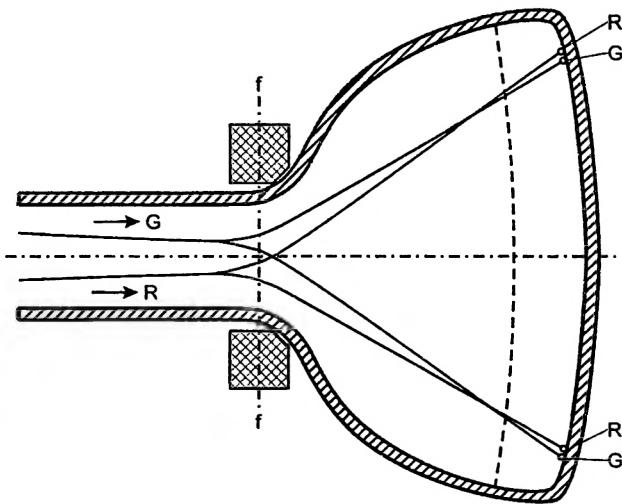
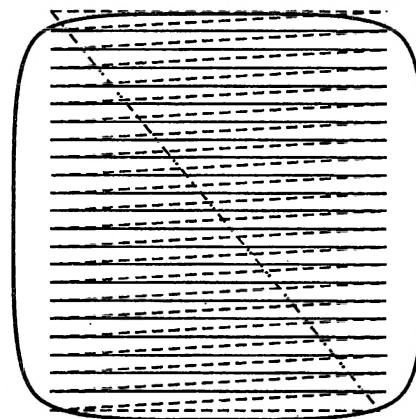


Рис. 9.24. Электронно-лучевая трубка



- Изображение
- - - Обратный ход луча
- - - Возврат луча?

Рис. 9.25. Развёртка ЭЛТ

Катушки развертки не могут мгновенно изменить направление магнитного поля, поэтому и обратный ход луча как по вертикали, так и по горизонтали, происходит не мгновенно. Чтобы луч на обратном ходе не был виден, устройство управления ЭЛТ должно генерировать *кадровый* (для вертикального обратного хода) и *строчный гасящие импульсы* (рис. 9.26).



Рис. 9.26. Стока развертки ЭЛТ

Вертикальное разрешение изображения соответствует количеству видимых строк на экране. Горизонтальное разрешение определяется двумя факторами: частотой, с которой схемы управления ЭЛТ способны модулировать луч и, у цветных мониторов, разрешением маски кинескопа. Второй параметр указывают в паспортных данных кинескопов под названием *размер точки* (*dot pitch*). У большинства мониторов первый параметр обычно более или менее соответствует второму. Электронно-лучевые трубы громоздки, создают сильные электромагнитные помехи и сами чувствительны к ним и, наконец, имеют высокое энергопотребление, но они во много раз дешевле всех альтернативных способов генерации изображений высокого качества и высокого разрешения.

Ко времени подготовки второго издания книги, мониторы на основе ЭЛТ вытесняются жидкокристаллическими дисплеями; даже те производители, которые не отказались от ЭЛТ, давно не обновляют модельные ряды и, по-видимому, не вкладывают средства в развитие производства, однако благодаря этому весьма приличные ЭЛТ-мониторы продаются по весьма низким ценам. Таким образом, хотя и очевидно, что вытеснение ЭЛТ с рынка — вопрос времени, но это время, скорее всего, будет измеряться многими годами.

Жидкокристаллические экраны основаны на способности некоторых органических соединений — жидких кристаллов — менять свою прозрачность и другие оптические свойства под воздействием электрического поля. Жидкокристаллический экран представляет собой две стеклянные или прозрачные пластиковые пластины (обкладки), на которые нанесены полупрозрачные металлические электроды. Пространство между пластинаами заполнено жидким кристаллом. За пластинаами находится подложка — у черно-белых дисплеев зеркальная или черная, у цветных дисплеев — снабженная поляризационным фильтром. Для повышения контрастности изображения подложка нередко подсвечивается.

Подавая напряжение на электроды, контроллер ЖКД может избирательно делать прозрачными те или иные участки экрана и, таким образом, формиро-

вать различные изображения. Изменяя напряжение, можно в определенных пределах управлять яркостью или, скорее, контрастностью изображения. В современных цветных дисплеях высокого разрешения используется жидкий кристалл, поворачивающий плоскость поляризации проходящего через него света. Этот угол зависит от напряжения на электродах и от частоты световой волны. Снабдив обкладки экрана поляризационными фильтрами, можно управлять цветом участка экрана.

Формы электродов ЖКД отличаются большим разнообразием. Нередко применяются прямоугольные матрицы точек, позволяющие создавать произвольные растровые изображения. Однако многие приложения — часы, калькуляторы, простые дисплеи — не требуют произвольных изображений, поэтому часто изготавливают пластины с электродами сложной формы, соответствующей элементам цифр и букв и/или различным пиктограммам.

ЖКД низкого разрешения дешевы, компактны, имеют низкое энергопотребление и находят широкое применение в самых разнообразных устройствах — сотовых и стационарных телефонах, калькуляторах, часах, измерительных и бытовых приборах. Однако высококачественные цветные ЖКД большой площади представляют собой весьма дорогостоящие устройства, цена которых составляет более половины цены современных портативных компьютеров. Относительно дешевые ЖКД больших размеров (вплоть до 21 дюйма по диагонали), вышедшие на рынок в начале столетия, нередко значительно уступают по качеству матрицам, применяемым в портативных компьютерах.

Основным недостатком типичного современного ЖКД для ноутбука следует признать ограничения угла зрения: при взгляде на дисплей не по нормали к его плоскости, расстояние, проходимое лучом света в жидким кристалле изменяется, соответственно изменяется и угол поворота плоскости поляризации, поэтому цвета изображения сильно искажаются.

Список распространенных недостатков дешевых стационарных ЖКД гораздо более обширен и включает в себя:

- инерционность матрицы: при быстром изменении изображения, даже если напряжение на обкладках изменяется достаточно быстро, жидкий кристалл не успевает перестроиться; при этом скорость перестройки определяется изменением напряжения, так что переход от черного к белому часто происходит намного быстрее, чем от одной градации серого к другой;
- нелинейность при передаче цвета; особенно раздражает, когда функции яркости для разных цветов различны, но этим недостатком страдает большинство дешевых дисплеев;
- недостаточный диапазон яростей: слишком темные или слишком яркие цвета "съедаются", оказываются неотличимы от черного или, соответственно, белого;

- недостаточная контрастность;
- видимые границы между пикселями и т. д.

Подача напряжения на каждую пару электродов ЖКД отдельным контроллером недопустимо дорога. К счастью, пара электродов представляет собой конденсатор, который способен некоторое время без вмешательства извне сохранять электрический заряд и, следовательно, изображение. Наличие у пикселя электрической емкости позволяет свести поддержание изображения на жидкокристаллической матрице к аналогу развертки ЭЛТ — периодическому сканированию электродов с подачей на них напряжения, соответствующего яркости элемента изображения (рис. 9.27).

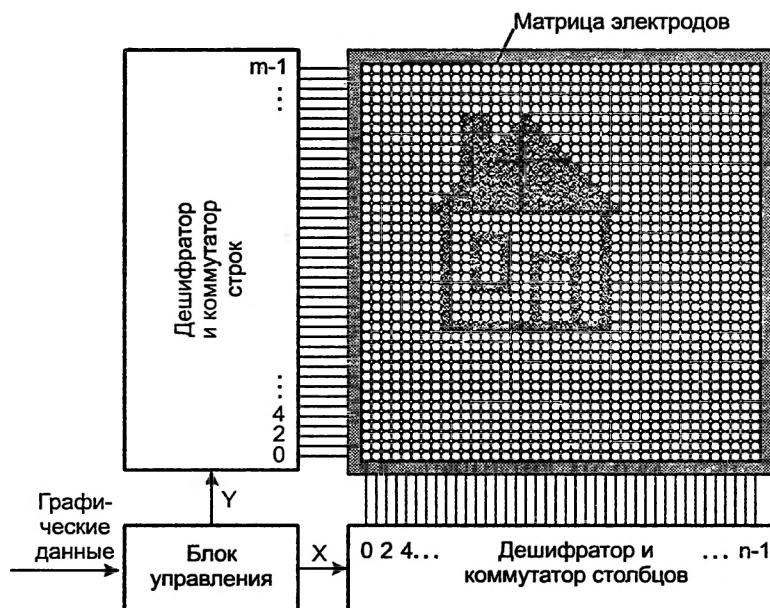


Рис. 9.27. Схема формирования изображения на жидкокристаллическом дисплее

При изменении изображения емкость элементов матрицы скорее вредна — быстрая перезарядка элемента требует больших токов, поэтому многие ЖКД, в том числе и дисплеи старых переносных компьютеров, обладают высокой инерционностью изображения. В современных высококачественных дисплеях каждый пиксель экрана снабжается собственной усилительной схемой, способной быстро перезарядить электроды. Такие экраны называются *активной матрицей*. Широкое распространение получили активные матрицы с усилителями, реализованными на основе *TFT* (*Thin Film Transistor* — тонкопленочный транзистор), нанесенных на обкладки экрана.

Несмотря на коренное различие физических способов формирования изображения в электронно-лучевых трубках и жидкокристаллических панелях, функции контроллера этих устройств весьма похожи и сводятся к передаче значений яркости элементов изображения с частотой, соответствующей частоте развертки дисплея. Контроллер, предназначенный для работы с ЭЛТ, должен уметь также делать в потоке данных паузы, соответствующие строчным и кадровым гасящим импульсам, и поэтому должен быть несколько сложнее. Однако общая структура графических контроллеров для дисплеев различных типов удивительно похожа.

Типичный графический контроллер (рис. 9.28) состоит из:

- видеобуфера (*frame buffer* — буфер кадра) — блока памяти, элементы которого соответствуют пикселям дисплея;
- генератора тактовой частоты;
- собственно контроллера — устройства, которое передает содержимое видеобуфера, а также, если это необходимо, гасящие импульсы и синхросигналы аналоговым схемам управления дисплеем.
- ЦАП (цифро-аналогового преобразователя), который преобразует значения пикселов в аналоговый сигнал, и, если это необходимо, усилителей видеосигнала.

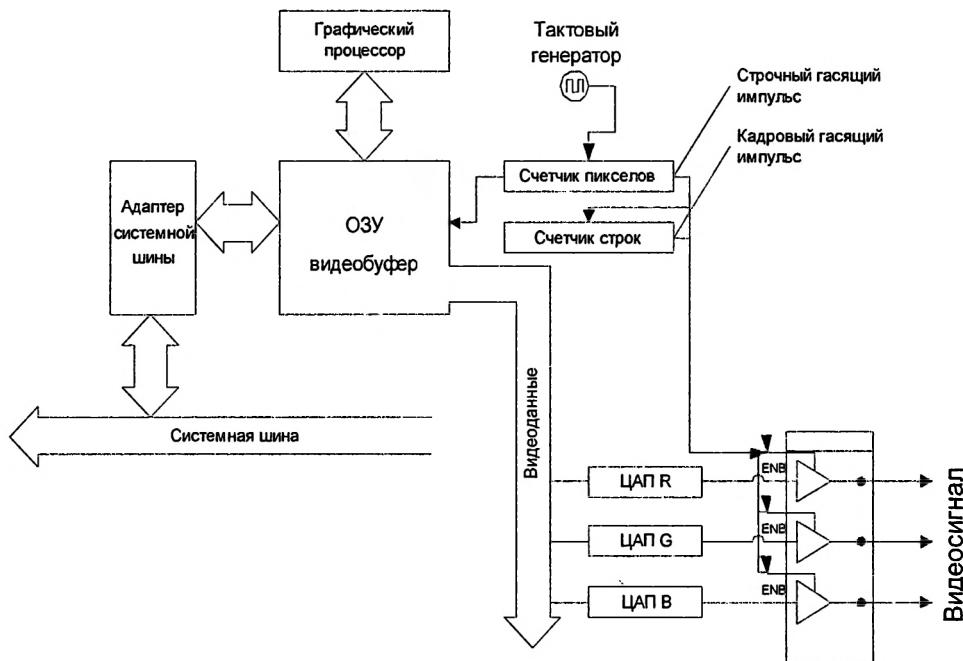


Рис. 9.28. Схема графического контроллера

Структура видеобуфера определяется цветовой глубиной дисплея. У черно-белых дисплеев обычно один бит соответствует одному пикселу, а один байт видеобуфера — восьми последовательным пикселям изображения. У черно-белых с градацией яркости ("черно-бело-серых") и цветных дисплеев видеобуфер должен иметь по несколько битов на пиксел.

Это достигается двумя способами. Первый состоит в организации *битовых плоскостей*, когда в видеобуфере последовательно хранятся сначала младшие биты значений пикселов, затем — вторые и т. д. (рис. 9.29). При этом восемь последовательных пикселов описываются несколькими байтами, размещенными в разных частях видеобуфера. Эта, немного странная на первый взгляд, логика кодирования изображения удобна тем, что облегчает перенос программного обеспечения, предназначенного для работы с черно-белыми дисплеями, на цветные дисплеи.

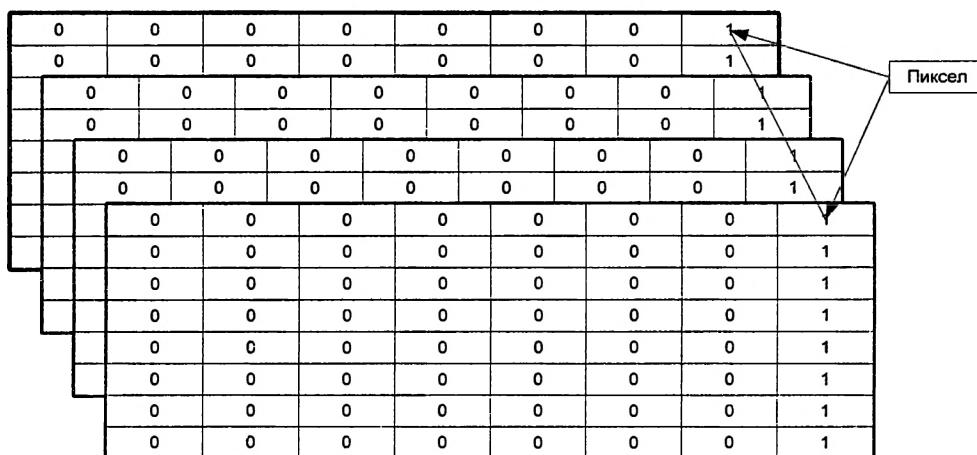


Рис. 9.29. Битовые плоскости

Второй подход, чаще применяемый при больших цветовых глубинах, состоит в кодировании одного пикселя одним или несколькими последовательными байтами видеобуфера. Графические контроллеры современных персональных компьютеров имеют цветовую глубину 24 или 32 бита, что соответствует трем или четырем байтам на пиксель. Этого хватает, чтобы (с учетом диапазона яркости современных дисплеев) представить все цвета, которые человеческий глаз способен отличить на дисплее друг от друга.

Контроллеры с такой цветовой глубиной обычно предоставляют по одному байту для кодирования каждой из трех цветовых составляющих цветного изображения. Устройства с меньшими цветовыми глубинами часто реализуют более сложную схему кодирования цвета, называемую *отображением*

цветов (*color mapping*). Значение пикселя при этом представляет собой индекс в специальной таблице, палитре или карте цветов (*color map*). Элементы палитры — это значения цветовых компонентов пикселя.

Большинство контроллеров включают в себя также более или менее сложную логику управления содержимым видеобуфера со стороны центрального процессора. Простейшим случаем такого управления является отображение видеобуфера на адресное пространство системной шины. Это, привлекательное во многих отношениях, решение не всегда применимо, например, если адресное пространство процессора слишком мало или плотно занято, либо устройство не подключается непосредственно ни к системной, ни к периферийной шине (например, контроллеры жидкокристаллических дисплеев, предназначенных для использования во встраиваемых приложениях, используют для общения с микропроцессором шину I²C или нестандартные протоколы последовательных портов).

При простой нехватке адресного пространства видеобуфернередко отображают на адресное пространство не целиком, а частями, банками. При этом контроллер должен иметь специальный регистр — селектор банка.

Более радикальное решение, применимое даже при подключении через последовательный порт, состоит в том, чтобы предоставить процессору два регистра — адрес в видеопамяти и регистр данных, соответствующий ячейке видеопамяти по этому адресу. Это решение можно считать вырожденным случаем банков видеопамяти, когда банк имеет размер один байт. При этом требуется всего два регистра в адресном пространстве ввода/вывода (этот метод редко применяется с видеобуферами объемом более 64 Кбайт), но доступ к видеобуферу замедляется минимум вдвое. При использовании последовательных шин и портов адрес и данные передаются последовательно. Многие графические контроллеры с такой организацией предоставляют различные способы доступа с автоинкрементом, позволяющие передать в буфер последовательность байтов без явного доступа к регистру адреса.

Контроллеры с битовыми плоскостями часто предоставляют групповые операции над байтами, кодирующими биты смежных пикселов.

Контроллеры, применяемые в современных персональных компьютерах и рабочих станциях, содержат более или менее сложные видеопроцессоры, способные без участия ЦПУ рисовать в видеобуфере различные графические примитивы, начиная от прямых линий и окружностей, и заканчивая проекциями и/или фотoreалистичными изображениями трехмерных объектов, описываемых языком OpenGL.

Видеоконтроллеры представляют собой довольно сложные устройства. Подробные описания современных видеоконтроллеров можно найти во многих доступных книгах, например в [Гук 2000]. В документе [www.microchip.com

PICMicro] описывается встроенный контроллер жидкокристаллического дисплея микроконтроллера PIC.

9.6. Запоминающие устройства прямого доступа

Приобрел себе винчестер у дороги во кустах,
Обнаружились бол-блоки в восемнадцати местах.
Ах!

Ю. Нестеренко

Как уже говорилось, основную массу устройств этого типа составляют диски — жесткие и гибкие, магнитные, магнитооптические и оптические. Ассоциация между понятиями "постоянное запоминающее устройство прямого доступа" и "диск" столь плотно укоренилась в общественном сознании, что запоминающие устройства, основанные на иных принципах, например, банки флэш-памяти, иногда называют "твердотельными дисками". В этой главе мы будем рассматривать преимущественно устройство магнитных дисковых накопителей. Принцип устройства дискового накопителя широко известен [Гук 2000].

Для хранения данных служит диск, покрытый ферромагнитным слоем. В современных накопителях запись осуществляется на обе стороны диска. Многие приводы с неудаляемыми дисками имеют несколько дисков. Диск может быть как гибким, так и жестким. Диск насажен на ось, называемую *шпиндель*.

В зависимости от способа крепления, диски делятся на съемные (удаляемые) и несъемные (фиксированные). В действительности, съемность накладывает серьезные ограничения не только на крепление диска к оси, но и на конструкцию блока головок и привода в целом, а также предъявляет определенные требования к контроллеру — тот должен отслеживать наличие диска в приводе и факт его замены. Несъемные диски, как правило, помещаются в герметичный корпус, защищающий головки и поверхность диска от пыли и других вредных атмосферных воздействий (рис. 9.30).

Включение двигателя и раскрутка шпинделя до рабочей скорости обязательно производятся до начала форматирования диска или обмена данными с ним. Контроллеры съемных дисков обычно останавливают мотор в то время, когда накопитель не используется, контроллеры же дисков несъемных обычно останавливают шпиндель только при выключении питания.

Запись на магнитный диск и считывание данных с него осуществляется головкой чтения/записи, по принципу действия похожей на головку аналогового магнитофона. Головка состоит из ферромагнитного сердечника, на который намотан провод. Сердечник имеет разрез, как раз в той части, которая

ближе всего к поверхности диска. Подавая ток по проводу, можно создавать в сердечнике и, соответственно, в разрезе сердечника магнитное поле, которое будет намагничивать диск. Таким образом осуществляется запись данных. С другой стороны, когда направление намагченности проходящего под головкой участка диска меняется, в проводах возникает индуктивный ток — детектируя его, можно осуществлять считывание. В отличие от простой передачи по проводу, головка считывает не сам записанный сигнал, а его производную по времени (в аналоговых магнитофонах это компенсируется аналоговым интегрированием сигнала).



Рис. 9.30. Жесткие диски в герметичном корпусе ("винчестеры"). Слева — диск Quantum Bigfoot емкостью 1.2 Гбайт с интерфейсом Parallel ATA, справа — диск Seagate ST-4053 емкостью 40 Мбайт с интерфейсом MFM. Дискета используется для масштаба

Способ кодирования нулей и единиц при такой записи представляет собой несколько более сложную задачу, чем при немодулированной передаче данных по проводу. Видно, что записать на диск постоянное высокое или низкое напряжение (соответствующее последовательности нулей или единиц) в принципе, можно — в виде участка дорожки с постоянной намагченностью, но такая последовательность не может быть считана головкой. Считывать можно только изменения намагченности. Таким образом, использовать простые изменения напряжения для кодирования единиц и нулей невозможно, и мы должны изобрести более сложную схему кодирования.

Выбирая схему кодирования, мы должны обязательно решить также вопрос о способе синхронизации считывания: добиться высокой стабильности от

мотора, врачающего шпиндель, невозможно. Кроме того, в приводах низкоскоростных, особенно гибких, дисков приходится считаться с опасностью механического заедания подшипников и самого диска, а в приводах жестких дисков с высокой скоростью вращения — с механическими вибрациями всей конструкции, которые могут повлиять на скорость прохождения диска под головкой даже при стабильной средней угловой скорости шпинделя.

На лентопротяжных устройствах с несколькими головками нередко используется отдельная синхродорожка, но на дисковых устройствах ее введение потребовало бы существенного усложнения и недопустимого утяжеления блока головок, поэтому обычно применяются схемы кодирования с совмещенным синхросигналом.

Простейшая форма такого кодирования, называемая *DFM (Double Frequency Modulation — модуляция с двойной частотой)*, приведена на рис. 9.31. Видно, что при записи нуля мы осуществляем одно изменение направления тока за интервал кодирования одного бита, а при кодировании единицы — два, причем первое из этих изменений смещено относительно начала интервала. Такое кодирование использовалось на 8-дюймовых гибких дисках.

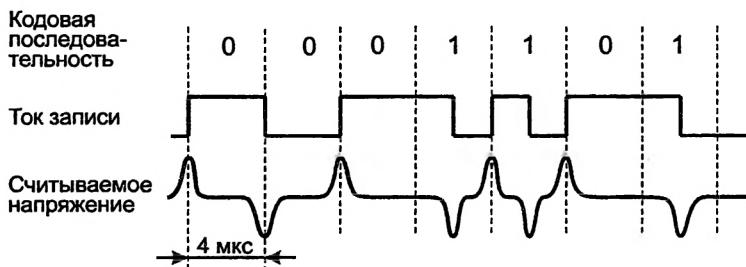


Рис. 9.31. Модуляция с двойной частотой

Несколько более сложная схема кодирования, приведенная на рис. 9.32, использовалась на 5- и 3-дюймовых гибких дисках низкой плотности. Эта схема называется модифицированной фазовой модуляцией. На жестких дисках в 80-е годы XX века использовались частотная и модифицированная частотная модуляция.

Позднее, уже в конце 80-х годов, появились жесткие диски, использующие модуляцию RLL (Run-Length Limited, ограниченная длина [повторяющихся] последовательностей). RLL обозначает обширное семейство способов кодирования и модуляции, в которых различными способами ограничено количество повторяющихся битов в потоке данных, благодаря чему можно исключить из потока синхросигнал и повысить информационную плотность почти вдвое по сравнению с простыми схемами кодирования, такими как МЧМ.

В действительности, выигрыш от использования RLL получается меньше, чем в два раза, потому что все схемы кодирования с ограничением длин последовательностей так или иначе предполагают включение в поток дополнительных битов либо замены всех возможных последовательностей битов на более длинные. Сами схемы кодирования отличаются большим разнообразием; среди них надо упомянуть уже встречавшееся нам битовое заполнение (в каждую достаточно длинную последовательность нулей обязательно вставляется единица) и логическое кодирование, когда для каждой из возможных битовых последовательностей определенной длины задается уникальная кодовая последовательность. Соответствие битовых и кодовых последовательностей определяется по прошитой в контроллер таблице. Каждая схема кодирования описывается двумя параметрами; RLL (X,Y) обозначает кодовую последовательность, в которой минимальный интервал между сменами фазы намагничивания составляет X тактов, а максимальный — Y.

На жестких дисках применялись схемы логического кодирования RLL (1,7) и RLL (2,7).

Современные жесткие диски используют сложные схемы группового кодирования, когда одно изменение фазы или частоты намагничивания кодирует несколько битов.

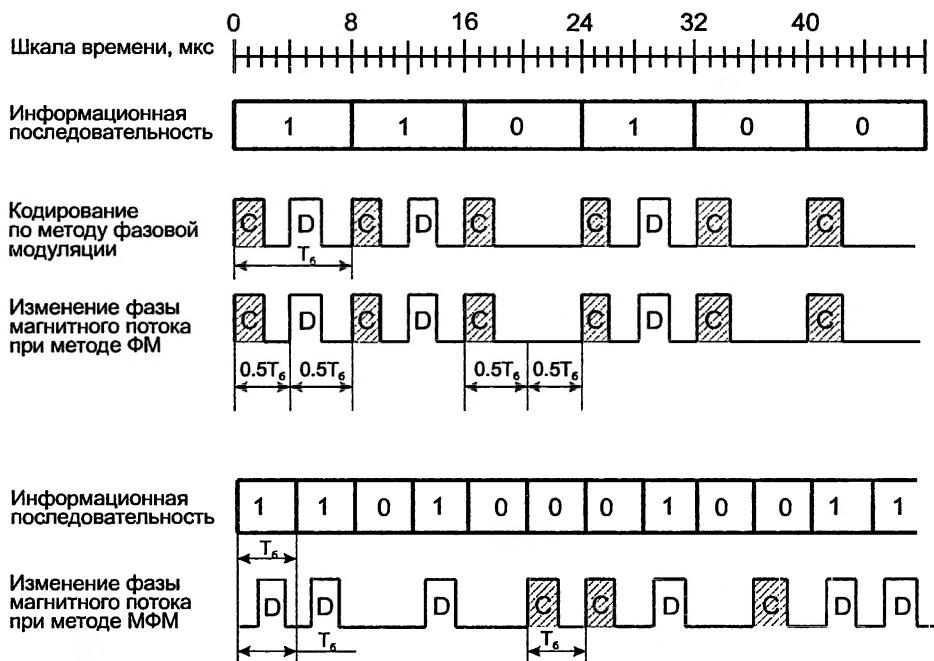


Рис. 9.32. Простая и модифицированная фазовая модуляция

Головка чтения/записи может перемещаться вдоль радиуса диска шаговым электродвигателем. У жестких дисков головка обычно размещена на рычаге, напоминающем звукосниматель граммофона (рис. 9.33). У гибких дисков головка движется по направляющим под действием червячной передачи, а у CD-ROM — зубчатой рейки (рис. 9.34). Накопители, имеющие более одной

Seagate ST4053:

40 MByte

5x 1/4 inch

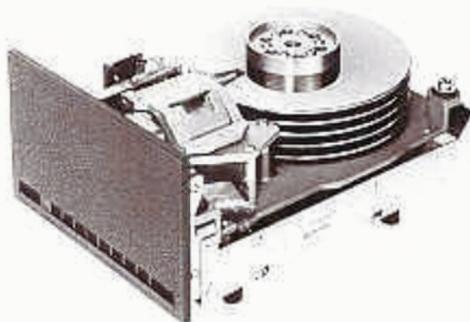


Рис. 9.33. Жесткий диск Seagate ST-4053 со снятой крышкой корпуса

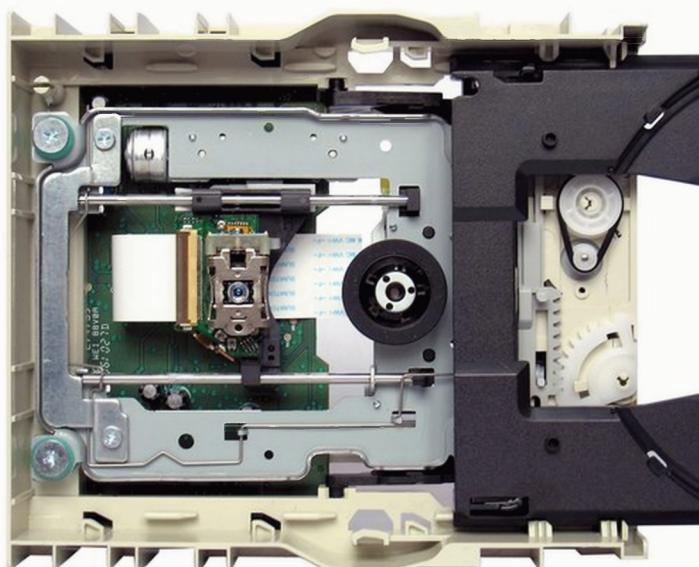


Рис. 9.34. Привод CD-ROM. В окне справа видны шестерня и зубчатая рейка подачи головки

рабочей поверхности, имеют столько же головок, сколько и поверхностей, но подача этого блока головок все равно осуществляется одним двигателем.

Когда блок головок неподвижен, каждая головка может считывать данные, записанные на диске в виде кольцевой *дорожки* (*track*). Совокупность дорожек всех поверхностей, соответствующих одному положению блока головок, образует *цилиндр* (*cylinder*). Количество цилиндров у накопителя определяется шириной магнитной головки (и обусловленной ею шириной намагниченной полосы) и точностью, которую может обеспечить механика подачи головки. Стандартные приводы 3-дюймовых дисков имеют 80 дорожек. Количество цилиндров у современных жестких дисков достигает нескольких тысяч. При всех перечисленных выше, а также при более сложных современных способах модуляции запись данных на дорожку осуществляется *блоками* или *секторами*. Секторы аналогичны кадрам, которыми осуществляется передача данных через последовательные порты и шины. Сектор состоит из заголовка и блока данных. Заголовок обычно содержит номера дорожки (чтобы контроллер мог убедиться, что правильно позиционировал головку) и сектора на дорожке, а иногда также и поверхности. Кроме того, заголовок практически всегда содержит контрольную сумму или иногда две отдельных контрольных суммы — для заголовка и для данных. Пространство между секторами заполнено специальными зонами, служащими для выравнивания и синхронизации (рис. 9.35 и табл. 9.7). Нередко используются также специальные маркеры — последовательности изменений сигнала, которые не могут появиться при принятой схеме модуляции. Маркеры используются для отметки начала дорожки или, реже, начала сектора.

Таблица 9.7. Структура сектора дискеты с двойной плотностью, цит. по [МикроЭВМ 1988]

Длина в байтах	Описание
12	Зона синхронизации
4	Адресный маркер
1	Номер дорожки
1	Номер стороны
1	Номер сектора
1	Длина сектора
2	Циклическая контрольная сумма
22	Зона типа 2 (см. рис. 9.35)
12	Зона синхронизации
4	Маркер данных

Таблица 9.7 (окончание)

Длина в байтах	Описание
256/512/1024	Данные
2	Циклическая контрольная сумма
36–116	Зона типа 3 (см. рис. 9.35)

Примечание. Зона типа 1 длиной 32 байта отмечает начало дорожки, а зона типа 4 длиной от 118 до 266 байт служит для заполнения дорожки до полной длины.

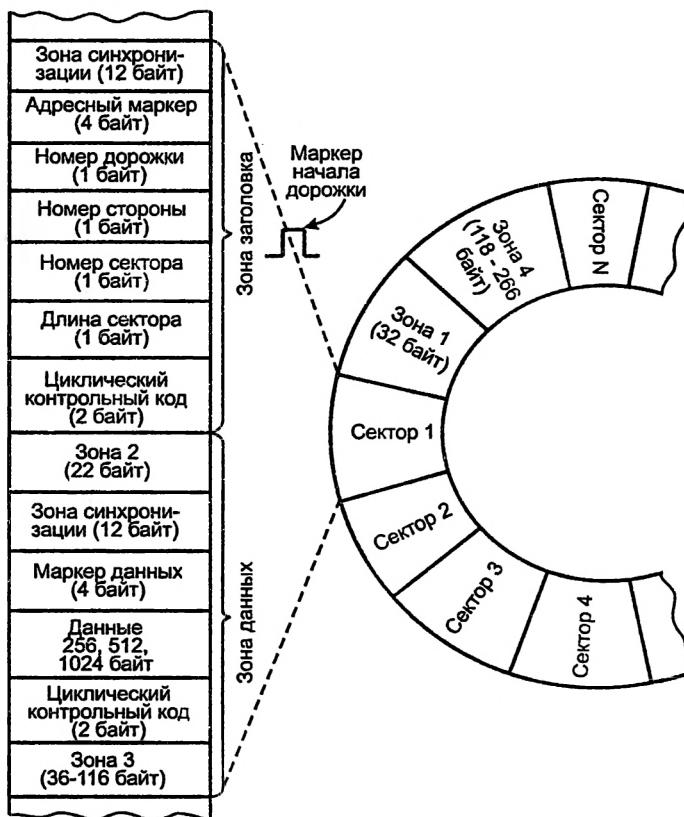


Рис. 9.35. Структура дорожки дискеты с двойной плотностью

Перед тем как диск может быть использован для записи данных, он должен быть отформатирован или, как говорили раньше, размечен — на его дорожки должны быть записаны заголовки секторов с правильными номерами дорожки и сектора, а также, если это необходимо, маркеры. Как правило, при этом

же происходит тестирование поверхности диска для поиска дефектов магнитного слоя. Не следует путать эту операцию — *физическое форматирование диска* — с логическим форматированием, созданием файловых систем. Современные жесткие диски конструктивов ATA и SCSI обычно требуют физического форматирования в заводских условиях.

Количество секторов на одной дорожке определяется, с одной стороны, длиной сектора, а с другой — частотой модуляции. Частота модуляции, в свою очередь, ограничена частотной характеристикой схемы управления магнитной головкой, индуктивностью самой головки и параметрами ферромагнитной поверхности диска (размером минимального домена намагничивания). Последний фактор фактически ограничивает линейную плотность записи (количество битов на миллиметр или дюйм длины дорожки), поэтому на более длинных внешних дорожках целесообразно делать больше секторов, чем на внутренних (рис. 9.36). Это решение усложняет адресацию секторов и логику контроллера, поэтому начало широко применяться лишь относительно недавно.

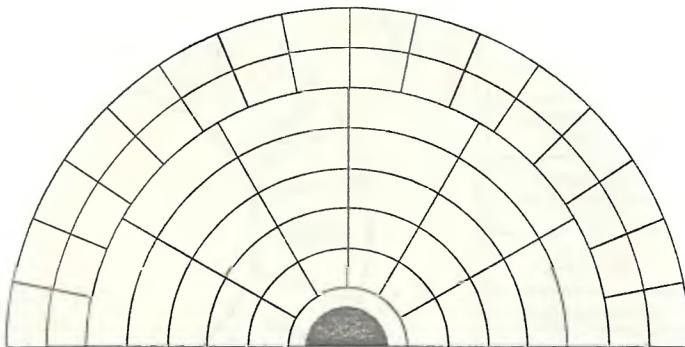


Рис. 9.36. Диск с переменным количеством секторов на дорожках

Привод магнитного диска, таким образом, состоит из трех разнородных электромеханических и аналоговых устройств, управление которыми должно осуществляться в строгом согласовании:

- электродвигателя шпинделя;
- шагового двигателя подачи головки;
- аналоговых каскадов управления магнитной головкой.

Устройство, управляющее всем этим, называется *дисковым контроллером* и состоит из кодеров и декодеров используемой схемы модуляции, логики, формирующей заголовок сектора при записи или проверяющей его целостность при считывании, буфера данных сектора и сдвигового регистра, подключенного к кодеру и декодеру управления головкой (рис. 9.37). Современ-

ные контроллеры обычно содержат более сложную логику, обеспечивающую передачу данных в основное ОЗУ компьютера в режиме ПДП (в том числе и распределение/сборку), очередь обслуживаемых запросов, опережающее считывание, отложенную запись и кэширование данных, переадресацию дефектных секторов и др.

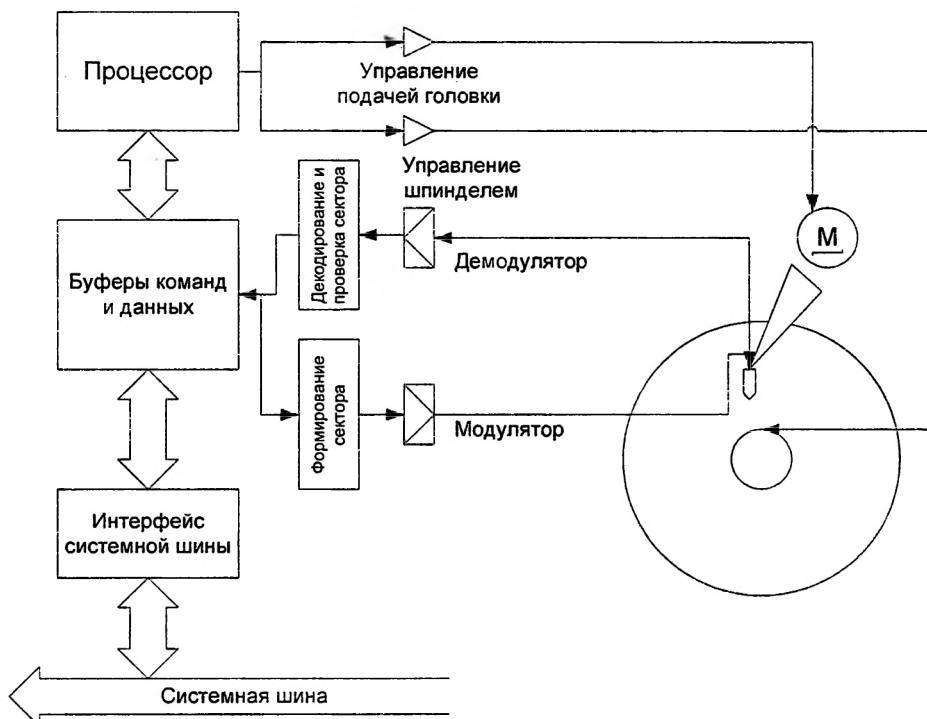


Рис. 9.37. Схема контроллера диска

Контроллер гибких дисков ДВК

В качестве простейшего дискового контроллера рассмотрим микросхему K1801ВП1-097 [МикроЭВМ 1988] (этот микросхема является советским функциональным аналогом микросхемы контроллера гибких магнитных дисков, разработанной фирмой DEC для использования в мини- и микрокомпьютерах семейства PDP-11. К сожалению, мне не удалось найти маркировку оригинальной микросхемы и ссылку на документацию о ней).

Микросхема допускает подключение четырех приводов 5-дюймовых гибких дисков и содержит в себе порты управления основным и шаговым электродвигателями привода, сдвиговый регистр, модулятор и демодулятор МФМ, логику обнаружения маркеров, генератор циклической контрольной суммы и интерфейс системной шины Q-Bus. У микросхемы есть два 16-разрядных регистра, регистр управления и регистр данных. Описание битов регистра управления приведено в табл. 9.8.

Таблица 9.8. Управляющий регистр K1801ВП1-097

Бит	Описание	Комментарий
00 (W)	Выбор накопителя 0	—
00 (R)	Дорожка 0	Головка чтения/записи установлена на дорожку 0
01 (W)	Выбор накопителя 1	—
01 (R)	Накопитель готов	—
02 (W)	Выбор накопителя 2	—
02 (R)	Запись запрещена	На диске установленна защита записи
03 (W)	Выбор накопителя 3	—
04	Включение двигателя	—
05	Поверхность	Если 1, запись (чтение) происходит с верхней поверхности диска
06	Направление шага	При записи 1, направление к оси диска
07 (W)	Шаг	При записи 1 в этот бит, головка перемещается на один шаг и бит очищается
07 (R)	Данные готовы	Контроллер выставляет этот бит, когда считано или записано очередное слово данных. ЦПУ должно прочитать или записать следующее слово сектора
08	Частичный сброс	
09	Запись маркера	
10	Включение внешней схемы прекоррекции фазовых искажений	
14	Успех операции	0, если операция завершилась ошибкой
15	Начало дорожки	

Видно, что микросхема предусматривает работу с ней центрального процессора в режиме опроса. Более сложный контроллер КМД (Контроллер Минидисков), основанный на этой микросхеме, содержит микропроцессор, ПЗУ с программой для него и 2 Кбайт ОЗУ (которое может использоваться и для промежуточных данных программы микропроцессора, и для хранения самих данных передаваемого блока) [МикроЭВМ 1988]. Работа с этим контроллером осуществляется посредством двух регистров, команды/состояния и данных. Центральный процессор записывает в эти регистры команду (табл. 9.9) и адрес блока параметров в основном ОЗУ. Контроллер в режиме ПДП считывает блок параметров и пытается выполнить операцию, передавая данные, если это необходимо, также в режиме ПДП. Указатель на блок данных содержится в блоке параметров. Одной операцией можно прочитать или записать несколько последовательных секторов. После завершения передачи контроллер генерирует прерывание.

Таблица 9.9. Список команд контроллера КМД, цит. по [МикроЭВМ 1988]

Код	Мнемоника	Описание
0h	RD	Чтение
1h	WR	Запись
2h	RDM	Чтение с маркером
3h	WRM	Запись с маркером
4h	RDTR	Чтение дорожки
5h	RDID	Чтение заголовка
6h	FORMAT	Форматирование дорожки
7h	SEEK	Подача головки
8h	SET	Установка параметров
9h	RDERR	Чтение состояния и ошибок
Fh	LOAD	Загрузка

Контроллер жестких дисков ST506

Более сложный контроллер ST506 был разработан фирмой Western Digital в конце 70-х годов прошлого века для управления жесткими дисками, подключаемыми к микропроцессорным системам. Этот контроллер и его усовершенствованная версия, ST412, широко применялись для подключения жестких дисков к IBM PC и совместимым с ними компьютерам [citforum.ukrpak.net IDE, Гук 2000, PC Guide IDE].

Контроллер ST506 допускал подключение двух жестких дисков емкостью до 40 Мбайт, использующих модифицированную частотную модуляцию, а более поздние версии контроллера — модуляцию RLL.

Как и описанный выше КМД, контроллер содержал логику управления мотором шпинделя и подачей головок, буфер для одного сектора и схемы модуляции и демодуляции МЧМ. Диски присоединялись к контроллеру двумя плоскими кабелями: один, с 34 проводами, использовался для передачи сигналов управления двигателями; а второй, 20-проводной, — для передачи модулированных сигналов МЧМ.

Центральному процессору контроллер ST506 доступен в виде двух блоков регистров, управляющего и командного. Блок управляющих регистров размещается по адресам 0x3F4-0x3F7, а блок командных — по адресам 1F0-1F7 (табл. 9.10).

Управляющие регистры используются для доступа к регистрам данных и статуса без подачи контроллеру сигнала о том, что регистр прочитан (смысл этого станет ясен далее). Пожалуй, единственный самостоятельно полезный бит в этих регистрах управляет разрешением прерываний от контроллера. Основная работа с контроллером происходит через блок из восьми командных регистров, один из которых является регистром данных.

При исполнении команды записи контроллер подает головку к дорожке, указанной регистрами CY и DH, генерирует прерывание и выставляет сигнал DRQ (табл. 9.11), сигнализируя процессору, что готов принять данные. Затем процессор производит 512 операций записи в порт данных, заполняя буфер. Контроллер выполняет запись, генерирует прерывание и, если счетчик секторов еще не сравнялся с нулем, увеличивает номер сектора и снова выставляет сигнал DRQ.

Операция чтения выполняется аналогично, с той лишь разницей, что контроллер сначала считывает сектор, и лишь потом выставляет DRQ. Циклы чтения и записи регистра данных, таким образом, приводят к увеличению счетчика буфера.

Набор команд контроллера приведен в табл. 9.12.

Таблица 9.10. Командные регистры контроллера ST506, цит. по [chip.ms.mff.cuni.cz ATA2]

Адрес	Функция (чтение)	Функция (запись)	Код
0x1F0	Данные (младш.)	Данные (младш.)	
0x1F1	Ошибка	Прекомпенсация	PC
0x1F2	Счетчик секторов	– " –	SC
0x1F3	Номер сектора	– " –	SN
0x1F4	Цилиндр (младш.)	– " –	CY
0x1F5	Цилиндр (старш.)	– " –	CY
0x1F6	Устройство/поверхность	– " –	DH
0x1F7	Статус	Команда	

Таблица 9.11. Биты регистра статуса ST506, цит. по [chip.ms.mff.cuni.cz IDE]

Бит	Мнемоника	Описание
7	BUSY	Устройство занято
6	DRDY	Устройство готово принимать команды
5	DWF	Ошибка записи
4	DSC	Подача головки завершена
3	DRQ	Запрос данных
2	CORR	Исправимая ошибка при чтении данных
1	INDEX	Маркер начала дорожки
0	ERROR	Ошибка

Таблица 9.12. Команды контроллера ST506, цит. по [chip.ms.mff.cuni.cz IDE]

Код	Описание	PC	SC	SN	CY	DH
90h	Диагностика привода	—	—	—	—	D+
50h	Форматировать дорожку	—	—	—	V	V
20h	Чтение секторов с повтором	—	V	V	V	V
21h	Чтение секторов	—	V	V	V	V
40h	Проверка секторов с повтором	—	V	V	V	V
41h	Проверка секторов	—	V	V	V	V
1Xh	Возврат к дорожке 0	—	—	—	—	D
7Xh	Подача головки	—	—	—	V	V
30h	Запись секторов с повтором	—	V	V	V	V
31h	Запись секторов	—	V	V	V	V

V — используется значение регистра.

D — используется только бит выбора диска.

D+ — независимо от бита выбора диска, реагируют оба диска.

Контроллеры жестких дисков ESDI, IDE, EIDE

Дальнейшее совершенствование ST506 шло в направлении переноса функций контроллера на сам жесткий диск. Первым шагом стал интерфейс ESDI, предложенный фирмой IBM в 1985 году. В ESDI модулирование сигнала выполнялось не контроллером, а платой, установленной на диске. Это позволило, впервых, использовать жесткие диски со схемами модуляции, отличными от МЧМ и RLL и, во-вторых, увеличить допустимую длину кабелей до трех метров. Контроллер ESDI по регистрам и набору команд был полностью совместим с ST506/412, но обеспечивал работу с дисками большей емкости и большей производительности.

Следующий, более радикальный, шаг был сделан в следующем году фирмами Western Digital и Compaq, предложившими перенести весь контроллер на жесткий диск. Соответствующий конструктив был назван IDE (*Integrated Drive Electronics* — интегрированная на приводе электроника).

40-жильный плоский кабель IDE представляет собой, фактически, расширение шин адреса и данных ISA. По нему передаются 16 бит данных и пять линий адреса. Первые две линии адреса выбирают группу регистров — контролльный или командный блок регистров, расположенных по тем же адресам шины ISA, что и соответствующие регистры ST506. Последние три линии выбирают один из регистров соответствующей группы и через линии данных кабеля подключают его кшине данных ISA.

IDE, как и ST506, допускает подключение двух жестких дисков. Каждый из дисков имеет свой контроллер и свои блоки регистров. Когда ЦПУ производит запись в эти регистры, она происходит в регистры обоих контроллеров. Однако на

запросы чтения и, тем более, исполнения команды откликается только тот контроллер, который выбран регистром DH. Эта весьма своеобразная архитектура позволила сохранить полную программную совместимость с контроллером ST506 при полном же изменении конструктива. Основным недостатком этой архитектуры является тот печальный факт, что пока один диск отрабатывает команду или передает данные, обращения ко второму диску невозможны.

IDE несколько позднее получил статус стандарта ANSI под названием ATA (AT Attachment — [интерфейс] присоединения к IBM PC/AT) [www.t13.org], и до сих пор является основным способом подключения жестких дисков к настольным компьютерам и иногда даже к маломощным серверам, причем не только на основе процессоров x86, но и PowerMac, SPARCStation и др.

Современные версии стандарта поддерживают обмен 16-разрядными данными (при нормальной отработке команды код ошибки не нужен, а прекомпенсация современными контроллерами вообще не используется, поэтому для данных можно использовать регистры 0x1F0 и 0x1F1, образующие при этом единый 16-разрядный регистр), автоидентификацию геометрии и других параметров жестких дисков, логическую адресацию блоков (LBA — Logical Block Addressing, когда регистры SN, CY и 4 младших бита DH образуют линейный 28-битный адрес сектора), работу в режиме ПДП и передачу данных со скоростями до 133 Мбайт/с [Гук 2000, citforum.ukrpan.net IDE, PC Guide IDE, www.t13.org].

Расширение протокола ATAPI (ATA Packet Interface — пакетный интерфейс ATA) предусматривает также возможность управления недисковыми устройствами (CD-ROM, стримерами, магнитооптическими дисками) путем передачи через регистр данных 12-байтовых блоков команд, аналогичных командам SCSI.

Пожалуй, самый значительный за последние годы прорыв в развитии технологии ATA — это стандарт SATA (Serial ATA), предполагающий отказ от плоских параллельных кабелей с разъемами для двух дисков и переход к последовательной передаче данных по витой паре. При этом сохраняется совместимость с контроллерами ATA по регистрам и командам, так что драйверы ST506/IDE/EIDE могут поддерживать контроллеры SATA без изменений или лишь с небольшими модификациями.

В отличие от параллельного ATA, SATA предполагает подключение каждым кабелем только одного диска, что облегчает синхронизацию адаптера системной шины и контроллера этого диска и, соответственно, обеспечивает более быстрый рост пропускной способности. Первая версия спецификаций SATA предусматривала скорости передачи от 150 Мбайт/с, ранее достижимые только для Ultra SCSI/LVD. Ожидается, что скорость интерфейса будет удваиваться каждые три-четыре года. Кроме того, подключенные к разным шинам диски не обязаны ожидать друг друга, т. е. эта смена конструктива привела к устранению одного из основных недостатков параллельного ATA. Переход к последовательнойшине значительно упростили как разводку проводников по материнской плате, так и разводку кабелей внутри корпуса компьютера. К сожалению, контроллеры и диски SATA первого поколения представляли собой контроллеры и диски ATA-6 с переходниками для нового кабеля. Такие устройства, конечно, облегчали монтаж компьютеров, но не давали сколько-нибудь существенных преимуществ по скорости по сравнению с ATA-6. Ко времени подготовки к печати второго издания книги на рынке уже начали появляться контроллеры и диски

второго поколения, сразу при проектировании рассчитанные на SATA и полностью использующие его преимущества.

Еще одна интересная перспектива — это конвергенция давно конкурирующих стандартов ATA и SCSI; с появлением стандарта SAS (Serially Attached SCSI) такая перспектива приобрела вполне конкретные черты. В настоящее время, диски SAS могут подключаться к контроллерам SATA, но не наоборот.

9.6.1. Производительность жестких дисков

Скорость работы дискового накопителя определяется рядом параметров. Во-первых, любой обмен данными с таким устройством предваряется двумя обязательными задержками: позиционированием блока головок и *ротационной задержкой (rotational delay)*, когда контроллер ждет, пока требуемый сектор подъедет к головке. Кроме того, само чтение сектора происходит не мгновенно, и, наконец, необходимо принимать во внимание скорость передачи данных по шине, к которой подключен диск, и возможные коллизии на этойшине.

Задержку позиционирования головки можно в определенных пределах сокращать облегчением блока головок и совершенствованием механики. Единственный способ сокращения ротационной задержки — это увеличение скорости вращения диска. У современных дисков эта скорость достигает 15 000 об/мин. Скорость вращения повышает также и скорость чтения отдельного сектора. Кроме того, увеличение плотности записи (как за счет улучшения физических характеристик магнитного слоя, так и за счет совершенствования способов модуляции) и обусловленное этим увеличение количества секторов на дорожке также ускоряет считывание каждого отдельного сектора.

Любопытный прием сокращения ротационной задержки при чтении последовательных секторов — это форматирование дисков с *чередованием (interleave)*, когда номера секторов на дорожке не совпадают с их физической последовательностью (рис. 9.38). Количество физических секторов, размещенных между логически последовательными секторами, называется коэффициентом чередования.

Благодаря чередованию, если ЦПУ сформирует команду чтения следующего сектора не сразу после завершения передачи предыдущего (на практике это всегда происходит не мгновенно), не придется ждать следующего оборота диска. Впрочем, если коэффициент чередования окажется слишком большим, ждать подхода следующего сектора все-таки придется. В годы моей молодости подбор оптимальных параметров чередования для жестких дисков был почти столь же популярным развлечением, каким ныне является "разгон" (подбор частот процессора и системной и периферийной шин, которые выше паспортных, но обеспечивают более или менее устойчивую работу системы).

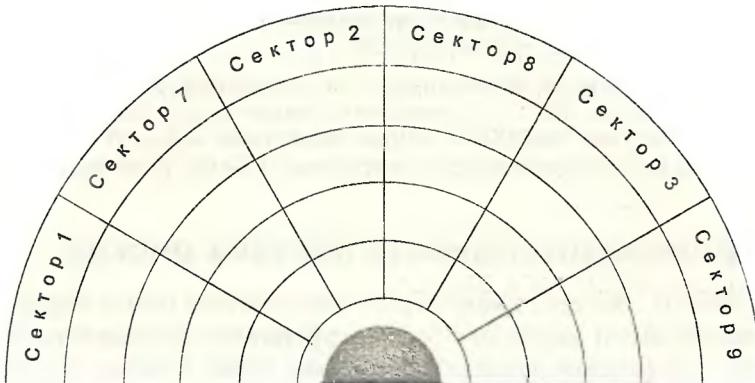


Рис. 9.38. Дорожка диска, отформатированного с чередованием

Ряд файловых систем, разработанных в 80-е годы XX века, реализовали свои собственные схемы чередования, размещая последовательные блоки файлов в чередующиеся физические секторы. Современные дисковые контроллеры предоставляют логику опережающего считывания и отложенной записи, которые снижают потребность в таких приемах. Кроме того, фабричное форматирование жестких дисков обычно сразу осуществляется с оптимальным чередованием.

Для сокращения времени подачи головки нередко используют сортировку запросов по номеру дорожки. В [Дейтел 1987] приводится анализ нескольких алгоритмов такой сортировки. В наше время шире всего используется элеваторная сортировка (в [Дейтел 1987] она называется SCAN). При отсортированных в соответствии с этим алгоритмом запросах, блок головок начинает от внутренней дорожки (или, точнее, от запроса, ближайшего к этой дорожке) и движется к наружной, выполняя последовательные запросы так, чтобы направление движения головки не изменялось (рис. 9.39). Запросы, поступаю-

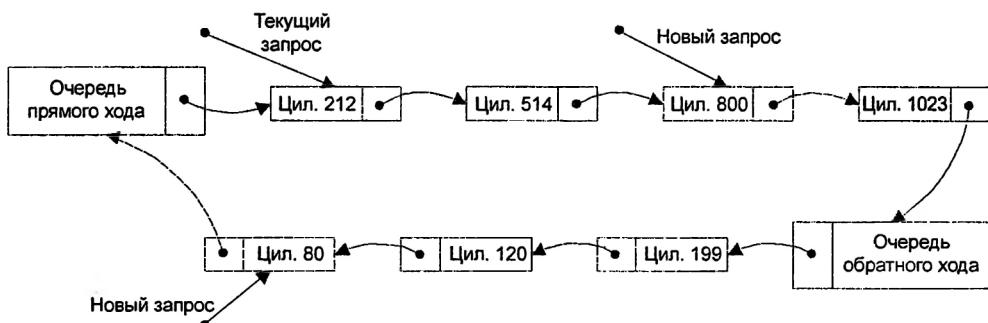


Рис. 9.39. Элеваторная сортировка

щие на уже пройденные дорожки, откладываютя на обратный проход. Достигнув внешней дорожки (или, точнее, самой внешней из дорожек, на которую есть запрос), блок головок меняет направление движения.

Сортировке подвергаются только запросы на чтение. По причинам, которые мы в полной мере поймем в главе 11, запросы на запись практически никогда не переупорядочиваются.

9.6.2. Дисковые массивы

Женщины замечательно умеют хранить секреты.
Они делают это сообща.

Еще один прием оптимизации производительности дисковых накопителей — это объединение нескольких физических дисков в один большой логический диск. При таком объединении некоторые диски могут передавать данные, в то время как другие позиционируют блок головок или ждут подхода нужного сектора к головке. Дисковые массивы выгодны не только с точки зрения производительности, но и повышают единичную емкость запоминающего устройства — это может быть, например, полезно для хранения крупных неделимых объектов, таких как таблицы реляционной СУБД.

Следует учесть, что объединение дисков приводит к резкому снижению надежности массива на отказ: вероятности независимых событий складываются, поэтому вероятность отказа любого из дисков массива равна сумме вероятностей отказа одиночного диска. Для компенсации или устранения этого недостатка данные в дисковых массивах обычно хранятся с избыточностью.

Общее название всех технологий объединения дисков — *RAID (Redundant Array of Inexpensive Disks* — избыточный массив недорогих дисков). Этот термин был предложен в работе [Gibson/Katz/Patterson 1988], в которой проведен анализ различных технологий создания дисковых массивов с точки зрения их производительности и надежности и было рассмотрено пять возможных способов размещения избыточных данных. Предложенная авторами статьи нумерация этих технологий без ссылки на источник используется в самых разнообразных публикациях в форме RAID уровня X. В современных публикациях, например [www.acnc.com], часто упоминаются также дополнительные и комбинированные уровни RAID, в именовании которых общего согласия не достигнуто.

Впрочем, один дополнительный уровень, не упоминавшийся в статье [Gibson/Katz/Patterson 1988], практически везде понимается и именуется одинаково. Под RAID уровня 0 практически единогласно понимают простой *стриппинг* (*stripping* — дословно, разделение на полосы). Стриппинг, строго говоря, не является методом создания избыточных массивов, потому что он

не предполагает избыточности: емкость результирующего логического диска равна сумме объемов физических дисков (рис. 9.40). Из-за этого RAID 0 не обсуждался в работе [Gibson/Katz/Patterson 1988].

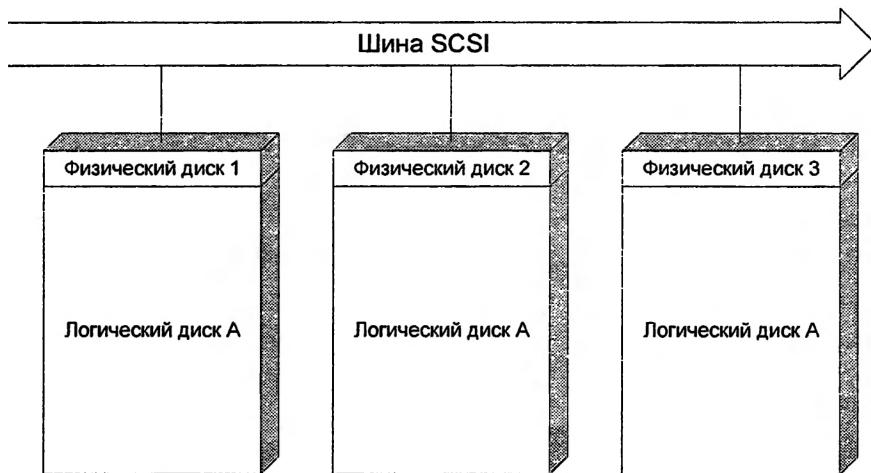


Рис. 9.40. RAID 0 (стриппинг)

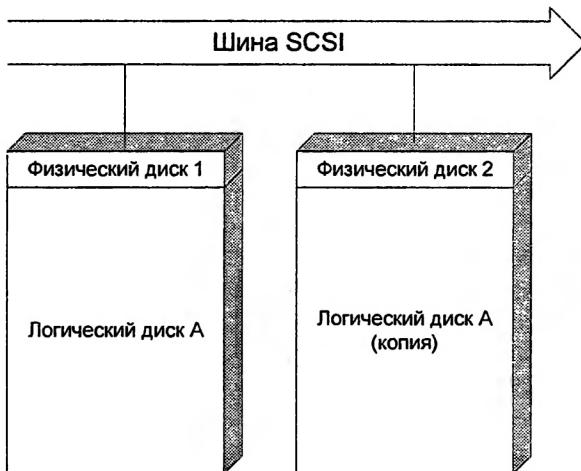


Рис. 9.41. RAID 1 (зеркалирование)

RAID уровня 1 известен также как *зеркалирование (mirroring)*. При зеркалировании на каждый из дисков записывается полная копия данных (рис. 9.41). Обычно в зеркальном режиме используется не более двух дисков. Зеркалирование обеспечивает некоторое падение производительности (обычно все-таки

менее, чем двукратное), но гарантирует работоспособность системы при отказе любого из дисков. Оба эти приема широко применяются на практике, обычно с небольшим количеством дисков, не более двух, реже, трех. Они не требуют значительных вычислительных ресурсов и потому обычно реализуются программно, драйвером диска или файловой системы.

RAID уровня 2 предлагает снабжение блоков данных на дисках кодом Хэмминга (*см. разд. 1.7*) и размещение этого кода на отдельном диске массива. Эта технология имеет смысл, если контроллеры дисков не хранят код восстановления ошибок в собственных заголовках секторов. Поскольку современные диски практически без исключения используют восстанавливающее кодирование и динамическое переназначение дефектных секторов, RAID 2 ныне практически не применяется.

RAID уровня 3 и 4 похожи во многих отношениях. Каждый из этих методов требует не менее трех дисков и подсчета контрольной суммы таким способом, чтобы данные могли быть однозначно восстановлены при полной потере одного из дисков. В RAID уровня 3 контрольные суммы подсчитываются для каждого байта записываемых данных, а в RAID уровня 4 — для групп блоков. Для хранения контрольных сумм отводится отдельный диск.

Эти методики обеспечивают более низкую (хотя и достаточную для большинства практических целей) избыточность, чем зеркалирование, но, как и стриппинг, увеличивают единичную емкость диска и повышают производительность. Для отказа всей системы требуется полный отказ более чем одного диска; многие конструктивы дисковых массивов уровня более 2 предусматривают "горячую" (без выключения системы) замену дисков, что допускает теоретически неограниченную наработку на останов, недостижимую при использовании одиночных дисков.

RAID 3 требует изменения формата секторов диска и соответствующей переделки контроллера и не может применяться с серийными дисковыми приводами. Основным недостатком RAID 4 является необходимость обращения к диску с контрольной суммой при каждой модификации одного из блоков группы. При записи большого количества логически последовательных блоков это не проблема, но может превратиться в проблему при случайно распределенных записях одиночных блоков.

Этого недостатка лишен RAID уровня 5, в котором контрольные суммы распределены по всем дискам массива (рис. 9.42). RAID 5 находит широкое применение в серверах уровня рабочей группы или подразделения. При программной реализации, впрочем, подсчет контрольных сумм требует значительной доли вычислительной мощности ЦПУ, поэтому широкое распространение получили "аппаратные" контроллеры RAID 5, имеющие собственный процессор и, как правило, довольно большой объем собствен-

ной памяти. Это не мешает некоторым ОС, в частности Windows NT/2000/XP, реализовать RAID 5 программно на уровне файловой системы.

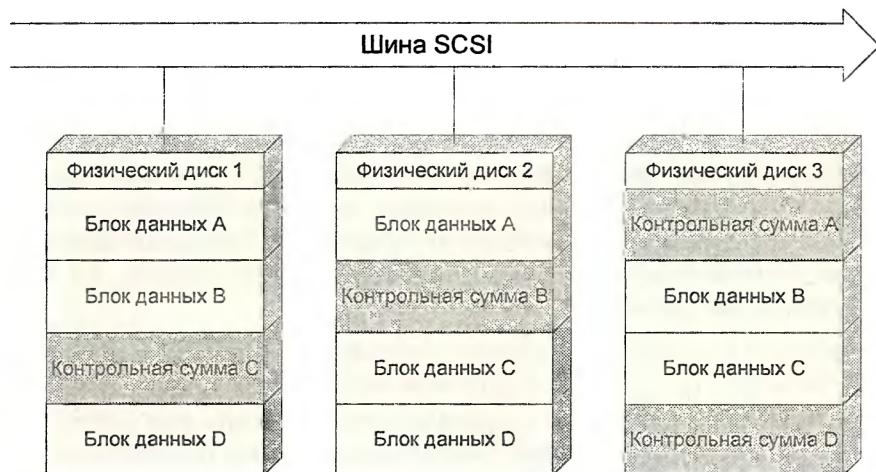


Рис. 9.42. RAID уровня 5

9.6.3. Сети доступа к дискам

А теперь прикинь, солдат, — где Москва, а где Багдад!
Али ты смотался за ночь до Багдаду и назад?

Л. Филатов

Волоконно-оптические каналы подключения дисковых устройств к компьютерам, как, например, упоминавшийся в разд. 9.4 FC-AL, обеспечивают высокую пропускную способность и возможность устанавливать диски на значительном удалении от компьютера, иногда даже в другом здании.

Высокая пропускная способность позволяет одновременно нескольким системам использовать одну и ту же логическую шину. В частности, это дает возможность перераспределить диски или даже их логические разделы между системами без их физического переключения. Распределение вычислительных систем, дисков (особенно зеркальных или включенных в RAID более высокого уровня) и устройств резервного копирования по разным помещениям позволяет снизить риск потери данных в случае пожара или какого-либо другого бедствия и даже обеспечить бесперебойный сервис в этих условиях (рис. 9.43).

Такая механика подключения запоминающих устройств к компьютерам изменяет взгляд на взаимоотношение между вычислительными и запоминающими устройствами: вместо сервера с подключенными к нему одним или несколькими дисками мы имеем массив постоянной памяти, к которому под-

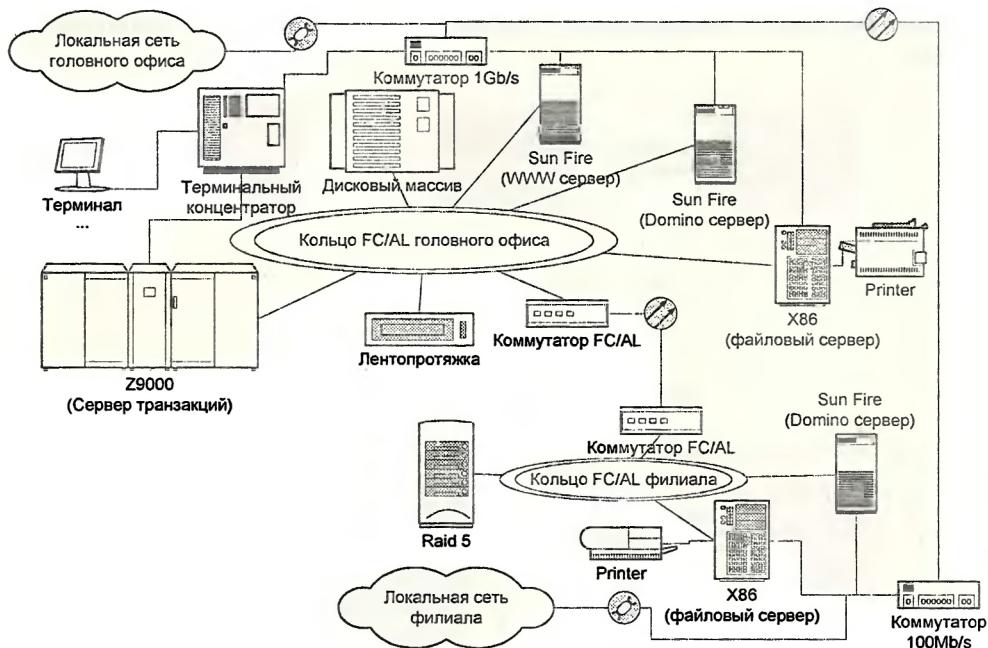


Рис. 9.43. Сеть FC-AL с несколькими кольцами и коммутаторами

ключены один или несколько серверов, осуществляющих поиск и транзакции над хранящимися в массиве данными. Такой взгляд на организацию внешней памяти называется *SAN (Storage Access Network)* — сеть доступа к запоминающим устройствам (рис. 9.44).

Не следует путать SAN с файловыми серверами, серверами транзакций и СУБД. Если последние предоставляют относительно высокоуровневые операции над хранимыми данными (поиск файлов в каталогах, транзакции над таблицами реляционной СУБД и т. д.), то SAN обеспечивает доступ к разделяемым устройствам на уровне команд *SCSI* — чтения и записи отдельных блоков данных.

Средства для разрешения возможных коллизий, возникающих при этом, весьма ограничены: *SCSI III* предусматривает лишь простые средства взаимоисключения при доступе нескольких инициаторов к одному целевому устройству, сводящиеся к захвату или освобождению целого логического устройства. Это может быть приемлемо лишь для разделения ресурсов, относительно редко используемых различными серверами (например, устройств резервного копирования) или применяемых преимущественно для чтения (например, загрузочных дисков). Как мы видели в разд. 7.3.4, для разрешения коллизий при интенсивной асинхронной модификации данных предпочтительнее архитектура сервера транзакций или мониторного процесса, для реа-

лизации которой у современных устройств SCSI явно недостаточно интеллекта.

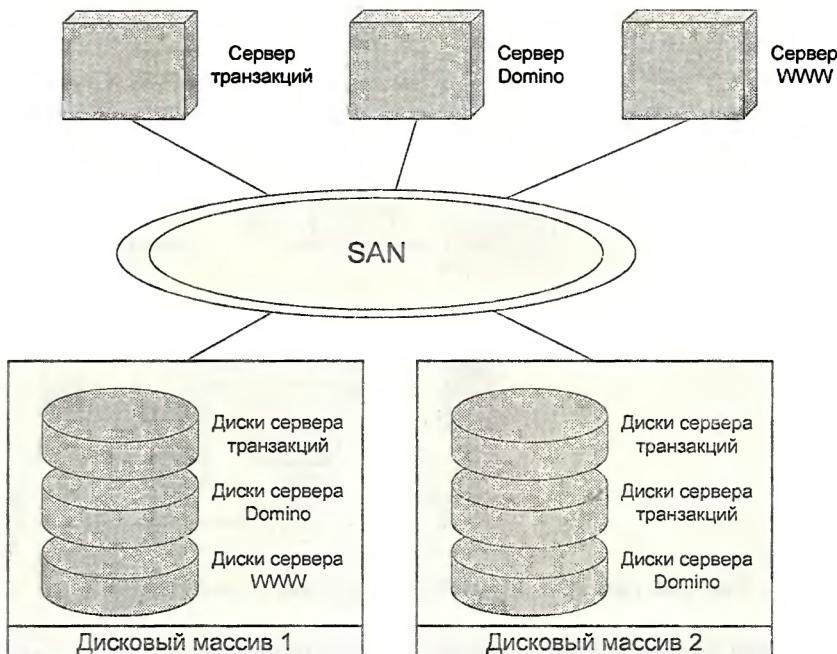


Рис. 9.44. Сеть доступа к дискам (SAN)

Впрочем, SAN представляет собой молодую технологию с неясными на момент написания книги перспективами развития. Повышение интеллекта запоминающих устройств и превращение их в серверы транзакций является одним из перспективных направлений. Второе возможное направление развития, не противоречащее первому, — конвергенция транспортных протоколов SAN с протоколами гигабитных локальных и даже глобальных сетей, например путем инкапсуляции команд и блоков данных SCSI в кадры Ethernet или даже в пакеты TCP/IP. Перспективы такой конвергенции обсуждаются (правда, в пессимистическом ключе) в работе [Neil 2000].

В настоящее время стандартизован протокол iSCSI (RFC 3270), обеспечивающий передачу команд SCSI через TCP/IP. Существуют также нестандартные протоколы HyperSCSI и AoE (ATA over Ethernet), обеспечивающие передачу команд SCSI и ATA через Ethernet. Передача через TCP/IP позволяет использовать маршрутизуемые сети сложной топологии, но сопровождается значительными накладными расходами, главным образом с точки зрения оконечных точек (необходимы значительные затраты процессорного времени для разбора пакетов TCP и сборки сообщений). Из-за этого некоторые реали-

зации iSCSI используют специальные периферийные процессоры, реализующие стек протоколов TCP/IP.

Протоколы HyperSCSI и AoE лишены этого недостатка, но обеспечивают существенно меньшую гибкость.

Вопросы для самопроверки

1. Приведите примеры устройств ввода, устройств вывода и устройств ввода/вывода.
2. Почему запоминающие устройства целесообразно выделить в отдельный класс, а не считать частным случаем устройств ввода/вывода?
3. Назовите преимущества и недостатки отдельного адресного пространства ввода/вывода и отраженного на память ввода/вывода.
4. Перечислите основные преимущества и недостатки фиксированной и географической адресации устройств. Подумайте, как можно решать проблемы, возникающие при подключении к системе двух однотипных устройств с фиксированной адресацией.
5. Почему конструктив USB называют шиной, несмотря на то, что он использует звездообразную топологию подключения устройств?
6. Какое максимальное количество оконечных устройств можно подключить к шине USB при использовании восьмипортовых хабов? Шестнадцатипортовых хабов?
7. Какие преимущества дает использование специальных периферийных шин, по сравнению с подключением устройств прямо к системной шине?
8. Какие периферийные шины используются в вашем домашнем компьютере?
9. Чем обусловлены ограничения, допускающие не более 8 устройств нашине SCSI II и 16 — нашине Wide SCSI? Почему количество устройств оказалось связано с разрядностью шины?
10. Простые подсчеты показывают, что для видеобуфера объемом 1248×1024 пикселов достаточно всего нескольких мегабайт ОЗУ; даже для обеспечения разрешения 2048×1536 можно обойтись 12 с небольшим мегабайтами. Для чего видеоадAPTERЫ современных персональных компьютеров используют 256 и даже 512 Мбайт ОЗУ?
11. Какие преимущества дает обеспечение совместимости по регистрам и командам со старыми моделями устройств?
12. Какие выгоды дает объединение дисковых устройств в дисковые массивы? Каковы недостатки такого объединения? Почему при создании дисковых массивов такое большое внимание уделяется обеспечению избыточности и надежности хранения данных?

ГЛАВА 10



Драйверы внешних устройств

Когда я на почте служил ямщиком,
Ко мне поступался косматый геолог.

И глядя на карту на белой стене,
Он усмехнулся мне.

Г. Самойлов

Драйвер (driver) представляет собой специализированный программный модуль, управляющий внешним устройством. Слово driver происходит от глагола *to drive* (вести) и переводится с английского языка как извозчик или шофер: тот, кто ведет транспортное средство. Драйверы обеспечивают единый интерфейс для доступа к различным устройствам, тем самым устранивая зависимость пользовательских программ и ядра ОС от особенностей аппаратуры.

Драйвер не обязательно должен управлять каким-либо физическим устройством. Многие ОС предоставляют также драйверы виртуальных устройств или *псевдоустройств* — объектов, которые ведут себя аналогично устройству ввода/вывода, но не соответствуют никакому физическому устройству.

В виде псевдоустройств реализуются трубы в системах семейства Unix и почтовые ящики в VMS. Еще одним примером полезного псевдоустройства являются устройства */dev/null* в Unix и аналогичное им NUL в MS DOS\Windows\OS/2. В современных системах семейства Unix в виде псевдоустройств, размещенных в псевдофайловой системе */proc*, реализован доступ к большинству параметров системы — адресным пространствам активных процессов, статистике и параметрам настройки ядра, данным отдельных подсистем, например таблице маршрутизации сетевого протокола IP.

Прикладные программы, использующие собственные драйверы, не так уж редки — примерами таких программ могут быть GhostScript (свободно распространяемый интерпретатор языка PostScript, способный выводить программы на этом языке на различные устройства, как печатающие, так и

экранные) или L_AT_EX, который также способен печатать на самых разнообразных устройствах. Однако эта глава посвящена преимущественно драйверам, используемым ядром ОС.

Большинство ОС общего назначения запрещают пользовательским программам непосредственный доступ к аппаратуре. Это делается для повышения надежности и обеспечения безопасности в многопользовательских системах. В таких системах драйверы являются для прикладных программ единственным способом доступа к внешнему миру.

Еще одна важная функция драйвера — это взаимоисключение доступа к устройству в средах с вытесняющей многозадачностью. Допускать одновременный неконтролируемый доступ к устройству нескольких параллельно исполняющихся процессов просто нельзя, потому что для большинства внешних устройств даже простейшие операции ввода/вывода не являются атомарными.

Например, в большинстве аппаратных реализаций последовательного порта RS232 передача байта состоит из четырех шагов: записи значения в регистр данных, записи команды "передавать" в регистр команды, ожидания прерывания по концу передачи и проверки успешности передачи путем считывания статусного регистра устройства. Нарушение последовательности шагов может приводить к неприятным последствиям — например, перезапись регистра данных после подачи команды, но до завершения передачи, может привести к остановке передачи или, что еще хуже, передаче искаженных данных и т. д.

Нельзя также забывать о неприятностях более высокого уровня — например, смешивании вывода разных процессов на печати или данных — на устройстве внешней памяти. Поэтому оказывается необходимо связать с каждым внешним устройством какой-то разграничитель доступа во времени. В современных ОС эта функция возлагается именно на драйвер. Обычно одна из нитей драйвера представляет собой процесс-монитор, выполняющий асинхронно поступающие запросы на доступ к устройству. В Unix, OS/2 и Windows NT/2000/XP этот процесс называется стратегической функцией. Подробнее этот механизм обсуждается в разд. 10.5.2 и 10.7.

При определении интерфейса драйвера разработчики ОС должны найти правильный баланс между противоречивыми требованиями:

- стремлением как можно сильнее упростить драйвер, чтобы облегчить его разработку и (косвенно) уменьшить вероятность опасных ошибок;
- желанием предоставить гибкий и интеллектуальный интерфейс к разнообразным устройствам.

Драйверы обычно разрабатываются не поставщиками операционной системы,

а сторонними фирмами — разработчиками и изготовителями периферийного оборудования. Поэтому интерфейс драйвера является ничуть не менее внешним, чем то, что обычно считается внешним интерфейсом ОС — интерфейс системных вызовов. Соответственно, к нему предъявляются те же требования, что и к любому другому внешнему интерфейсу: он должен быть умопостижимым, исчерпывающе документированным и стабильным — не меняться непредсказуемо от одной версии ОС к другой. Идеальным вариантом была бы полная совместимость драйверов хотя бы снизу вверх, чтобы драйвер предыдущей версии ОС мог использоваться со всеми последующими версиями.

Потеря совместимости в данном случае означает, что все независимые изготовители оборудования должны будут обновить свои драйверы. Организация такого обновления оказывается сложной, неблагодарной и часто попросту невыполнимой задачей — например, потому, что изготовитель оборудования уже не существует как организация или отказался от поддержки данного устройства.

Отказ от совместимости драйверов на практике означает "брошенное" периферийное оборудование и, как следствие, "брошенных" пользователей, которые оказываются вынужденны либо отказываться от установки новой системы, либо заменять оборудование. Оба варианта, естественно, не улучшают отношения пользователей к поставщику ОС, поэтому многие поставщики просто не могут позволить себе переделку подсистемы ввода/вывода. Таким образом, интерфейс драйвера часто оказывается наиболее консервативной частью ОС.

Подсистема ввода/вывода OS/2

В качестве примера такого консерватизма можно привести подсистему ввода/вывода OS/2. Совместный проект фирм IBM и Microsoft — OS/2 1.x разрабатывалась как операционная система для семейства персональных компьютеров Personal System/2. Младшие модели семейства были основаны на 16-разрядном процессоре 80286, поэтому вся ОС была полностью 16-битной (отличия "16-разрядного" кода процессора 80286 от "32-разрядного" кода x86 упоминались в разд. 5.1 и подробно описываются в *приложении 2*).

Позднее разработчики фирмы IBM реализовали 32-битную OS/2 2.0, но для совместимости со старыми драйверами им пришлось сохранить 16-битную подсистему ввода/вывода. Все точки входа драйверов должны находиться в 16-битных ('USE16') сегментах кода; драйверам передаются только 16-разрядные ('far') указатели и т. д.

По утверждению фирмы IBM, они рассматривали возможность реализации также и 32-битных драйверов, но их измерения не показали значительного повышения производительности при переходе к 32-битной модели. Позднее, впрочем,

чем, суперскалярные процессоры архитектуры x86, оптимизированные для исполнения 32-разрядного кода, все-таки продемонстрировали значительный рост производительности для такого кода по сравнению с 16-разрядным, и, начиная с OS/2 4.0, 32-битный интерфейс для драйверов и других подсистем ядра все-таки предоставляется, но без отказа от поддержки старых, 16-разрядных драйверов.

Благодаря этому сохраняется возможность использовать драйверы, разработанные еще в конце 80-х годов и рассчитанные на OS/2 1.x. Эта возможность оказывается особенно полезной при работе со старым оборудованием.

Напротив, разработчики фирмы Microsoft отказались от совместимости с 16-битными драйверами OS/2 1.x в создававшейся ими 32-битной версии OS/2, называвшейся OS/2 New Technology. Фольклор утверждает, что именно это техническое решение оказалось причиной разрыва партнерских отношений между Microsoft и IBM, в результате которого OS/2 NT вышла на рынок под названием Windows NT 3.1. Эта история подробнее излагается в [разд. П.1.4.2—П.1.4.4](#).

Подсистема ввода/вывода Windows 9x/ME

Сама фирма Microsoft, впрочем, демонстрирует почти столь же трогательную приверженность к совместимости со старыми драйверами: системы линии Windows 95/98/ME до сих пор используют весьма своеобразную архитектуру, основной смысл которой — возможность применять, пусть и с ограничениями, драйверы MS DOS.

Windows 9x и Windows 3.x в enhanced-режиме предоставляют вытесняющую многозадачность для VDM (Virtual DOS Machine — виртуальная машина [для] DOS), однако сами используют DOS для обращения к дискам и дискетам. Ядро однозадачной DOS не умеет отдавать управление другим процессам во время исполнения запросов ввода/вывода. В результате во время обращения к диску все остальные задачи оказываются заблокированы.

У современных РС время исполнения операций над жестким диском измеряется десятыми долями секунды, поэтому фоновые обращения к жесткому диску почти не приводят к нарушениям работы остальных программ. Однако скорость работы гибких дисков осталась достаточно низкой, следовательно, работа с ними в фоновом режиме блокирует систему на очень заметные промежутки времени.

Эффектная и убедительная демонстрация этой проблемы очень проста: достаточно запустить в фоновом режиме форматирование дискеты или просто комманду `COPY C:\TMP*.* A:`, если в каталоге C:\TMP достаточно много данных. При этом работать с системой будет практически невозможно: во время обращений к дискете даже курсор мыши не будет отслеживать ее движений, будут теряться нажатия клавиш и т. д.

Windows 95/98/ME использует несколько методов обхода DOS при обращениях к диску, поэтому пользователи этой системы не всегда сталкиваются с описанной проблемой. Однако при использовании блочных драйверов реального режима система по-прежнему применяет DOS в качестве подсистемы ввода/вывода, и работа с дискетами в фоновых задачах также нарушает работу задач первого плана.

Если говорить о совместимости, то со многих точек зрения очень привлекательной представляется идея универсального драйвера — модуля, который без изменений или с минимальными изменениями может использоваться для управления одним и тем же устройством в различных ОС. В частности, это позволило бы прошивать драйверы в установленное на плате устройства ПЗУ. Это могло бы резко упростить установку новых устройств в систему. Если устройство может подключаться к вычислительным системам с различными центральными процессорами (например, периферийные шины Q-Bus и Unibus компании Dec использовались в компьютерах PDP-11 и VAX, шина ISA — в PC-совместимых компьютерах 8086, 80286 и x86, PCI — в компьютерах x86, PowerPC, PowerMAC, SPARC и некоторых других, PCMCIA — в тех же, что PCI, и в ряде карманных ПК), плата могла бы содержать несколько разных ПЗУ. Впрочем, даже если плата содержит ПЗУ только для одной архитектуры центрального процессора, это не исключало бы поставки "нормальных" драйверов для остальных компьютеров, так что поставщик устройств с таким ПЗУ упростил бы жизнь пользователям одной (скорее всего, наиболее распространенной) аппаратной архитектуры, но пользователи остальных аппаратных платформ ничего бы не потеряли.

В 80-е годы было несколько относительно успешных примеров таких конструктивов. Наиболее известен конструктив NuBus, использовавшийся в компьютерах Apple Macintosh. Устройства этого конструктива имели ПЗУ, содержащее драйвер устройства для Mac OS. Версия Unix для Mac (AUX) также могла использовать эти драйверы.

В микрокомпьютерах Yamaha (в 80-е годы они в довольно большом количестве закупались СССР для использования в школах и вузах, поэтому многие программисты старшего поколения в той или иной степени знакомы с ними) использовалась похожая схема; в ПЗУ плат расширения для этих компьютеров прошивались драйверы для специализированной версии *CP/M*.

Устройства с интерфейсами ISA и PCI, предназначенные для работы в PC-совместимых компьютерах, также могут содержать ПЗУ, содержащее обработчик Int 13h (функции BIOS, которая обеспечивает работу с дисковыми устройствами). На платах ISA стартовый адрес этого ПЗУ можно было задавать перемычками так, чтобы ПЗУ различных плат не конфликтовали. Нашине PCI адрес ПЗУ может задаваться средствами автоконфигурации этой шины. При включении системы BIOS материнской платы сканирует допустимые стартовые адреса (их диапазон довольно узок) и, найдя по какому-то из адресов программу, запускает ее, таким образом "подключая" BIOS платы и инициализируя устройство.

Если BIOS платы дискового контроллера или НВА содержит обработчик Int 13h, то устройства, подключенные через эту плату (например, жесткие диски

с интерфейсом SCSI), могут затем использоваться для загрузки ОС. Далее, если ОС сама использует сервисы Int 13h (как это делает DOS), она может обойтись без каких-либо дополнительных драйверов.

К сожалению, — несмотря даже на то, что, как мы увидим далее, в общих чертах архитектура драйвера в большинстве современных ОС удивительно похожа, — идея эта, вероятно, нереализуема. Даже для близкородственных ОС — например, систем семейства Unix, — драйверы одного и того же устройства не всегда могут быть легко перенесены из одной ОС в другую, не говоря уж о возможности использования без модификаций. В частности, поэтому все решения, использующие ПЗУ на плате устройства, имели успех только на компьютерах, на которых поддерживалась только одна ОС, или, во всяком случае, наблюдалось резко выраженное доминирование одной ОС.

На PC-совместимых компьютерах это усугублялось тем, что BIOS работал в "реальном" режиме процессора 80286/x86 (в режиме эмуляции 8086), и его использование из ОС, работавших в "защищенном" режиме (т. е. в основных режимах 80286 и x86), было сопряжено со значительными трудностями и большими накладными расходами на создание виртуальной сессии 8086 и сложные механизмы обмена данными с программами, исполняющимися в этой сессии. Одна из немногих ОС защищенного режима, которая могла работать через BIOS, — это IBM OS/2, в поставке которой есть драйвер IBMINT13.SYS, позволяющий работать с дисковыми устройствами через сервисы прерывания 13h. Впрочем, производительность этого драйвера такова, что IBM рекомендует его использование только в крайних случаях, когда не удается найти или заставить работать "нормальные" драйверы для соответствующего дискового контроллера.

Один из немногих относительно удачных примеров единого интерфейса драйверов, используемых в ОС с различными архитектурами ядра — это WDM (Windows Driver Model — модель драйвера Windows) компании Microsoft, который поддерживается как Windows 98/ME, так и Windows 2000/XP/2003, но не более ранними версиями обоих ОС — Windows 95 и NT 4.0. При этом обе линии систем, в основном, сохраняют также совместимость с драйверами к более ранним версиям системы внутри линии — большинство драйверов для DOS и Windows 95 пригодны для 98/ME, а модули ядра Windows NT 4.0 и даже 3.51 работают под 2000/XP (впрочем, такие модули обычно необходимо регистрировать вручную, потому что штатные инсталляционные программы проверяют версию ОС и отказываются работать под новыми версиями системы).

Проблема здесь в том, что интерфейс между драйвером и ядром ОС всегда двусторонний: не только прикладные программы и ядро вызывают функции драйвера, но и, наоборот, драйвер должен вызывать функции ядра. Структура

интерфейсов ядра, доступных драйверу, определяет многие аспекты архитектуры ОС в целом.

В предыдущих главах мы уже обсуждали многие из ключевых вопросов: способ сборки ядра, стратегию управления памятью, способы обмена данными между ядром и пользовательскими процессами, и, наконец, механизмы межпоточного взаимодействия — между нитями самого драйвера (ниже мы увидим, что подавляющее большинство драйверов состоит как минимум из двух нитей), между драйвером и остальными нитями ядра и между драйвером и нитями пользовательских процессов.

Изложение решений перечисленных проблем составляет если и не полное описание архитектуры ОС, то, во всяком случае, значительную его часть. Так, переход от однозадачной системы или кооперативной многозадачности к вытесняющей многозадачности может потребовать не только изменения планировщика, но и радикальной переделки всей подсистемы ввода/вывода, в том числе и самих драйверов.

Таким образом, до тех пор, пока используются ОС различной архитектуры, разработка универсального интерфейса драйвера, если теоретически и возможна, то практически вряд ли осуществима.

10.1. Функции драйверов

Прежде всего, драйвер должен иметь функции, вызываемые ядром при загрузке и выгрузке модуля и при подключении модуля к конкретным устройствам. Например, в Sun Solaris это перечисленные далее функции:

- `int _init(void)` — инициализация драйвера. Эта функция вызывается при загрузке модуля. Драйвер должен зарезервировать все необходимые ему системные ресурсы и проинициализировать собственные глобальные переменные. Инициализация устройства на этом этапе не происходит;
- `int probe(dev_info_t *dip)` — проверить наличие устройства в системе. Во многих системах эта функция реализуется не самим драйвером, а специальным модулем-“снiffeром” (*sniffer* — дословно, “нюхач”), используемым программой автоконфигурации;
- `int attach(dev_info_t * dip, ddi_attach_cmd_t cmd)` — инициализация копии драйвера, управляемой конкретным устройством. Этую функцию можно рассматривать как аналог конструктора объекта в объектно-ориентированном программировании. Если в системе присутствует несколько устройств, управляемых одним драйвером, некоторые ОС загружают несколько копий кода драйвера, но в системах семейства Unix функция `attach` просто вызывается многократно.

Каждая из инициализированных копий драйвера имеет собственный блок локальных переменных, в которых хранятся переменные состояния устройства. При вызове `attach` драйвер должен прочитать конфигурационный файл, где записаны параметры устройства (номенклатура этих параметров зависит от устройства и от драйвера), разместить и проинициализировать блок переменных состояния, зарегистрировать обработчики прерываний, проинициализировать само устройство и, наконец, зарегистрировать устройство как доступное для пользовательских программ, создав для него *минорную запись* (*minor node*). В ряде случаев драйвер создает для одного устройства несколько таких записей.

Например, каждый жесткий диск в Unix SVR4 должен иметь 16 записей — по две (далее мы поймем, для чего они нужны) для каждого из восьми допустимых слайсов (логических разделов, см. разд. 3.11) диска. Другой пример: в большинстве систем семейства Unix лентопротяжные устройства имеют две минорные записи. Одно из этих устройств при открытии перематывает ленту к началу, другое не перематывает. В действительности оба устройства управляются одним и тем же драйвером, который определяет текущий режим работы в зависимости от указанной минорной записи.

Современные Unix-системы, в частности Solaris, используют отложенную инициализацию, когда для многих устройств `attach` вызывается только при первой попытке доступа пользовательской программы к устройству.

- `int detach(dev_info_t *dip, ddi_detach_cmd_t cmd)` — аналог деструктора объекта в ООП. Впрочем, в отличие от деструктора, эта операция не безусловна — если не удается нормально завершить обрабатываемые в данный момент операции над устройством, драйвер может и даже обязан отказаться деинициализироваться. При деинициализации драйвер должен освободить все системные ресурсы, которые он занял при инициализации и в процессе работы (в том числе и уничтожить минорную запись) и может, если это необходимо, произвести какие-то операции над устройством, например, выключить приемопередатчик, запарковать головки чтения-записи и т. д. После того как все устройства, управляемые драйвером, успешно деинициализированы, система может его выгрузить.
- `int _fini(void)` — функция, вызываемая системой перед выгрузкой модуля. Драйвер обязан освободить все ресурсы, которые он занял на этапе инициализации модуля, а также все ресурсы, занятые им во время работы на уровне модуля (не привязанные к конкретному управляемому устройству).

После того как драйвер проинициализировался и зарегистрировал минорную запись, пользовательские программы могут начинать обращаться к нему и к управляемым им устройствам. Понятно, что обеспечить единый интерфейс к

разнообразным категориям устройств, перечисленным в главе 9, по меньшей мере, сложно. Наиболее радикально подошли к этой проблеме разработчики системы UNIX, разделивши все устройства на два класса: *блочные* (block device, высокоскоростные устройства памяти с произвольным доступом, в первую очередь, дисковые устройства) и *последовательные* или *символьные* устройства (character device, все остальное). В действительности, у современных систем семейства Unix типов драйверов несколько больше, но об этом далее.

Над последовательными устройствами определен следующий набор операций, которые могут осуществляться прикладной программой (в простых случаях эти операции непосредственно транслируются в вызовы функций драйвера):

- `int open(char * fname, int flags, mode_t mode)` — процедура открытия устройства. В некоторых случаях она может содержать и дополнительные шаги инициализации устройства — например, для лентопротяжек эта процедура может включать в себя перемотку ленты к началу. Функция возвращает целочисленный идентификатор-“ручку” (handle), часто называемый также дескриптором файла, который используется программой при всех последующих обращениях к устройству;
- `int read(int handle, char * where, size_t how_much)` — чтение данных с устройства. Если устройство приспособлено только для вывода (например, принтер), эта функция может быть не определена;
- `int write(int handle, char * what, size_t how_much)` — запись данных на устройство. Если устройство приспособлено только для ввода, (например, перфоленточный ввод или мышь), эта функция также может быть не определена;
- `void close(int handle)` — процедура закрытия (освобождения) устройства;
- `int ioctl(int handle, int cmd, ...)` — процедура задания специальной команды, которая не может быть сведена к операциям чтения и записи. Набор таких команд зависит от устройства. Например, для растровых графических устройств могут быть определены операции установки видеорежима; для последовательных портов RS232 это могут быть команды установки скорости, количества битов, обработки бита четности и т. д., для дисководов — команды форматирования носителя;
- `off_t lseek(int handle, off_t offset, int whence)`, `long seek` — команда перемещения головки чтения/записи к заданной позиции. Драйверы устройств, не являющихся устройствами памяти, например модема или принтера, как правило, не поддерживают эту функцию.

Слово `long` в названии функции появилось по историческим причинам: в версиях Unix для 16-разрядных машин индекс позиции не мог обозначаться словом, потому что это ограничивало бы логическую длину устройства недопустимо малым значением 65 384 байт. Поэтому необходимо было использовать двойное слово, что соответствовало типу `long` языка C. Современные системы используют 64-разрядный `off_t`;

- `caddr_t mmap(caddr_t addr, size_t len, int prot, int flags, int handle, off_t offset)` memory map — отображение устройства в адресное пространство процесса. Параметр `prot` задает права доступа к отображеному участку: на чтение, на запись и на исполнение. Отображение может происходить на заданный виртуальный адрес, или же система может выбирать адрес для отображения сама.

Функция `mmap` отсутствовала в старых версиях системы, но большинство современных систем семейства (BSD 4.4, ряд наследников BSD 4.3, SVR4 и Linux) поддерживают ее.

Речь идет об отображении в память данных, хранящихся на устройстве. Для устройств ввода/вывода, например, для принтера или терминала, эту функцию невозможно реализовать разумным образом. Напротив, для лент и других последовательных устройств памяти, поддерживающих функцию `lseek`, отображение может быть реализовано с использованием аппаратных средств виртуализации памяти и операций `read` и `write`. Необходимость специальной функции отображения появляется у драйверов устройств, использующих большие объемы памяти, отображенной в адресное пространство системной шины, например, для растровых видеoadаптеров, некоторых звуковых устройств или страниц общой памяти (`backplane memory` — двухпортовой памяти, используемой как высокоскоростной канал обмена данными в много-процессорных системах).

Механизм отображения доступных прикладной программе системных вызовов в функции драйвера относительно сложен. Этот механизм должен включать в себя следующее:

- изменение способа идентификации устройства. "Ручка" представляет собой специфичный для пользовательского процесса номер, в то время как к драйверу могут обращаться разные процессы. В системах семейства Unix для идентификации устройства используется упомянутая ранее минорная запись, которая должна содержать указатель на блок переменных состояния устройства;
- передачу или отображение данных из пользовательского адресного пространства в системное и обратно;

- взаимодействие потоков пользовательского процесса (а в общем случае — нескольких пользовательских процессов, одновременно использующих устройство) с потоками драйвера.

Способы, которыми эти вопросы решаются в современных операционных системах, обсуждаются в последующих разделах. А пока что мы подробнее обсудим, какие именно операции над устройством следует определить и почтому.

Видно, что предлагаемый системами семейства Unix набор операций рассматривает устройство как неструктурированный поток байтов (или, для устройств ввода/вывода, два разнонаправленных потока — для ввода и для вывода). Такое рассмотрение естественно для устройств алфавитно-цифрового ввода/вывода и простых запоминающих устройств, например магнитных лент, однако далеко не столь естественно для более сложных устройств.

Стандартный ответ Unix-культуры в этом случае таков: любая, сколь угодно сложная структура данных может быть *сериализована* — преобразована в последовательный поток байтов. Например, изображение может быть превращено в последовательный поток байтов в виде растровой битовой карты или последовательности описаний графических примитивов — линий, прямоугольников и пр. Примерами такой сериализации для изображений могут являться язык PostScript [partners.adobe.com] и протокол распределенной оконной системы X Window [www.x.org] (оба протокола поддерживают как растровые образы, так и довольно богатые наборы векторных примитивов). В настоящее время разработаны и находят все более широкое применение универсальные самодокументированные форматы данных, позволяющие сериализовать произвольные структуры данных — XML и ISO ASN.1; впрочем, эти форматы обычно используются для общения между пользовательскими программами (например, компонентами многоуровневого распределенного приложения), но не для взаимодействия прикладной программы с устройством.

Нередки, впрочем, ситуации, когда нам интересна не только структура поступающих данных, но и время их поступления (в предыдущей главе мы предложили классифицировать устройства, которые могут быть использованы подобным образом, как генераторы событий) — это бывает в приложениях реального времени, а также в задачах, которые сейчас стало модно называть "задачами мягкого реального времени" — мультимедийных программах, генерирующих поток звука, синхронизованного с изображением, и, особенно, в компьютерных играх.

Для работы с таким устройством прикладная программа, так или иначе, должна зарегистрировать обработчик поступающих от устройства событий. В системах Unix такая регистрация состоит в открытии устройства для чтения, а ожидание события заключается в выполнении над этим устройством

операции чтения (или, если программа ожидает событий от нескольких устройств и/или средств межпроцессного взаимодействия, системных вызовов `poll/select`). Для последовательных устройств ввода операция чтения разблокируется, когда с устройства поступят хоть какие-то данные (а не тогда, когда будет заполнен весь буфер), поэтому, если пришло только одно событие, мы его не пропустим. Драйверы многих устройств, способных работать в качестве генераторов событий, имеют команды `ioctl`, позволяющие более тонко управлять условием разблокирования функции `read`.

Другие ОС предоставляют для работы с устройствами-генераторами событий более сложные механизмы, зачастую основанные на *callback* (дословно — "вызов назад"; механизм взаимодействия подсистем, когда подсистема, запрашивающая сервис, передает обслуживающей подсистеме указатель на функцию, которую необходимо вызвать при наступлении определенного события).

Работа с генераторами событий требует решения еще одной задачи — хранения поступающих событий в периоды, когда пользовательская программа их не успевает обрабатывать. Необходимость относительно сложных схем работы с требуемыми для этого буферами вынудила разработчиков Unix System V Release 3 ввести еще один тип драйверов — *потоковые* (STREAMS) [docs.sun.com 805-7478-10]. Для прикладной программы потоковый драйвер не отличается от обычного символьного устройства, но отличий с точки зрения системы довольно много. Некоторые из этих отличий будут рассматриваться далее.

Unix System V Release 3 (SCO OpenDesktop, SCO OpenServer), Release (SCO UnixWare, SGI Irix, Sun Solaris) и системы, испытавшие влияние OSF Unix (IBM AIX, HP/UX), используют потоковые драйверы для реализации таких важных псевдоустройств, как трубы и сокеты TCP/IP. Кроме того, потоковыми в этих системах являются драйверы сетевых адаптеров и терминальных устройств.

С другой стороны, в OS/2 и Windows NT/2000/XP существуют обширные номенклатуры типов драйверов с различными наборами функций. Так, в OS/2 используются драйверы физических устройств следующих типов:

- простые драйверы последовательных устройств ввода/вывода, аналогичные драйверам символьных устройств в Unix;
- драйверы запоминающих устройств прямого доступа, аналогичные драйверам блочных устройств в Unix;
- драйверы видеоадаптеров, используемые графической оконной системой Presentation Manager (PM);

- драйверы позиционных устройств ввода (мышей и др.), также используемые РМ;
- драйверы принтеров и других устройств вывода твердой копии;
- драйверы звуковых устройств, используемые подсистемой "мультимедиа" MMOS/2;
- драйверы сетевых адаптеров стандарта NDIS, используемые сетевым программным обеспечением фирм IBM и Microsoft;
- драйверы сетевых адаптеров стандарта ODI, используемые программным обеспечением фирмы Novell;
- DMD (Device Manager Driver — драйвер-менеджер класса устройств) (в разд. 10.2 мы подробнее разберемся с назначением драйверов этого типа);
- различного рода "фильтры", например, ODINSUP.SYS — преобразователь ODI-интерфейса в NDIS.

10.2. Многоуровневые драйверы

Массивное тело Сабляк-Паши выглядело необычно, словно под кожей у него была одежда, а на голове, под скальпом, тюрбан.

М. Павич

Нередка ситуация, когда драйвер не может осуществлять управление устройством полностью самостоятельно. Как пример можно рассмотреть драйвер лентопротяжного устройства конструктива SCSI. Для выполнения любой операции над устройством такой драйвер должен сформировать команду или последовательность команд SCSI, передать ее устройству, дождаться ответа и проанализировать его.

Проблема здесь в том, что передача команды и получение ответа происходят с помощью еще одного устройства, НВА. НВА могут быть разнотипными и нуждаются в собственных драйверах. Дополнительная проблема состоит в том, что к одному НВА скорее всего подключены несколько устройств, и наверняка большинство из них не является лентопротяжками, поэтому целесообразно выделение драйвера НВА в отдельный модуль и отдельную мониторную нить.

В системах семейства Unix драйверы целевых устройств SCSI представляют собой обычные символьные или блочные драйверы, с тем лишь отличием, что они не управляют своими устройствами непосредственно, а формируют команды драйверу НВА (рис. 10.1). Команда содержит собственно команду

протокола SCSI и указатели на буферы, откуда следует взять и куда следует положить данные, передачей которых должна сопровождаться обработка команды. Передав команду драйверу HBA, драйвер целевого устройства может либо дождаться ее завершения, либо заняться чем-то другим, например обработкой следующего запроса. Во втором случае, после завершения операции HBA, будет вызван предоставленный целевым драйвером callback.

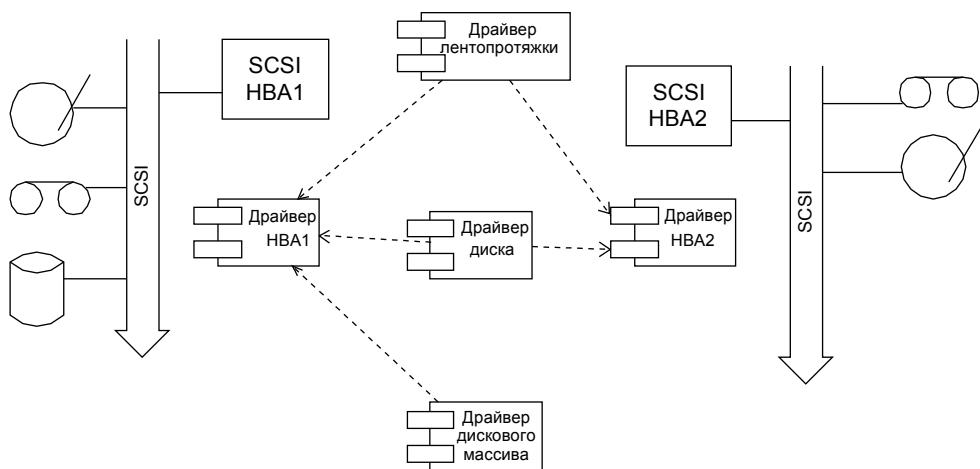


Рис. 10.1. Драйверы целевых устройств SCSI и драйвер HBA

Как правило, драйвер HBA напрямую не доступен прикладным программам (в отдельных случаях, впрочем, позволяют осуществлять над этим драйвером операции `ioctl`), а набор функций этого драйвера отличается от символьных и блочных устройств. Порядок инициализации таких драйверов (функции `_init`, `attach` и т. д.) в целом такой же, как и у обычных драйверов.

Как правило, драйвер HBA имеет функцию `transport` (например, в Solaris эта функция называется `tran_start`), которая осуществляет передачу целевому устройству команды, сформированной драйвером этого устройства. Если до завершения отработки предыдущей команды поступит следующая, драйвер может либо положиться на способность целевого устройства поддерживать очередь запросов, либо, если устройство этого не умеет, реализовать очередь запросов самостоятельно.

Кроме того, драйвер адаптера производит первичный анализ пришедших ответов на команды: какой из ранее переданных команд соответствует ответ, чем завершилась операция — успехом или ошибкой, пришли ли в ответ данные и если пришли, то сколько именно и куда их положить и т. д.

После завершения обработки запроса драйвер HBA вызывает callback драйвера целевого устройства. Эта callback-процедура может сформировать следу-

ющий запрос к адаптеру (или повторную попытку, если операция завершилась восстановимой ошибкой) либо просто оповестить пользовательский процесс о завершении операции и освободить память, занятую структурой запроса и буферами данных.

Аналогичные ситуации возникают и с другими устройствами. Например, с IBM PC-совместимыми компьютерами могут работать три основных типа устройств позиционного ввода: мыши, использующие протокол обмена, совместимый с Microsoft Mouse, мыши с протоколом Logitech, планшеты и дигитайзеры с протоколом Summagraphics. Устройства всех этих, а также нескольких менее распространенных типов могут подсоединяться как минимум к четырем различным периферийным портам: специальному "мышиному" разъему PS/2, к последовательной шине USB и к последовательному порту RS232, причем в качестве порта RS232 может использоваться как один из четырех стандартных портов IBM PC, так и, например, один из выходов мультипортовой платы (рис. 10.2).

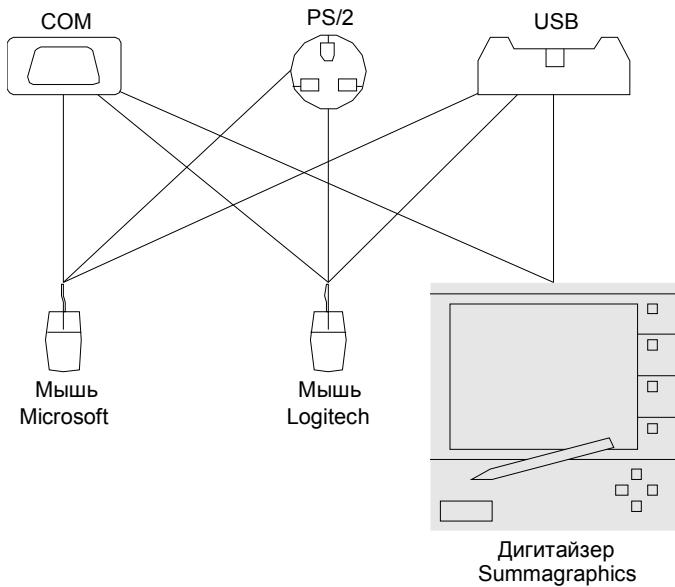


Рис. 10.2. Различные типы позиционных устройств ввода

В этом случае также целесообразно реализовать четыре самостоятельных драйвера транспортных портов (тем более что к этим портам могут подключаться и другие устройства) и три драйвера протоколов обмена, способных работать с любым транспортным портом. Объединение этих драйверов в одном модуле может создать некоторые — довольно, впрочем, сомнительные — удобства администратору системы, в которой используется стандарт-

ная комбинация порта и мыши, но поставит в безвыходное положение администратора системы, в которой комбинация нестандартная, например, не позволяя использовать мышь, подключенную к мультипортовой плате.

В современных системах семейства Unix многоэтапная обработка запросов штатно поддерживается потоковыми драйверами: ОС предоставляет специальные системные вызовы `push` и `pop`, позволяющие добавлять и удалять дополнительные драйверы, обслуживающие поток. Дополнительные драйверы могут преобразовывать данные потока (например, символы протокола мыши в координаты курсора и нажатия и отпускания кнопок) или обрабатывать запросы `ioctl` [docs.sun.com 805-7478-10].

В частности, в современных системах семейства Unix драйверы терминальных устройств должны уметь обрабатывать достаточно обширный набор запросов `ioctl` и выполнять ряд важных функций по управлению заданиями [Хевиленд/Грей/Салама 2000]. В монолитных системах эти функции обязан реализовать сам драйвер устройства (хотя ядро и облегчает создателю драйвера эту работу, предоставляя библиотеку сервисных функций — см. [Максвелл 2000]), в то время как в системах, имеющих потоковые драйверы, драйвер устройства может ограничиться решением своей собственной задачи — обеспечением обмена данными с устройством, а все сложные терминальные сервисы, если это необходимо, предоставляются простым добавлением к потоку драйвера модуля терминальной дисциплины (рис. 10.3).

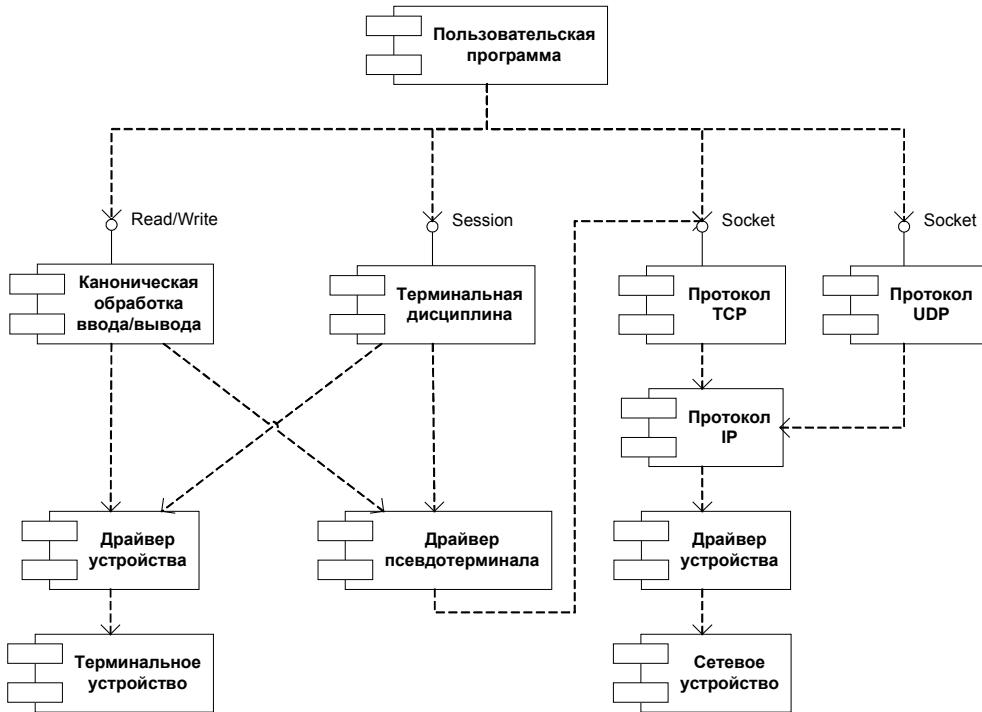


Рис. 10.3. Модули STREAMS

Многоуровневые драйверы в OS/2

Рассмотрим еще один подход к организации многоуровневых драйверов на примере DMD (Device Manager Driver — драйвер-менеджер класса устройств) в OS/2 [www.ibm.com OS/2 DDK] и достаточно типичной аппаратной конфигурации, содержащей HBA SCSI, к которому подключены пять устройств: жесткий диск, привод CD-ROM, магнитооптический диск, лентопротяжка и сканер. При этом каждое из устройств имеет свою специфику, так что управление ими сложно свести к общему набору функций.

Жесткий и магнитооптический диски наиболее схожи между собой, т. к. и то, и другое является запоминающим устройством большой емкости с произвольным доступом. Однако жесткий диск — неудаляемое устройство, а магнитооптический носитель можно извлечь из привода, не выключая компьютера. Это накладывает определенные требования на стратегию кэширования соответствующего устройства и требует от драйвера способности понимать и обрабатывать аппаратный сигнал о смене устройства.

Такой сигнал следует передать модулям управления *дисковым кэшем* и *файловой системой*, которые, в свою очередь, обязаны разумно обработать его: как минимум, дисковый кэш должен объявить все связанные с диском буферы неактуальными, а менеджер файловой системы должен сбросить все свои внутренние структуры данных, связанные с удаленным диском, и объяснить всем пользовательским программам, работавшим с этим диском, что их данные про-

пали. Другие аспекты работы с удаляемыми носителями обсуждаются в разд. 11.4.

CD-ROM, в свою очередь, нельзя рассматривать как удаляемый диск, доступный только для чтения: практически все приводы CD-ROM, кроме функции считывания данных, еще имеют функцию проигрывания музыкальных компакт-дисков.

Лентопротяжка и сканер вообще не являются устройствами памяти прямого доступа, а сканер даже с самой большой натяжкой нельзя рассматривать как устройство памяти.

Когда OS/2 управляет описанной аппаратной конфигурацией, оказываются задействованы пять DMD (рис. 10.4).

OS2DASD.DMD управляет классом запоминающих устройств прямого доступа и предоставляет стандартные функции для доступа к дискам.

OPTICAL.DMD обеспечивает управление устройствами прямого доступа с удаляемыми носителями. Основная его задача — обработка аппаратного сигнала смены носителя и оповещение других модулей системы (дискового кеша, файловой системы) об этой смене.

OS2CDROM.DMD обеспечивает специфические для приводов CD-ROM функции, например проигрывание аудиозаписей.

OS2SCSI.DMD и OS2ASPI.DMD — эти два модуля будут описаны далее.

Каждый из этих DMD не работает непосредственно с аппаратурой, а транслирует запросы пользовательских программ и других модулей ядра (в первую очередь, менеджеров файловых систем) в запросы к драйверу нижнего уровня. Такой подход позволяет вынести общую для класса устройств логику в DMD и не заниматься повторной реализацией этой логики в каждом новом драйвере.

В данном случае запросы предыдущих трех драйверов исполняет четвертый DMD: OS2SCSI.DMD. Этот DMD преобразует запросы к устройствам в команды SCSI и передает эти команды драйверу ADD (Adapter Device Driver — драйверу устройства-адаптера), т. е. собственно драйверу НВА. От ADD требуется только умение передавать команды на шину SCSI, обрабатывать и осуществлять диспетчеризацию пришедших на них ответов, т. е. он функционально аналогичен драйверу НВА в системах семейства Unix.

Пятый DMD — OS2ASPI.DMD — обеспечивает сервис ASPI (Advanced SCSI Programming Interface — продвинутый интерфейс для программирования SCSI). Он дает возможность прикладным программам и другим драйверам формировать произвольные команды SCSI и таким образом осуществлять доступ к устройствам, которые не являются дисками. Сервисом ASPI пользуются драйверы лентопротяжки и сканера [www.ibm.com OS/2 DDK].

При работе с устройствами ATA/ATAPI используется более простая и бедная возможностями структура, состоящая из драйвера IBM1S506.ADD (для некоторых типов адаптеров EIDE может понадобиться другой драйвер) и фильтра IBMATAPI.FLT. Драйвер обеспечивает инициализацию адаптера, передачу команд и работу с жесткими дисками ATA, а фильтр — формирование команд для подключаемых к тому же адаптеру устройств ATAPI (CD-ROM и магнитооптических дисков).

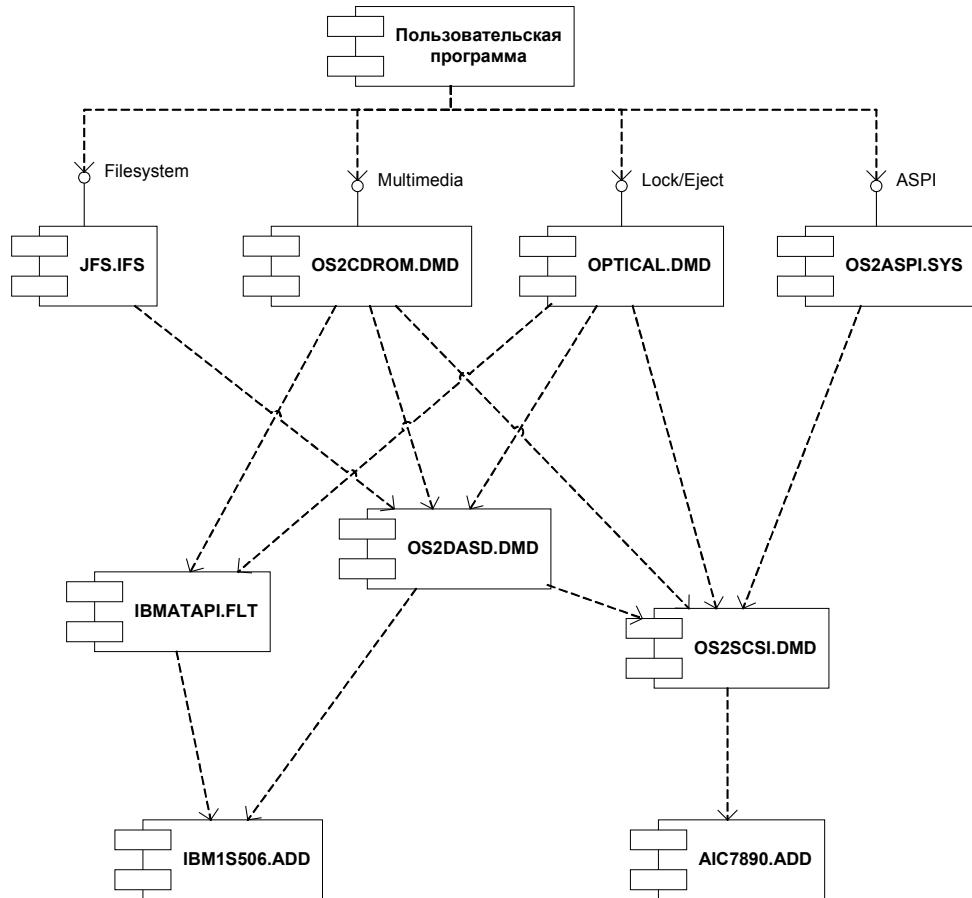


Рис. 10.4. Взаимодействие между DMD и ADD в OS/2 (в качестве примера драйвера файловой системы приведен модуль JFS.IFS)

На практике драйверы многих устройств используются исключительно или преимущественно другими модулями ОС (не обязательно драйверами), а не пользовательскими программами. Например, драйверы сетевых интерфейсов взаимодействуют практически исключительно с модулями ядра, реализующими сетевые и транспортные протоколы (напрямую с сетевым интерфейсом работают разве что конфигурационные утилиты, а также анализаторы протоколов и другие контрольно-диагностические средства), а драйверы жестких дисков преимущественно исполняют запросы файловых систем.

Именно этим и отличаются блочные устройства в системах семейства Unix от символьных: в классических ОС Unix блочные устройства вообще не доступны пользовательским программам, а все операции с ними осуществляются посредством файловой системы. Это несколько упрощает логику исполнения

операций — драйверу не надо заботиться об обмене данными с пользовательским адресным пространством. Кроме того, в отличие от обычных операций чтения и записи, в которых допустим обмен пакетами данных произвольного размера, блочный драйвер передает данные блоками, размер которых кратен 512 байтам.

На случай, если все-таки понадобится доступ к диску в обход файловой системы, драйвер блочного устройства создает две минорные записи для устройства — одну блочную и одну символьную, и все-таки предоставляет обычные, "символьные" операции чтения и записи. На практике, такой доступ почти всегда требуется утилитам создания и восстановления файловых систем, поэтому создание двух записей является обязательным.

Более радикально подошел к решению проблемы разработчик Linux Линус Торвальдс — в этой системе драйверы блочных устройств обязаны представлять символьные операции чтения и записи, но им не нужно создавать вторую минорную запись: пользователям разрешено работать с блочными устройствами (открывать, читать их и писать на них) так же, как и с символьными.

10.3. Защита драйверов

Чаще всего драйверы размещаются в адресном пространстве ядра системы, исполняются в высшем кольце защиты и имеют доступ для записи к сегментам данных пользовательских программ, и, как правило, к данным самого ядра. Это означает, что ошибка в драйвере может привести к разрушению пользовательских программ и самой ОС.

Чтобы избежать этого, пришлось бы выделять каждому драйверу свое адресное пространство и обеспечивать обмен данными между драйвером, ядром и пользовательской программой посредством статических разделяемых буферов или динамического отображения блоков данных между разными адресными пространствами. Оба решения приводят к значительным накладным расходам, а второе еще и предъявляет своеобразные требования к архитектуре диспетчера памяти (*подробнее эта проблема обсуждалась в разд. 5.2*).

Терминальный интерфейс в Unix

На практике иногда — особенно при использовании многоуровневых драйверов — оказывается возможным перенести отдельные функции работы с устройствами в контекст пользовательского процесса. Одна из относительно удачных попыток такого переноса была осуществлена в 70-е годы при разработке экранного редактора *vi* для ОС Unix (которая тогда еще была единственным представителем того, что потом превратилось в весьма обширное семейство ОС).

По замыслу разработчиков этот редактор должен был работать с большим количеством разных видеотерминалов, использовавших различные несовместимые системы команд для перемещения курсора и редактирования текста в буфере терминала и столь же несовместимые схемы кодирования специальных клавиш (стрелочек, функциональных клавиш и пр.). Как и в примере с мышами, разные терминалы могли подключаться к компьютеру через различные типы последовательных портов: "токовую петлю", позднее — RS232 и др.

Как в обсуждавшемся в разд. 10.2 в примере с мышами, в этом случае тоже было естественно разделить драйвер терминала на драйвер порта и модуль, занимающийся генерацией команд и анализом приходящих от терминала кодов расширенных клавиш. Вместо разработки отдельного модуля, решающего вторую часть задачи, была создана системная база данных для систем команд разных терминалов. Эта база данных представляет собой текстовый файл с именем /etc/termcap (terminal capabilities — возможности терминала).

Файл /etc/termcap состоит из абзацев. Заголовок абзаца представляет собой имя терминала, а текст состоит из фраз вида `Name=value`, разделенных символом ':'. `Name` представляет собой символьическое имя того или иного свойства, а `value` — его значение. Чаще всего это последовательность символов, формирующих соответствующую команду, или генерируемых терминалом при нажатии соответствующей клавиши. Кроме того, это может быть, например, ширина экрана терминала, измеренная в символах.

Для работы с терминальной базой данных позднее было создано несколько библиотек подпрограмм — низкоуровневая библиотека `termcap` и высокуюровневый пакет `curses`.

Для терминалов описанный подход оказался, если и не идеальным, то, во всяком случае, приемлемым. Но, например, для графических устройств он не подошел — системы команд различных устройств оказались слишком неподходящими и не сводимыми к единой системе "свойств".

Первым приближением к решению этой проблемы стало создание специализированных программ-фильтров. При использовании фильтров пользовательская программа генерирует графический вывод в виде последовательности команд некоторого языка. Фильтр принимает эти команды, синтезирует на их основе изображение и выводит его на графическое устройство. Вся поддержка различных устройств вывода возлагается на фильтр, пользовательская программа должна лишь знать его входной язык.

Самым удачным вариантом языка графического вывода в наше время считается PostScript — язык управления интеллектуальными принтерами, разработанный фирмой Adobe. PostScript предоставляет богатый набор графических примитивов, но основное его преимущество состоит в том, что это полнофункциональный язык программирования с условными операторами, циклами и подпрограммами.

Первоначально этот язык использовался только для управления дорогими моделями лазерных принтеров, но потом появились интерпретаторы этого языка, способные выводить PostScript на более простые устройства. Наиболее известной программой этого типа является GhostScript — программа, реализованная в рамках проекта GNU и доступная как freeware. GhostScript поставляется в виде исходных текстов на языке C, способна работать во многих операционных системах: практически во всех ОС семейства Unix, OS/2, MS/DR DOS и т. д. и

поддерживает практически все популярные модели графических устройств, принтеров, плоттеров и прочего оборудования.

Аналогичный подход используется в оконной системе X Window. В этой системе весь вывод на терминал осуществляется специальной программой-сервером. На сервер возлагается задача поддержки различных графических устройств. В большинстве реализаций сервер исполняется как обычная пользовательская программа, осуществляя доступ к устройству с помощью функций `ioctl` "установить видеорежим" и "отобразить видеопамять в адресное пространство процесса".

В обоих случаях "драйвер" оказывается разбит на две части: собственно драйвер, исполняющийся в режиме ядра, который занимается только обменом данными с устройством, и программу, интерпретирующую полученные данные и/или формирующую команды для устройства. Эта программа может быть довольно сложной, но ошибка в ней не будет фатальной для системы, т. к. она исполняется в пользовательском кольце доступа.

Интерес к драйверам, частично или полностью работающим с пользовательским или "повышенным пользовательским" уровнем привилегий, возникал многократно и по разным причинам. Еще в 1960-е в ОС DEC TOPS-10 для PDP-6/10 была предусмотрена возможность пользовательских задач, осуществляющих ввод/вывод. Для этого в диспетчере памяти был реализован промежуточный уровень привилегий, а в операционной системе — класс планирования реального времени. Нельзя не вспомнить также о i432 (эта система рассматривалась в *разд. 5.2*) и первых экспериментальных микроядерных ОС, появившихся в конце 1980-х.

В первом десятилетии XXI века интерес к драйверам, работающим вне ядра, возник снова. Один из доводов в пользу выноса драйверов из ядра уже приводился — это защита ядра от ошибок в драйверах. Нельзя также забывать и об опасности установки под видом драйвера троянского кода. Однако основные причины, по которым разработчики Linux и Windows заинтересовались драйверами, работающими вне ядра, напрямую не связаны с безопасностью.

Ядро Linux распространяется на условиях GPL, а эта лицензия не допускает интеграции в ядро кода, распространяющегося на других условиях. Это относится как к коду с открытыми исходниками, распространяемому на условиях других лицензий, например CDDL, так и к бинарному коду. При этом ряд поставщиков оборудования по разным причинам не хочет открывать исходный код драйверов для своих устройств. Так, по неофициальным объяснениям производителей, некоторые сетевые адAPTERы WiFi проходят лицензирование в комплексе с драйвером. Аппаратура сетевого адаптера может передавать сигнал на различных частотах, в том числе и на тех, работа на которых требует получения отдельной лицензии на передатчик. Драйвер производителя, разумеется, работает только на частотах, разрешенных для нелицензируемого

вещания, но программы, написанные третьими лицами, могут включить и лицензируемые частоты. Это объяснение звучит довольно странно и, скорее всего, дело не в этом, а все-таки в каких-то ложных понятых коммерческих интересах.

Так или иначе, проблемы с лицензионной чистотой включаемого в ядро Linux кода пробудили интерес к драйверам и другим модулям, традиционно являющимися модулями ядра, работающим за пределами ядра. Наиболее известны проекты FUSE (драйверы файловых систем, работающие в пользовательском контексте), FUSD и Gelato (драйверы устройств, работающие в пользовательском контексте). Производительность таких драйверов и таких файловых систем, разумеется, оставляет желать лучшего, поэтому они используются главным образом для работы с низкоскоростными устройствами, удаленныминосителями и др. Однако драйверы Gelato могут представлять собой полнофункциональные драйверы устройств PCI, осуществляющие доступ к портам устройства и обработку прерываний (для этого требуется специальный патч к ядру Linux), формирующие запросы ПДП и обслуживающие запросы модулей ядра.

FUSE интегрирована в ядро Linux 2.6.14. Поддержка FUSD и Gelato до сих пор остается экспериментальной.

В ядре 64-битной Windows Vista предполагается принудительная подпись всех устанавливаемых модулей. По некоторым сведениям, введение этой подписи было продиктовано лицензионным соглашением с консорциумом HD-DVD, который отказывался давать Microsoft право на поддержку HD-DVD в операционной системе без средств защиты данных от несанкционированного консорциумом доступа. Дело в том, что HD-DVD защищен криптографической системой цифрового управления ограничениями (DRM). Программные или аппаратные модули, работающие с ним, должны содержать ключи шифрования, необходимые для доступа к содержимому. Разумеется, код, исполняющийся с системным уровнем привилегий, может прочитать как ключи шифрования, так и дешифрованные данные и, таким образом, может снять защиту с содержимого диска.

Подпись всех модулей ядра применяется на некоторых закрытых вычислительных платформах, например на Microsoft Xbox. Однако PC-совместимые компьютеры — открытая платформа, для которой практически кто угодно может производить комплектующие, а комплектующие нуждаются в драйверах. В Windows 2000/XP/2003 подпись драйверов применялась, но не была обязательной. При этом подпись драйверов осуществлялась Microsoft и на условиях Microsoft. Эти условия в открытых источниках не разглашаются, но, по неофициальным данным, цена за подпись драйвера может достигать \$10 с каждого проданного устройства. Разумеется, многих производителей

такие условия не устраивают, поэтому попытка потребовать подписи от всех производителей драйверов могла бы привести к революции.

В Vista Microsoft пытается обойти эту проблему, продавая SDK для разработки драйверов за достаточно символическую цену (\$500) всем желающим. Этот SDK содержит ключ, которым можно подписать драйвер. Ключ во всех SDK одинаковый, поэтому утечка этого ключа и его попадание в руки разработчиков руткитов и других типов malware практически неизбежна.

Другой обходной путь, также реализованный в Vista, — это UMDF (User Mode Drivers Framework, среда исполнения драйверов в пользовательском режиме) [www.microsoft.com/UMDF]. Инфраструктура UMDF доступна также для Windows XP. Драйверы UMDF представляют собой компоненты COM (Common Object Model), не могут обрабатывать прерывания, напрямую обращаться к портам ввода-вывода, размещать физическую память и запрашивать ПДП, поэтому многие классы устройств (диски, дисплеи, сетевые контроллеры) не могут обслуживаться такими драйверами. Кроме того, UMDF драйвер не может отвечать на запросы модулей ядра — это еще одна причина, по которой UMDF драйверы не могут быть драйверами сетевых и дисковых устройств (ведь с сетевыми устройствами работают драйверы протоколов сетевого и транспортного уровня, а с дисковыми устройствами — драйверы файловых систем). Однако для низкоскоростных оконечных устройств, подключаемых к USB или IEEE 1394, и для некоторых псевдоустройств эти ограничения более или менее терпимы. По данным [www.microsoft.com/UMDF], список поддерживаемых устройств включает в себя видеокамеры, сотовые телефоны, PDA и портативные медиаплееры.

Наиболее известна поддержка драйверов пользовательского режима в QNX. Эта микроядерная ОС предоставляет класс планирования жесткого реального времени и допускает обращения к портам PCI и обработку прерываний пользовательскими процессами (при условии, что этот процесс обладает достаточным уровнем привилегий). При этом все драйверы (как ядерные, так и пользовательские) взаимодействуют с остальной системой через порты обмена данными, предоставляемые микроядром. Поэтому переделка пользовательских драйверов в ядерные и наоборот достаточно проста, а выбор между типами драйвера определяется лишь соображениями производительности и безопасности.

Так или иначе, в большинстве современных ОС большинство драйверов реализуются и исполняются как модули ядра.

Проще всего происходит загрузка драйверов в системах, в которых ядро собирается в статический загрузочный модуль. В них драйвер просто присоединяется редактором связей к образу ядра и без каких-либо дополнительных усилий оказывается в памяти в процессе загрузки системы.

В системах с динамической подгрузкой модулей ядра драйвер представляет собой перемещаемый загрузочный или объектный модуль, иногда того же формата, что и стандартные объектные или загрузочные модули в системе, а иногда и специализированного. В этом случае ядро должно содержать редактор связей, возможно являющийся функциональным подмножеством полноценного системного линкера.

Связывание кода драйвера с используемыми им функциями ядра обычно производится редактором связей в момент подключения модуля к образу ядра при статической сборке и в момент подгрузки модуля при сборке динамической. Однако связывание кода ядра с функциями драйвера, как правило, производится иным образом. Дело в том, что к ядру может подключаться переменное и, в большинстве систем, практически неограниченное количество драйверов. В этом случае обычно оказывается более удобным поддерживать таблицу точек входа зарегистрированных в системе драйверов.

Два основных подхода к формированию такой таблицы — это использование таблицы точек входа в качестве обязательного элемента формата драйвера либо регистрация точек входа в системной таблице функциями инициализации драйвера.

10.4. Архитектура драйвера

Типичный протокол работы с внешним устройством состоит из анализа запроса, передачи команды устройству, ожидания прерывания по завершении этой команды, анализа результатов операции и формирования ответа внешнему устройству. Многие запросы не могут быть выполнены в одну операцию, поэтому анализ результатов операции может привести к выводу о необходимости передать устройству следующую команду.

Драйвер, реализующий этот протокол, естественным образом распадается на две нити: основную, которая осуществляет собственно обработку запроса, и обработчик прерывания. В зависимости от ситуации, основная нить может представлять собою самостоятельную нить, либо ее код может исполняться в рамках нити, сформировавшей запрос.

В примере 10.1 приводится скелет функции `write()` драйвера последовательного устройства в системе Linux. Скелет упрощенный (в частности, никак не решается проблема реентерабельности функции `foo_write`. Использованный механизм синхронизации с обработчиком прерывания также оставляет желать лучшего), но имеет именно такую архитектуру, которая была описана ранее. Текст цитируется по документу [HOWTO khg], перевод комментариев и дополнительные комментарии мои.

Пример 10.1. Скелет драйвера последовательного устройства для ОС Linux

```
/* Основная нить драйвера */
static int foo_write(struct inode * inode, struct file * file,
                     char * buf, int count)
{
    /* Получить идентификатор устройства: */
    unsigned int minor = MINOR(inode->i_rdev);
    unsigned long copy_size;
    unsigned long total_bytes_written = 0;
    unsigned long bytes_written;

    /* Найти блок переменных состояния устройства */
    struct foo_struct *foo = &foo_table[minor];

    do {
        copy_size = (count <= FOO_BUFFER_SIZE ?
                     count : FOO_BUFFER_SIZE);

        /* Передать данные из пользовательского контекста */
        memcpy_fromfs(foo->foo_buffer, buf, copy_size);

        while (copy_size) {
            /* Здесь мы должны инициализировать прерывания*/

            if (some_error_has_occurred) {
                /* Здесь мы должны обработать ошибку */
            }

            current->timeout = jiffies + FOO_INTERRUPT_TIMEOUT;
            /* Установить таймаут на случай, если прерывание будет пропущено */
            interruptible_sleep_on(&foo->foo_wait_queue);

            if (some_error_has_occurred) {
                /* Здесь мы должны обработать ошибку */
            }

            bytes_written = foo->bytes_xfered;
            foo->bytes_written = 0;
            if (current->signal & ~current->blocked) {
                if (total_bytes_written + bytes_written)
                    return total_bytes_written + bytes_written;
            }
        }
    }
```

```
        return -EINTR; /* Ничего не было записано,
                         системный вызов был прерван, требуется
                         повторная попытка */
    }
}

total_bytes_written += bytes_written;
buf += bytes_written;
count -= bytes_written;

} while (count > 0);

return total_bytes_written;
}

/* Обработчик прерывания */
static void foo_interrupt (int irq)
{
    struct foo_struct *foo = &foo_table[foo_irq[irq]];

    /* Здесь необходимо выполнить все действия, которые должны
       быть выполнены по прерыванию.

       Флаг в foo_table указывает, осуществляется операция
       чтения или записи. */

    /* Увеличить foo->bytes_xfered на количество фактически
       переданных символов */

    if (буфер полон/пуст)
        wake_up_interruptible(&foo->foo_wait_queue);
}
```

Примечание

Обратите внимание, что кроме инициализации устройства драйвер перед засыпанием еще устанавливает "будильник" — таймер, который должен разбудить процесс через заданный интервал времени. Это необходимо на случай, если произойдет аппаратная ошибка и устройство не генерирует прерывания. Если бы такой будильник не устанавливался, драйвер в случае ошибки мог бы заснуть навсегда, заблокировав при этом пользовательский процесс. В нашем случае таймер также используется, чтобы разбудить процесс, если прерывание произойдет до вызова `interruptible_sleep_on` основной нитью.

Многие устройства, однако, требуют для исполнения некоторых, даже относительно простых, операций, несколько команд и несколько прерываний. Так, при записи данных посредством контроллера гибких дисков, драйвер должен:

- включить мотор дисковода;
- дождаться, пока диск разгонится до рабочей скорости (большинство контроллеров генерируют по этому случаю прерывание);
- дать устройству команду на перемещение считающей головки;
- дождаться прерывания по концу операции перемещения;
- запрограммировать ПДП и инициировать операцию записи;
- дождаться прерывания, сигнализирующего о конце операции.

Лишь после этого можно будет передать данные программе. Наивная реализация таких многошаговых операций могла бы выглядеть так (за основу по-прежнему взят код из [HOWTO khg], обработка ошибок опущена), как показано в примере 10.2.

Пример 10.2. Простой драйвер контроллера гибкого диска

```
/* Обработчики прерываний в зависимости от состояния */
void handle_spinup_interrupt (int irq, fdd_struct *fdd)
{
    if (motor_speed_ok(fdd))
        wake_up_interruptible((&fdd->fdd_wait_queue));
}

void handle_seek_interrupt (int irq, fdd_struct *fdd)
{
    if (verify_track(fdd))
        wake_up_interruptible((&fdd->fdd_wait_queue));
}

void handle_dma_interrupt (int irq, fdd_struct *fdd)
{
    /* Увеличить fdd->bytes_xfered на количество фактически
     переданных символов */

    if (буфер полон/пуст)
        wake_up_interruptible(&fdd->fdd_wait_queue);
}
```

```
/* Основная нить драйвера */
static int fdd_write(struct inode * inode, struct file * file,
                     char * buf, int count)
{
    /* Получить идентификатор устройства: */
    unsigned int minor = MINOR(inode->i_rdev);
    unsigned long copy_size;
    unsigned long total_bytes_written = 0;
    unsigned long bytes_written;
    int state;

    /* Найти блок переменных состояния устройства */
    struct fdd_struct *fdd = &fdd_table[minor];

    do {
        copy_size = (count <= FDD_BUFFER_SIZE ?
                     count : FDD_BUFFER_SIZE);
        /* Передать данные из пользовательского контекста */
        memcpys_fromfs(fdd->fdd_buffer, buf, copy_size);

        while (copy_size) {
            if (!motor_speed_ok(fdd)) {
                fdd->handler = handle_spinup_interrupt;
                turn_motor_on(fdd);
                current->timeout = jiffies + FDD_INTERRUPT_TIMEOUT;
                interruptible_sleep_on(&fdd->fdd_wait_queue);
                if (current->signal & ~current->blocked) {
                    if (total_bytes_written)
                        return total_bytes_written;
                    else
                        return -EINTR; /* Ничего не было записано,
                                         системный вызов был прерван, требуется
                                         повторная попытка */
                }
            }
        }

        if (fdd->current_track != CALCULATE_TRACK(file)) {
            fdd->handler = handle_seek_interrupt;
            seek_head(fdd, CALCULATE_TRACK(file));
            current->timeout = jiffies + FDD_INTERRUPT_TIMEOUT;
            interruptible_sleep_on(&fdd->fdd_wait_queue);
        }
    }
```

```
if (current->signal & ~current->blocked) {
    if (total_bytes_written)
        return total_bytes_written;
    else
        return -EINTR; /* Ничего не было записано,
                        системный вызов был прерван, требуется
                        повторная попытка */
}
}

fdd->handler = handle_dma_interrupt;
setup_fdd_dma(fdd->fdd_buffer+bytes_xfered, copy_size);
issue_write_command(fdd);
current->timeout = jiffies + FDD_INTERRUPT_TIMEOUT;
interruptible_sleep_on(&fdd->fdd_wait_queue);

bytes_written = fdd->bytes_xfered;
fdd->bytes_written = 0;
if (current->signal & ~current->blocked) {
    if (total_bytes_written + bytes_written)
        return total_bytes_written + bytes_written;
    else
        return -EINTR; /* Ничего не было записано,
                        системный вызов был прерван, требуется
                        повторная попытка */
}
}

total_bytes_written += bytes_written;
buf += bytes_written;
count -= bytes_written;

} while (count > 0);

return total_bytes_written;
}

/* Обработчик прерывания */
static void fdd_interrupt (int irq)
{
    struct fdd_struct *fdd = &fdd_table[fdd_irq[irq]];
}
```

```
if (fdd->handler != NULL) {  
    fdd->handler (irq, fdd);  
    fdd->handler=NULL;  
} else  
{  
/* Не наше прерывание? */  
}  
}
```

Видно, что предлагаемый драйвер осуществляет обработку ошибок и формирование последующих команд в основной нити драйвера. Велик соблазн перенести эти функции или их часть в обработчик прерываний. Такое решение позволяет сократить интервал между последовательными командами и, таким образом, возможно, повысить производительность работы устройства.

Однако слишком большое время, проводимое в обработчике прерывания, неизбежно с точки зрения других модулей системы, т. к. может увеличить реальное время реакции для них. Особенно важно это для систем, которые выключают планировщик на время обслуживания прерываний. Поэтому многие ОС накладывают ограничения на время обслуживания прерываний, и часто это ограничение исключает возможность формирования команд и производства других сложных действий в обработчике.

Обработчик, таким образом, должен выполнять лишь те операции, которые требуется выполнить немедленно. В частности, многим устройствам требуется так или иначе объяснить, что прерывание обработано, чтобы они сняли сигнал запроса прерывания. Если этого не сделать, после возврата из обработчика и обусловленного этим снижения приоритета ЦПУ, обработчик будет вызван опять.

Впрочем, нередко предлагается путь к обходу этого ограничения: обработчикам прерываний разрешено создавать высокоприоритетные нити, которые начнут исполняться сразу же после того, как будут обслужены все прерывания. В дальнейшем мы будем называть эти высокоприоритетные нити *fork-процессами* (этот термин применяется в VMS; другие ОС, хотя и используют аналогичные понятия, часто не имеютнятой терминологии для их описания).

Fork-процессы в VMS

С точки зрения планировщика VMS, fork-процесс представляет собой нить с укороченным контекстом. Вместо обычного дескриптора процесса (PCB — Process Control Block) используется UCB — Unit Control Block, блок управления

устройством. Укорочение заключается в том, что эта нить может работать только с одним банком виртуальной памяти из трех, имеющихся у процессора VAX, а именно с системным (полный список банков памяти VAX приведен в главе 5); таким образом, при переключении контекста задействуется меньше регистров диспетчера памяти. Fork-процесс имеет более высокий приоритет, чем все пользовательские процессы, и может быть вытеснен только более приоритетным fork-процессом и обработчиком прерывания.

При использовании fork-процессов обслуживание прерывания распадается на собственно обработчик (вызываемый по сигналу прерывания и исполняемый с соответствующим приоритетом) и код постобработки, исполняемый fork-процессом, на который не распространяются ограничения времени и который вполне может осуществить планирование следующих операций (пример 10.3).

Пример 10.3. Более сложный драйвер контроллера гибкого диска

```
/* Обработчики прерываний в зависимости от состояния */
void schedule_seek(fdd_struct *fdd)
{
    if (!motor_speed_ok(fdd)) {
        fdd->handler = schedule_seek;
        retry_spinup();
    }
    if (fdd->current_track != CALCULATE_TRACK(fdd->file)) {
        fdd->handler = schedule_command;
        seek_head(fdd, CALCULATE_TRACK(file));
    } else
        /* Мы уже на нужной дорожке */
        schedule_operation(fdd);
}

void schedule_operation(fdd_struct *fdd)
{
    if (fdd->current_track != CALCULATE_TRACK(fdd->file)) {
        fdd->handler = schedule_operation;
        retry_seek(fdd);
        return;
    }
    switch (fdd->operation) {
        case FDD_WRITE:
            fdd->handler = handle_dma_write_interrupt;
```

```
    setup_fdd_dma(fdd->fdd_buffer+fdd->bytes_xfered, fdd->copy_size);
    issue_write_command(fdd);
    break;
case FDD_READ:
    fdd->handler = handle_dma_read_interrupt;
    setup_fdd_dma(fdd->fdd_buffer+fdd->bytes_xfered, fdd->copy_size);
    issue_read_command(fdd);
    break;
/* Здесь же мы должны обрабатывать другие команды,
 требующие предварительного SEEK */
}

void handle_dma_write_interrupt (fdd_struct *fdd)
{
/* Увеличить fdd->bytes_xfered на количество фактически
 переданных символов */

if (буфер полон/пуст)
/* Здесь мы не можем передавать данные из пользовательского
 адресного пространства. Надо будить основную нить */
    wake_up_interruptible(&fdd->fdd_wait_queue);
else {
    fdd->handler = handle_dma_write_interrupt;
    setup_fdd_dma(fdd->fdd_buffer+fdd->bytes_xfered, fdd->copy_size);
    issue_write_command(fdd);
}
}

/* Основная нить драйвера */
static int fdd_write(struct inode * inode, struct file * file,
                     char * buf, int count)
{
/* Получить идентификатор устройства: */
unsigned int minor = MINOR(inode->i_rdev);
/* Обратите внимание, что почти все переменные основной нити
 "переехали" в описатель состояния устройства */
/* Найти блок переменных состояния устройства */
struct fdd_struct *fdd = &fdd_table[minor];

fdd->total_bytes_written = 0;
fdd->operation = FDD_WRITE;
```

```
do {
    fdd->copy_size = (count <= FDD_BUFFER_SIZE ?
                        count : FDD_BUFFER_SIZE);
    /* Передать данные из пользовательского контекста */
    memcopy_fromfs(fdd->fdd_buffer, buf, copy_size);

    if (!motor_speed_ok()) {
        fdd->handler = schedule_seek;
        turn_motor_on(fdd);
    } else
        schedule_seek(fdd);
    }

    current->timeout = jiffies + FDD_INTERRUPT_TIMEOUT;
    interruptible_sleep_on(&fdd->fdd_wait_queue);
    if (current->signal & ~current->blocked) {
        if (fdd->total_bytes_written+fdd->bytes_written)
            return fdd->total_bytes_written+fdd->bytes_written;
        else
            return -EINTR; /* Ничего не было записано,
                           системный вызов был прерван, требуется
                           повторная попытка */
    }
    fdd->total_bytes_written += fdd->bytes_written;
    fdd->buf += fdd->bytes_written;
    count -= fdd->bytes_written;

} while (count > 0);

return total_bytes_written;
}

static struct tq_struct floppy_tq;

/* Обработчик прерывания */
static void fdd_interrupt (int irq)
{
    struct fdd_struct *fdd = &fdd_table[fdd_irq[irq]];

    if (fdd->handler != NULL) {
```

```
void (*handler)(int irq, fdd_struct * fdd);
floppy_tq.routine = (void *)(void *)fdd->handler;
floppy_tq.parameter = (void *)fdd;
fdd->handler=NULL;
queue_task (&floppy_tq, &tq_immediate);
} else
{
/* Не наше прерывание? */
}
}
```

Видно, что теперь наш драйвер представляет собой последовательность функций, вызываемых обработчиком прерываний. Обратите внимание, что если мы торопимся, очередную функцию можно вызывать и непосредственно в обработчике, а не создавать для нее fork-процесс посредством `queue_task`. Но самое главное, на что нам следует обратить внимание, — последовательность этих функций не задана жестко: каждая из функций сама определяет, какую операцию вызывать следующей. В том числе, она может решить, что следующая операция может состоять в вызове той же самой функции. В примере 10.3 мы используем эту возможность для простой обработки ошибок: повтора операции, которая не получилась.

Для того чтобы понять, что же у нас получилось, какие возможности нам открывает такая архитектура и как ими пользоваться, нам следует сделать экскурс в одну из важных областей теории программирования.

10.4.1. Введение в конечные автоматы

Конечный автомат (в современной англоязычной литературе используется также более выразительное, на мой взгляд, обозначение, не имеющее хорошего русского эквивалента — *state machine*, дословно переводимое как машина состояний) представляет собой устройство, имеющее внутреннюю память (переменные состояния), а также набор входов и выходов. Объем внутренней памяти у конечных автоматов, как следует из названия, конечен. Автоматы с неограниченным объемом внутренней памяти называются *бесконечными автоматами*, они нереализуемы и используются только в теоретических построениях [Минский 1971].

Однако некоторые разновидности теоретически бесконечных автоматов — например, стековые — могут быть реализованы в форме автоматов с практически неограниченной памятью — например, достаточно глубоким стеком — и находят практическое применение, например при синтаксическом анализе языков с вложенными структурами [Кормен/Лейзерсон/Ривест 2000].

Работа автомата состоит в том, что он анализирует состояния своих входов, и, в зависимости от значений входов и своего внутреннего состояния, изменяет значения выходов и внутреннее состояние. Правила, в соответствии с которыми происходит изменение, описываются *таблицей* или *диаграммой переходов*. Диаграмма переходов представляет собой граф, вершины которого соответствуют допустимым состояниям внутренних переменных автомата, а ребра — допустимым переходам между ними. Переходы между вершинами направленные: наличие перехода из А в В не означает, что существует переход из В в А. Наличие перехода в обоих направлениях символизируется двумя ребрами, соединяющими одну пару вершин. Такой граф называется ориентированным [Кормен/Лейзерсон/Ривест 2000]. Таблица переходов может рассматриваться как матричное представление диаграммы переходов.

Блок-схемы (рис. 10.5) являются обычным способом визуализации графов переходов и используются для описания алгоритмов с 60-х годов. Любой алгоритм, исполняющийся на фон-неймановском компьютере с конечным объемом памяти (а также любой физически исполнимый алгоритм), может быть описан как конечный автомат и изображен в виде блок-схемы.

У конечных автоматов с ограниченным числом допустимых значений входов граф переходов всегда конечен, хотя и может содержать циклы (замкнутые пути) и контуры (совокупности различных путей, приводящих к одной и той же вершине). Понятно, что для автомата с графом, содержащим циклы, невозможно гарантировать *финитности* — завершения работы за конечное время. Как известно, задача доказательства финитности алгоритма, хотя и решена во многих частных случаях, в общем случае алгоритмически неразрешима [Минский 1971].

Применительно к драйверам внешних устройств, циклический граф может соответствовать повторным попыткам выполнения операции после ее неудачи. Понятно, что на практике количество таких попыток следует ограничивать. Самый простой способ такого ограничения — введение счетчика попыток. Формально после этого состояния с различными значениями счетчика превращаются в наборы состояний, а граф переходов становится ациклическим (рис. 10.6), но для достаточно большого количества повторений опять-таки необозримым, поэтому на практике часто используют сокращенную блок-схему, в которой состояния с разными значениями счетчика цикла изображаются как одно состояние.

Анализ полной или сокращенной блок-схемы алгоритма методами теории графов, хотя и не может однозначно дать ответ на вопрос о его финитности, может оказать значительную помощь в оценке алгоритма, в том числе и в поиске "узких" с точки зрения финитности мест. В [Кнут 2000] приводятся примеры такого анализа для некоторых простых алгоритмов.

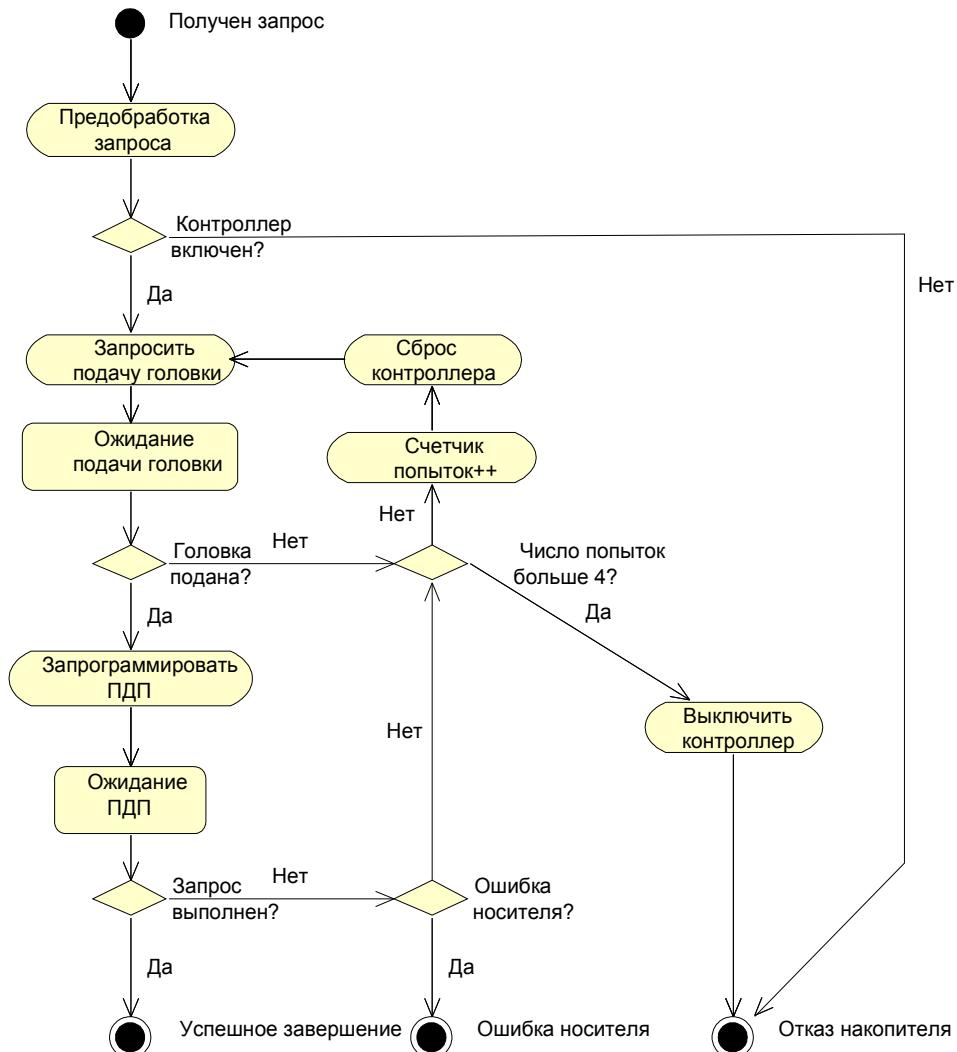


Рис. 10.5. Блок-схема драйвера

Алгоритмы основной массы реально применяемых программ (особенно использующих переменные состояния большого объема) имеют совершенно небозримые блок-схемы. Отчасти это обходится декомпозицией программного комплекса на отдельные модули с более обозримой функциональностью и алгоритмом, но все-таки далеко не для всех алгоритмов представление в виде конечного автомата естественно.

С другой стороны, ряд даже довольно сложных алгоритмов естественным образом описывается автоматами с небольшим числом состояний, которые

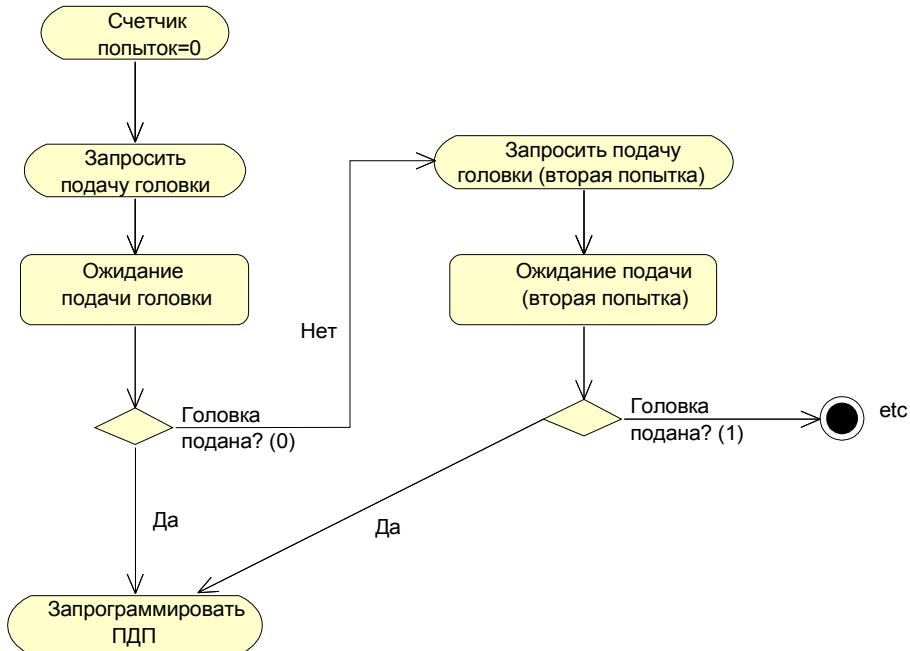
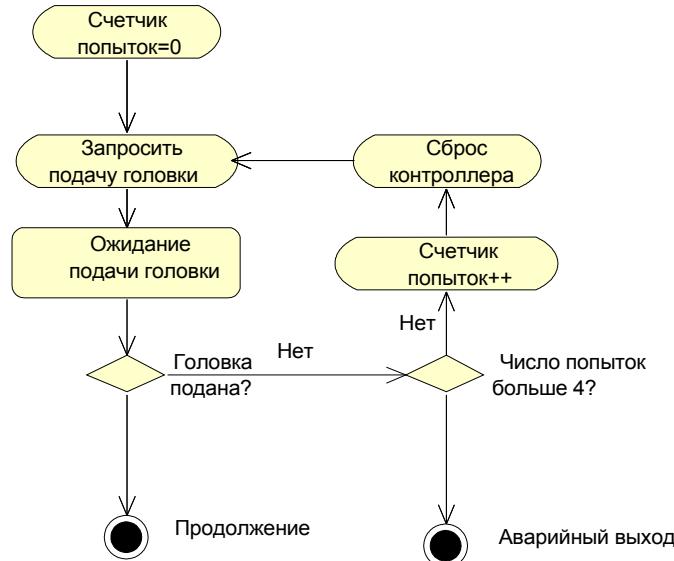


Рис. 10.6. Развёртывание циклов в графе состояния

могут быть закодированы одной скалярной переменной состояния или стеком таких переменных. Такие автоматы находят применение в самых разнообразных задачах: лексическом и синтаксическом разборе контекстно-свободных и многих типах контекстно-связанных языков [Кормен/Лейзерсон/Ривест 2000], реализации сетевых протоколов, задачах корпоративного документооборота [Керн/Линд 2000] и др. В частности, легко понять, что обсуждаемый нами алгоритм драйвера относится именно к этой категории алгоритмов.

Два основных подхода к реализации конечных автоматов — это *развернутые* (unrolled) *автоматы* и *автоматы общего вида*. Примером развернутого конечного автомата является код основной нити примера 10.2. Понятно, что развертыванию поддаются только автоматы с весьма специфической — линейной или древовидной — структурой графа состояний, и если в процессе уточнения требований мы выясним, что структура автомата должна быть более сложной, нам придется полностью реорганизовать код.

Автомат общего вида выглядит несколько сложнее, но, научившись распознавать его конструкцию, легко разрабатывать такие программы по заданной блок-схеме и, наоборот, восстанавливать график состояний по коду программы. Главным преимуществом грамотно реализованного конечного автомата является легкость модификации: если график переходов изменится, нам надо будет изменить код только тех узлов, которые затронуты изменением.

Примеры реализации конечных автоматов такого типа на процедурном языке программирования приводятся во многих учебниках программирования, например [Грогоно 1982]. Чаще всего реализация состоит из цикла, условием выхода из которого является достижение автоматом финального состояния, и размещенного в теле цикла оператора вычислимого перехода с переменной состояния в качестве селектора. Конечный автомат, похожий на эту классическую реализацию, приведен в примере 10.4.

Пример 10.4. Конечный автомат драйвера контроллера IDE/ATA для OS/2

```
VOID NEAR StartSM( NPACB npACB )
{
    /*-----*/
    /* Проверка счетчика использований ACB */
    /* ----- */
    /* Автомат реentrantен для каждого ACB
    /*-----*/
    DISABLE

    npACB->UseCount++;

    /*-----*/
```

```
if( npACB->UseCount == 1 )
{
do
{
ENABLE
do
{
npACB->Flags &= ~ACBF_WAITSTATE;

switch (npACB->State)
{
case ACBS_START:
    StartState(npACB);
break;
case ACBS_INTERRUPT:
    InterruptState(npACB);
break;
case ACBS_DONE:
    DoneState(npACB);
break;
case ACBS_SUSPEND:
    SuspendState(npACB);
break;
case ACBS_RETRY:
    RetryState(npACB);
break;
case ACBS_RESETCHECK:
    ResetCheck(npACB);
break;
case ACBS_ERROR:
    ErrorState(npACB);
break;
}

} while ( !(npACB->Flags & ACBF_WAITSTATE) );

DISABLE
}

while ( --npACB->UseCount );
}
}
```

Конечный автомат драйвера OS/2

Несмотря на простоту, пример 10.4 нуждается в комментариях. Параметр функции `StartSM` — `ACB` (*Adapter Control Block* — блок управления адаптером, так в OS/2 называется блок переменных состояния устройства). `ACB` содержит указатель на очередь запросов `IORB` (*Input/Output Request Block* — блок запроса на ввод/вывод) и скалярную переменную `State`, которая указывает, в каком состоянии сейчас находится обработка первого запроса в очереди. По коду этого состояния определяется, какую функцию следует вызвать. В телах этих функций, в зависимости от результата операции, происходит установка следующего значения переменной состояния и, возможно, флага `ACB_WAITSTATE`.

Функция `StartSM` (*Start State Machine*) вызывается как из функции обработки запросов, так и из обработчика прерывания. Поэтому перед входом в собственный автомат и после выхода из него стоит код, использующий поле `prACB->UseCount` как флаговую переменную, чтобы не допустить одновременного входа в автомат из обоих возможных нитей исполнения. Обратите также внимание, что макросами `ENABLE` и `DISABLE` (запрет и разрешение прерываний) окружена работа с флаговой переменной, но не сам автомат.

В качестве упражнения читателю предлагается понять, как же обеспечивается вызов функции `InterruptState`, если во время прерывания основной поток драйвера все еще находился в теле автомата.

Полный текст драйвера IDE/ATA для OS/2 включен в стандартную поставку DDK (Driver Development Kit — набор инструментов [для] разработчика драйверов), который до 2005 года мог быть найден на сайте [www.ibm.com] OS/2 DDK].

Построенный нами в примере 10.3 код внешне совсем не похож на пример 10.4, но в действительности также представляет собой конечный автомат: в качестве переменной состояния используется переменная `fdd->handler`, а в качестве дискретных значений этой переменной — указатели на функции, обрабатывающие конкретные состояния.

10.4.2. Архитектура драйвера

Драйвер, таким образом, состоит из основной нити, обработчика прерывания и, возможно, одной или нескольких высокоприоритетных нитей, создаваемых обработчиком. Все эти нити совместно (и, как правило, гарантируя взаимоисключение) исполняют более или менее сложный конечный автомат, состояния которого соответствуют этапам выполнения очередного запроса к устройству.

Как правило, первое состояние автомата обрабатывается основной нитью драйвера, а последующие — обработчиком прерываний. В финальном состоянии автомата мы сообщаем процессу, породившему запрос, что запрос отработан. В зависимости от протокола взаимодействия этого процесса с основной нитью драйвера, такое сообщение может осуществляться как `fork`-процессом, так и пробуждением основной нити.

Драйвер IDE/ATA для Linux

В примере 10.5 приведена основная функция обработки запроса и функция обработки прерывания, используемая при записи нескольких секторов. Обе эти функции вызываются драйвером контроллера IDE/ATA, который представляет собой диспетчер запросов к подключенным к контроллеру устройствам.

Структура `*hwgroup` представляет собой блок переменных состояний контроллера устройства. Эта структура содержит также указатель на текущий запрос к устройству. Информации, содержащейся в этих структурах, достаточно, чтобы очередная функция конечного автомата драйвера узнала все, необходимое ей для выполнения очередного этапа запроса. В данном случае конечный автомат весьма прост и состоит из многократного вызова функции `ide_multwrite`, копирующей в контроллер очередной блок данных. Условием завершения автомата служат ошибка контроллера либо завершение запроса. Функции `ide_dma_read`, `ide_dma_write`, `ide_read` и `ide_write`, исполняемые машиной состояний при обработке других запросов, не приводятся.

Пример 10.5. Фрагменты драйвера диска IDE/ATA ОС Linux 2.2, перевод комментариев автора

```
/*
 * ide_multwrite() передает приводу блок из не более, чем mcount
 * секторов как часть многосекторной операции записи.
 *
 * Возвращает 0 при успехе.
 *
 * Обратите внимание, что мы можем быть вызваны из двух контекстов -
 * контекста do_rw и контекста IRQ. IRQ (Interrupt Request,
 * запрос прерывания) может произойти в любой
 * момент после того, как мы выведем полное количество секторов,
 * поэтому мы должны обновлять состояние _до_ того, как мы выведем
 * последнюю часть данных!
 */
int ide_multwrite (ide_drive_t *drive, unsigned int mcount)
{
    ide_hwgroup_t *hwgroup= HWGROUP(drive);
    struct request *rq = &hwgroup->wrq;

    do {
        char *buffer;
        int nsect = rq->current_nr_sectors;

        if (nsect > mcount)
            nsect = mcount;
```

```
    mcount -= nsect;
    buffer = rq->buffer;

    rq->sector += nsect;
    rq->buffer += nsect << 9;
    rq->nr_sectors -= nsect;
    rq->current_nr_sectors -= nsect;

/* Переходим ли мы к следующему bh после этого? */
if (!rq->current_nr_sectors) {
    struct buffer_head *bh = rq->bh->b_reqnext;

/* Завершиться, если у нас кончились запросы */
    if (!bh) {
        mcount = 0;
    } else {
        rq->bh = bh;
        rq->current_nr_sectors = bh->b_size >> 9;
        rq->buffer = bh->b_data;
    }
}

/*
 * Теперь мы все настроили, чтобы прерывание
 * снова вызвало нас после последней передачи.
 */
    idedisk_output_data(drive, buffer, nsect<<7);
} while (mcount);

return 0;
}

/*
 * multwrite_intr() — обработчик прерывания многосекторной записи
 */
static ide_startstop_t multwrite_intr (ide_drive_t *drive)
{
    byte stat;
    int i;
    ide_hwgroup_t *hwgroup = HWGROUP(drive);
    struct request *rq = &hwgroup->wrq;
```

```
if (OK_STAT(stat=GET_STAT(), DRIVE_READY, drive->bad_wstat)) {
    if (stat & DRQ_STAT) {
        /*
         * Привод требует данных. Помним, что rq -
         * копия запроса.
        */
        if (rq->nr_sectors) {
            if (ide_multwrite(drive, drive->mult_count))
                return ide_stopped;
            ide_set_handler (drive, &multwrite_intr, WAIT_CMD, NULL);
            return ide_started;
        }
    } else {
        /*
         * Если копирование всех блоков завершилось,
         * мы можем завершить исходный запрос.
        */
        if (!rq->nr_sectors) { /* all done? */
            rq = hwgroup->rq;
            for (i = rq->nr_sectors; i > 0;){
                i -= rq->current_nr_sectors;
                ide_end_request(1, hwgroup);
            }
            return ide_stopped;
        }
    }
    return ide_stopped; /* Оригинальный код делал это
    здесь (?) */
}
return ide_error (drive, "multwrite_intr", stat);
}

/*
 * do_rw_disk () передает команды READ и WRITE приводу,
 * используя LBA, если поддерживается, или CHS, если нет, для адресации
 * секторов. Функция do_rw_disk также передает специальные запросы.
*/
static ide_startstop_t do_rw_disk (ide_drive_t *drive, struct request
*rq, unsigned long block)
{
```

```
if (IDE_CONTROL_REG)
    OUT_BYT(ide->ctl, IDE_CONTROL_REG);
OUT_BYT(rq->nr_sectors, IDE_NSECTOR_REG);
if (ide->select.b.lba) {
    OUT_BYT(block, IDE_SECTOR_REG);
    OUT_BYT(block>>8, IDE_LCYL_REG);
    OUT_BYT(block>>8, IDE_HCYL_REG);
    OUT_BYT((block>>8)&0x0f|ide->select.all, IDE_SELECT_REG);
} else {
    unsigned int sect, head, cyl, track;
    track = block / ide->sect;
    sect = block % ide->sect + 1;
    OUT_BYT(sect, IDE_SECTOR_REG);
    head = track % ide->head;
    cyl = track / ide->head;
    OUT_BYT(cyl, IDE_LCYL_REG);
    OUT_BYT(cyl>>8, IDE_HCYL_REG);
    OUT_BYT(head|ide->select.all, IDE_SELECT_REG);
}

if (rq->cmd == READ) {
#endif CONFIG_BLK_DEV_IDEDMA
    if (ide->using_dma && !(HWIF(ide)->dmaproc(ide_dma_read, ide)))
        return ide_started;
#endif /* CONFIG_BLK_DEV_IDEDMA */
    ide_set_handler(ide, &read_intr, WAIT_CMD, NULL);
    OUT_BYT(ide->mult_count ? WIN_MULTREAD : WIN_READ,
            IDE_COMMAND_REG);
    return ide_started;
}
if (rq->cmd == WRITE) {
    ide_startstop_t startstop;
#endif CONFIG_BLK_DEV_IDEDMA
    if (ide->using_dma && !(HWIF(ide)->dmaproc(ide_dma_write,
                                                    ide)))
        return ide_started;
#endif /* CONFIG_BLK_DEV_IDEDMA */
    OUT_BYT(ide->mult_count ? WIN_MULTWRITE : WIN_WRITE,
            IDE_COMMAND_REG);
    if (ide_wait_stat(&startstop, ide, DATA_READY, ide->bad_wstat,
```

```
        WAIT_DRQ)) {  
    printk(KERN_ERR "%s: no DRQ after issuing %s\n", drive->name,  
          drive->mult_count ? "MULTWRITE" : "WRITE");  
    return startstop;  
}  
if (!drive->unmask)  
    __cli(); /* только локальное ЦПУ */  
if (drive->mult_count) {  
    ide_hwgroup_t *hwgroup = HWGROUP(drive);  
/*  
 * Эта часть выглядит некрасиво, потому что мы ДОЛЖНЫ установить  
 * обработчик перед выводом первого блока данных.  
 * Если мы обнаруживаем ошибку (испорченный список буферов)  
 * в ide_multiwrt(),  
 * нам необходимо удалить обработчик и таймер перед возвратом.  
 * К счастью, это НИКОГДА не происходит (правильно?).  
 * Кажется, кроме случаев, когда мы получаем ошибку...  
 */  
    hwgroup->wrq = *rq; /* scratchpad */  
    ide_set_handler (drive, &multwrite_intr, WAIT_CMD, NULL);  
    if (ide_multwrite(drive, drive->mult_count)) {  
        unsigned long flags;  
        spin_lock_irqsave(&io_request_lock, flags);  
        hwgroup->handler = NULL;  
        del_timer(&hwgroup->timer);  
        spin_unlock_irqrestore(&io_request_lock, flags);  
        return ide_stopped;  
    }  
} else {  
    ide_set_handler (drive, &write_intr, WAIT_CMD, NULL);  
    idedisk_output_data(drive, rq->buffer, SECTOR_WORDS);  
}  
    return ide_started;  
}  
printk(KERN_ERR "%s: bad command: %d\n", drive->name, rq->cmd);  
ide_end_request(0, HWGROUP(drive));  
return ide_stopped;  
}
```

10.5. Запросы к драйверу

Обработку запроса можно разделить на три фазы: предобработку, исполнение запроса и постобработку. Пользовательская программа запрашивает операцию, исполняя соответствующий системный вызов. В ОС семейства Unix это может быть, например, системный вызов `write(int file, void * buffer, size_t size)`.

Предобработка выполняется модулем системы, который, как правило, исполняется в нити процесса, сформировавшей запрос, но имеет привилегии ядра. Фаза предобработки включает в себя:

- проверку допустимости параметров. Пользователь должен иметь право выполнять запрошенную операцию над данным устройством, адрес буфера должен быть допустимым адресом пользовательского адресного пространства и т. д.;
- возможно, копирование или отображение данных из пользовательского адресного пространства в системное;
- возможно, преобразование выводимых данных. Например, в системах семейства Unix при выводе на терминал система может заменять символ горизонтальной табуляции на соответствующее число пробелов (если терминал не поддерживает горизонтальную табуляцию) и преобразовывать символ перевода строки. Дело в том, что внутри системы в качестве разделителя строк используется символ новой строки '\n' (ASCII NL), а различные модели терминалов и принтеров могут использовать также '\r' (ASCII RET, возврат каретки) или последовательности '\r"\n' или '\n"\r';
- возможно, обращение к процедурам драйвера. Эти процедуры могут блокировать код и данные драйвера в физической памяти и выделять буферы для ПДП. Эти операции реализуются нереентерабельными сервисами ядра и не всегда могут быть выполнены драйвером во время обработки запроса;
- передачу запроса драйверу. Некоторые системы реализуют передачу запроса как простой вызов соответствующей функции драйвера, но чаще используются более сложные асинхронные механизмы, которые будут обсуждаться далее.

Выполнив запрос, драйвер активизирует программу постобработки, которая анализирует результат операции, предпринимает те или иные действия по восстановлению в случае неудачи, копирует или отображает полученные данные в пользовательское адресное пространство и оповещает пользовательский процесс о завершении запроса.

Некоторые системы на этой фазе также производят преобразование введенных данных. В качестве примера можно вновь привести системы семейства

Unix, которые при вводе с терминала выполняют трансляцию символа перевода строки и ряд других операций редактирования, например, стирание последнего введенного символа по запросу пользователя. Разбиение потока терминальных данных на строки в этих системах также происходит на фазе постобработки.

В той или иной форме эти три фазы обработки запроса ввода/вывода присутствуют во всех многопоточных и даже многих однопоточных системах.

10.5.1. Синхронный ввод/вывод

Самым простым механизмом вызова функций драйвера был бы косвенный вызов соответствующих процедур, составляющих тело драйвера, подобно тому, как это делается в MS DOS и ряде других однозадачных систем.

В старых системах семейства Unix драйвер последовательного устройства исполняется в рамках той нити, которая сформировала запрос, хотя и с привилегиями ядра. Ожидая реакции устройства, драйвер переводит процесс в состояние ожидания доступными ему примитивами работы с планировщиком. В примере 10.1 это `interruptible_sleep_on`. В качестве параметра этой функции передается блок переменных состояния устройства, и в этом блоке сохраняется ссылка на контекст блокируемой нити.

Доступные прикладным программам функции драйвера исполняются в пользовательском контексте — в том смысле, что хотя драйвер и работает в адресном пространстве ядра, но при его работе определено и пользовательское адресное пространство, поэтому он может пользоваться примитивами обмена данными с ним (в примере 10.1 это `memcpy_from_fs`).

Обработчик прерывания, наоборот, работает в контексте прерывания, когда пользовательское адресное пространство не определено. Поэтому, чтобы при обслуживании прерывания можно было получить доступ к пользовательским данным, основная нить драйвера вынуждена копировать их в буфер в адресном пространстве ядра.

Синхронная модель драйвера очень проста в реализации, но имеет существенный недостаток, приведенный в примере 10.1, — драйвер нереентрабелен. Обращение двух нитей к одному устройству приведет к непредсказуемым последствиям (впрочем, для практических целей достаточно того, что среди возможных последствий числится нарушение целостности данных ядра и последующий вызов процедуры `panic()` с выдачей регистров на экран). Предсказуемость последствий обеспечивается включением в контекст устройства семафора, установкой этого семафора при входе в функцию `foo_write` и снятием его при выходе. Семафор имеет очередь ожидающих его процессов, и, таким образом, реентрантно (т. е. во время обработки предыду-

шего аналогичного запроса) приходящие запросы будут устанавливаться в очередь.

Альтернативный подход к организации ввода/вывода состоит в том, чтобы возложить работу по формированию очереди запросов не на драйвер, а на функцию предобработки запроса. При этом первый запрос к драйверу, какое-то время бывшему неактивным, может по-прежнему осуществляться в нити процесса, сформировавшего этот запрос, но все последующие запросы извлекаются из очереди fork-процессом драйвера при завершении предыдущего запроса. Такой подход называется асинхронным.

Примечание

Здесь возникает интересный вопрос: если запрос обрабатывается асинхронно, то обязана ли пользовательская программа ожидать окончания операции? Вообще говоря, не обязана, но этот вопрос подробнее будет обсуждаться в разд. 10.7.

10.5.2. Асинхронный ввод/вывод

В системах семейства Unix драйверы блочных устройств обязательно асинхронные. Кроме того, в современных версиях системы асинхронными драйверами являются драйверы потоковых устройств. Многие другие ОС, в том числе однозадачные (такие, как DEC RT-11), используют исключительно асинхронные драйверы.

Драйвер, применяющий асинхронную архитектуру, обычно предоставляет вместо отдельных функций `read`, `write`, `ioctl` и т. д. единую функцию, которая в системах семейства Unix называется `strategy`, а мы будем называть *стратегической функцией* (рис. 10.7).

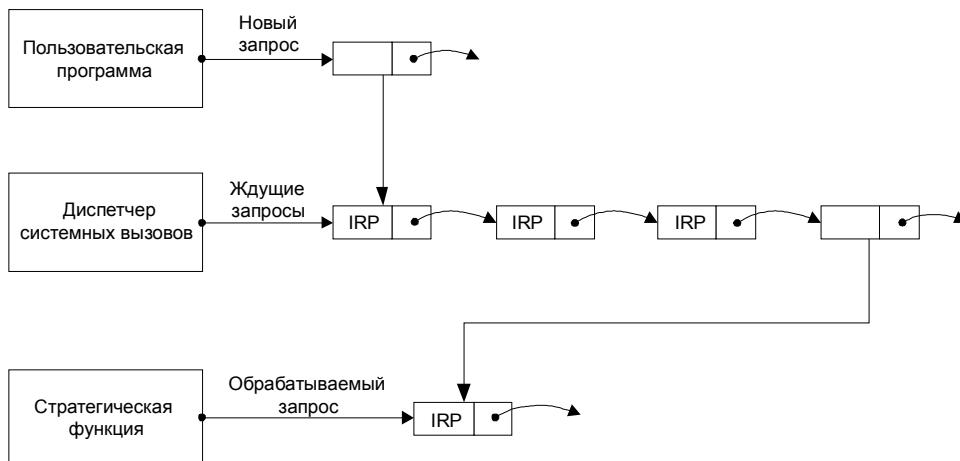


Рис. 10.7. Стратегическая функция и очередь запросов

Запросы к драйверу в VMS

В операционной системе VAX/VMS драйвер получает запросы на ввод/вывод из очереди запросов. Элемент очереди называется **IRP** (Input[Output] Request Packet — пакет запроса ввода/вывода). Обработав первый запрос в очереди, драйвер начинает обработку следующего. Операции над очередью запросов выполняются специальными командами процессора VAX и являются атомарными. Если очередь пуста, основная нить драйвера завершается. При появлении новых запросов система вновь запустит ее.

IRP содержит:

- код операции (чтение, запись или код SPFUN — специальная функция, подобная `ioctl` в системах семейства Unix);
- адрес блока данных, которые должны быть записаны, или буфера, куда данные необходимо поместить;
- информацию, используемую при постобработке, в частности, идентификатор процесса, запросившего операцию.

В зависимости от кода операции драйвер запускает соответствующую подпрограмму. В VAX/VMS адрес подпрограммы выбирается из таблицы FDT (Function Definition Table). Подпрограмма инициирует операцию и приостанавливает процесс, давая системе возможность выполнить другие активные процессы. Затем, когда происходит прерывание, его обработчик инициирует fork-процесс, исполняющий следующие этапы этого запроса. Завершив один запрос, fork-процесс сообщает об этом процедурам постобработки (разбудив соответствующий процесс) и, если в очереди еще что-то осталось, начинает выполнение следующего запроса.

В качестве параметра стратегическая функция получает указатель на структуру запроса, в которой содержится код требуемой операции и блок данных. При этом возникает сложный вопрос, а именно — в каком адресном пространстве размещается этот блок?

На первый взгляд, идеальным решением было бы размещение этого блока сразу в пользовательском адресном пространстве. Проблема здесь в том, что стратегическая функция — особенно при обработке не первого запроса в очереди — исполняется не в пользовательском контексте, когда можно применять примитивы обмена данными с адресным пространством задачи, а в контексте fork-процесса, а то и в контексте прерывания, когда адресное пространство пользователя не определено.

Возможны два варианта решения этой проблемы: хранить в структуре запроса также и указатель на пользовательское адресное пространство, либо все-таки копировать данные в адресное пространство системы на этапе предобработки, и обратно в пользовательское на этапе постобработки запроса. Драйвер в этом случае не должен беспокоиться ни о каком копировании, зато разработчик ОС получает дополнительную головную боль в виде логики управления буферами в адресном пространстве системы и выделения памяти для них.

Буферизация запросов и формирование очереди к блочным устройствам в Unix осуществляется специальным модулем системы, который называется *дисковым кэшем*. Принцип работы дискового кэша будет обсуждаться в разд. 10.7. Очереди запросов к потоковым устройствам имеют меньший объем, поэтому выделение памяти для них осуществляется обычным `kmalloc`.

10.6. Сервисы ядра, доступные драйверам

Следует провести различие между системными вызовами и функциями ядра, доступными для драйверов. Наборы системных вызовов и драйверных сервисов совершенно независимы друг от друга. Как правило, системные вызовы недоступны для драйверов, а драйверные сервисы — для пользовательских программ.

Системный вызов включает в себя переключение контекста между пользовательской программой и ядром. В системах с виртуальной памятью во время такого переключения процессор переходит из "пользовательского" режима, в котором запрещены или ограничены доступ к регистрам диспетчера памяти, операции ввода/вывода и ряд других действий, в "системный", в котором все ограничения снимаются. Обычно системные вызовы реализуются с использованием специальных команд процессора, чаще всего — команд программного прерывания.

Драйвер же исполняется в "системном" режиме процессора и, как правило, в контексте ядра, поэтому для вызова сервисов ядра драйверу не надо делать никаких переключений контекста. Практически всегда такие вызовы реализуются обычными командами вызова подпрограммы.

Еще одно важное различие состоит в том, что, исполняя системный вызов, программисту не надо заботиться о его реентерабельности: ядро либо обеспечивает подлинную реентерабельность, либо создает иллюзию реентерабельности благодаря тому, что исполняется с более высоким приоритетом, чем все пользовательские программы. Напротив, доступные драйверам сервисы ядра делятся на две группы — те сервисы, которые можно вызывать из обработчиков прерываний, и те, которые нельзя.

Сервисы, доступные для обработчиков прерываний, должны удовлетворять двум требованиям: они должны быть реентерабельными и завершаться за гарантированное время. Например, выделение памяти может потребовать сборки мусора или даже поиска жертвы для удаления в адресных пространствах пользовательских задач. Кроме того, выделение памяти требует работы с разделяемым ресурсом (пулью памяти ядра) и его достаточно сложно реализовать реентерабельным образом, поэтому обработчикам прерываний очень редко разрешают запрашивать память.

Копирование данных между пользовательским и системным адресными пространствами может привести к возникновению страничного отказа, время обработки которого может быть непредсказуемо большим.

Далее, для краткости, мы будем называть доступные для обработчиков прерываний сервисы реентерабельными, хотя для них важна не только реентерабельность, но и завершение в течение фиксированного времени. В предыдущих разделах мы упоминали некоторые категории сервисов, предоставляемых драйверам. Эти сервисы включают в себя (приведенный список не является исчерпывающим) следующие:

- взаимодействие с конфигурацией системы — доступ к данным средств автоконфигурации и, кроме того, регистрация драйвера и управляемых им устройств в системе;
- сбор и сохранение статистики;
- запросы на выделение и освобождение системных ресурсов, в первую очередь памяти;
- примитивы межпоточного взаимодействия — между нитями самого драйвера, между драйвером и другими нитями ядра и, наконец, между драйвером и нитями пользовательского процесса;
- таймеры;
- передача данных из пользовательского адресного пространства и обратно и другие операции над пользовательским адресным пространством;
- сервисные функции.

Некоторые из групп этих функций будут подробнее описаны далее.

10.6.1. Автоконфигурация

— В моем поле зрения появляется новый объект.
Возможно, ты шкаф?
— Нет.
— Возможно, ты стол?
— Нет.
— Каков твой номер?
— Шесть.
— Каков твой пол?
— Женский.
— Иду на вы!
— Иди.

Б. Гребеницков

Автоматическое определение установленных в системе устройств и их конфигурации экономит время и силы администратора при установке и перенастройке ОС. Разработчику ОС, впрочем, не следует полностью полагаться на

автоконфигурацию, особенно при широком спектре оборудования, поддерживаемого системой: ошибки бывают не только в программном обеспечении, но и в аппаратных устройствах, в том числе и в той части логики контроллера, которая обеспечивает его идентификацию и автонастройку.

Автонастройка возможна только при определенной поддержке со стороны аппаратуры. Такая поддержка может обеспечиваться несколькими способами:

- каждое устройство имеет фиксированные адреса регистров. Системная шина либо генерирует исключение по отсутствию адресуемого устройства, либо при чтении с несуществующего адреса возвращает фиксированное значение (чаще всего 0 или 0xFFFFFFFF). Во втором случае достаточно, чтобы один из регистров устройства после включения питания обязательно содержал значение, отличающееся от этого фиксированного. Функция `probe` драйвера обращается к регистрам этого устройства, и, прочитав правильное значение и не получив при этом ошибки шины, может сделать вывод, что устройство присутствует. Устройства, у которых нет драйверов, таким способом не могут быть обнаружены, но ОС все равно не сможет с ними работать. Этот метод плох тем, что трудно применим при большом числе изготовителей периферийных устройств и широкой номенклатуре этих устройств — конфликты между адресами устройств различных изготовителей практически неизбежны;
- каждое устройство имеет ПЗУ, которое географически отображается на адреса системной шины. После запуска загрузочный монитор сканирует все возможные адреса таких ПЗУ и исполняет хранящийся в найденных микросхемах код. Этот код регистрирует устройство в конфигурационной базе данных загрузочного монитора. ОС после загрузки обращается к этой базе. Данный метод плох тем, что применим, только если устройства подключаются к системам с бинарно совместимыми центральными процессорами;
- каждое устройство содержит набор конфигурационных регистров, обычно также адресуемых географически. Эти регистры содержат тот или иной уникальный идентификатор устройства и, возможно, сведения о его конфигурации. Сканирование этих регистров может осуществляться как самой системой, так и загрузочным монитором. Этот метод лишен недостатков двух предыдущих и широко применяется в большинстве современных периферийных шин, например в PCI.

Приведенные рассуждения справедливы не только для устройств, имеющих адреса на системнойшине, но и для устройств, подключаемых к шинам с собственной адресацией, например SCSI. Количество адресов шины SCSI невелико, поэтому подсистема автоконфигурации быстро может просканировать их все. Каждое устройство SCSI должно поддерживать команду `INQUIRY`, в ка-

честве ответа на которую обязано вернуть тип устройства и другую информацию, в частности, название изготовителя и модели, поддерживаемую версию протокола SCSI и др. Одна из форм команды `INQUIRY` позволяет также проверить, поддерживает ли устройство какую-либо другую команду из набора команд SCSI.

Одна из серьезных неприятностей, которая может возникнуть при автоконфигурации, — это конфликт устройств по каким-либо ресурсам, чаще всего — адресам всех или некоторых регистров; линии запроса или вектору прерывания. Если первый конфликт разрешим только перенастройкой аппаратуры (устройства с одним и тем же адресом на шине гарантированно неработоспособны, а в некоторых случаях могут сделать неработоспособной всю шину), то второй может быть обойден сугубо программными средствами.

Действительно, ничто не мешает драйверам конфликтующих устройств разделять один вектор прерывания. При приходе запроса прерывания по этому вектору вызывается один из обработчиков. Он анализирует состояние своего устройства и если обнаруживает, что прерывание вызвано им, обрабатывает его. Если же прерывание не его, он просто вызывает следующий обработчик.

Эта схема допускает каскадирование потенциально неограниченного количества обработчиков, с тем очевидным недостатком, что каждое дополнительное звено цепочки значительно увеличивает задержку прерывания для всех последующих драйверов.

Большинство современных ОС использует более сложный механизм обработки прерываний, когда пролог и эпилог прерывания исполняются сервисной функцией, предоставленной ядром системы, и уже эта функция вызывает собственно обработчик. Протокол обмена вызывающей функции с обработчиком может включать в себя код возврата: обработчик должен при этом сообщать, "его" это прерывание или не "его". В последнем случае необходимо вызвать следующий обработчик и т. д.

Эта схема также допускает неограниченное каскадирование и обладает тем же недостатком, что и предыдущая, а именно — увеличивает задержку для последних обработчиков в цепочке.

В любом случае, разделение векторов прерываний требует активной кооперации со стороны обработчиков этих прерываний.

10.6.2. Выделение памяти

Алгоритмы выделения памяти, в том числе и пригодные для использования в ядре ОС, подробно обсуждались в главе 4. Кроме того, мы уже упомянули тот печальный факт, что в контексте прерывания система обычно не позволяет запрашивать ресурсы. Важно остановиться еще на двух аспектах проблемы.

Во-первых, ядро ОС обычно размещается в физической памяти и не подвергается страничному обмену, поэтому память ядра представляет собой более дефицитный ресурс, чем виртуальная память, доступная прикладным программам. Разработчик модуля ядра должен иметь это в виду и не прибегать к экстравагантным схемам управления буферами, которые иногда применяются в прикладных программах. Управлению большими объемами буферов, в частности дисковым кэшем, в книге посвящен *разд. 10.7*.

Во-вторых, в ряде случаев драйвер не может удовлетвориться любым участком физического ОЗУ: например, некоторые старые контроллеры ПДП или периферийной шины не могут адресовать всю физическую память. Так, контроллер ПДП шины ISA имеет 24-разрядный адрес и способен, таким образом, адресовать только младшие 16 Мбайт ОЗУ (рис. 10.8). Некоторые контроллеры ПДП требуют выравнивания буфера, чаще всего на границу страницы.

Это является дополнительным доводом в пользу того, чтобы при обмене данными с внешним устройством копировать их в системный буфер, а не использовать непосредственно пользовательскую память, которая может быть размещена где угодно и с каким угодно выравниванием.

При выделении буферов для ПДП многие ОС позволяют задать ограничения, которым должен удовлетворять физический адрес выделенного буфера.

Не каждый требуемый драйверу ресурс может быть выделен немедленно. Многие ОС предоставляют драйверу callback-функции, вызываемые, когда требуемый ресурс становится доступен.

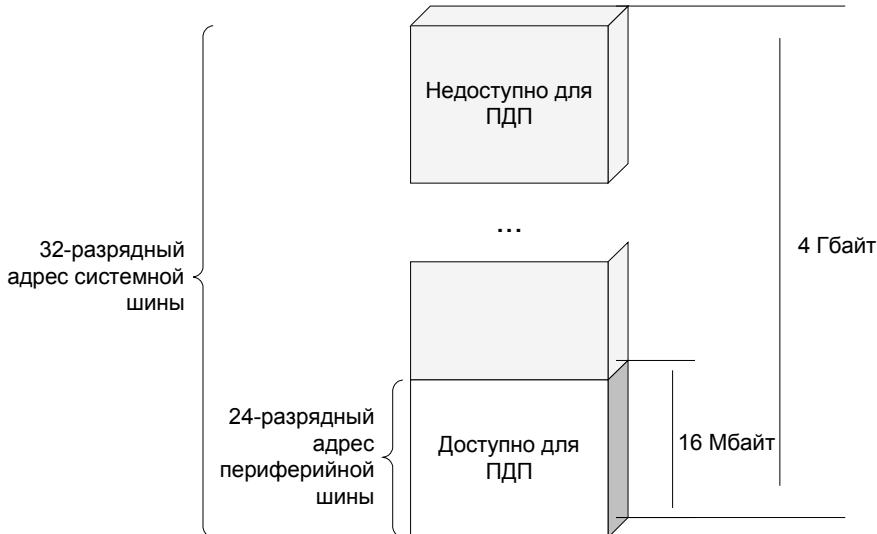


Рис. 10.8. Ограничения для буферов ПДП

10.6.3. Таймеры

Ядро обычно предоставляет два типа таймеров — часы реального времени, указывающие астрономическое время (драйверу это время обычно интересно только для сбора статистики), и собственно таймеры — механизмы, позволяющие отмерять интервалы времени.

Таймеры интересны драйверам с нескольких точек зрения. Один из важных способов их использования приведен в примере 10.1: если устройство из-за какой-либо ошибки не генерирует прерывания, наивный драйвер может остаться в состоянии ожидания навсегда. Чтобы этого не происходило, драйвер должен устанавливать будильник, который сообщит основному потоку, что устройство подозрительно долго не отвечает.

Таймеры используются также как альтернатива непрерывному опросу устройства при исполнении длительных операций, например сброса устройства, если использование прерываний почему-либо нежелательно или невозможно. Если говорить именно о сбросе, автору не известно ни одного устройства, которое генерировало бы прерывание при завершении этой операции.

10.6.4. Обмен данными с пользовательским процессом

Мы уже упоминали, что общение с пользовательским процессом допустимо только в одном из возможных контекстов нити ядра, а именно в пользовательском. Во всех остальных контекстах пользовательский процесс попросту не определен — точнее сказать, активный пользовательский процесс не совпадает с тем процессом, запрос которого в данный момент обрабатывается драйвером.

В этой ситуации вызов примитивов взаимодействия с этим процессом может привести к непредсказуемым результатам.

В ряде случаев вместо драйвера этот обмен осуществляют другие модули ядра, например процедуры пред- и постобработки запроса. Непосредственно с пользовательским адресным пространством драйвер должен общаться только в синхронной модели. В этой модели драйвер иногда оказывается вынужден решать и другие, казалось бы не свойственные ему, задачи.

Обработка сигналов драйвером в Unix

Так, в системах семейства Unix все операции ввода/вывода, а также все остальные операции, переводящие процесс в состояние ожидания, могут быть прерваны сигналом. Сигнал представляет собой примитив обработки исключений, отчасти похожий на аппаратное прерывание тем, что при обработке сигнала может быть вызвана предоставленная программистом функция-обработчик. Необработанный сигнал обычно приводит к принудительному завершению процесса.

Будучи прерван сигналом, системный вызов останавливает текущую операцию, и, если это была операция обмена данными, но данных передано не было, возвращает код ошибки `EINTR`, говорящий о том, что вызов был прерван и, возможно, операцию следует повторить. Код, делающий это, присутствует в примере 10.1.

Например, пользовательский процесс может использовать сигнал `SIGALARM` для того, чтобы установить свой собственный будильник, сигнализирующий, что операция над устройством исполняется подозрительно долго.

Если драйвер не установит своего будильника и не станет отрабатывать сигналы, посланные процессу, может возникнуть очень неприятная ситуация.

Дело в том, что в Unix все сигналы, в том числе и сигнал безусловного убийства `SIGKILL`, обрабатываются процедурой постобработки системного вызова. Если драйвер не передает управления процедуре постобработки, то и сигнал, соответственно, оказывается необработанным, поэтому процесс остается висеть.

Других средств, кроме посылки сигнала, для уничтожения процесса в системах семейства Unix не предусмотрено. Поэтому процесс, зависший внутри обращения к драйверу, оказывается невозможно прекратить ни изнутри, ни извне.

Я столкнулся с этим при эксплуатации многопроцессорной версии системы SCO Open Desktop 4.0. Система была снабжена лентопротяжным устройством,

подключаемым к внешней SCSI-шине. Из-за аппаратных проблем это устройство иногда "зависало", прекращая отвечать на запросы системы. Драйвер лентопротяжки иногда правильно отрабатывал это состояние как аппаратную ошибку, а иногда тоже впадал в ступор, не пробуждаясь ни по собственному будильнику, ни по сигналам, посланным другими процессами. (По имеющимся у меня сведениям, эта проблема специфична именно для многопроцессорной версии системы. По-видимому, это означает, что ошибка допущена не в драйвере, а в коде сервисных функций.) В результате процесс, обращавшийся в это время к ленте, также намертво зависал, и от него нельзя было избавиться.

Из-за наличия неубиваемого процесса оказывалось невозможno выполнить нормальное закрытие системы; в частности, не получалось размонтировать файловые системы, где зависший процесс имел открытые файлы. Выполнение холодной перезагрузки системы с неразмонтированными файловыми томами приводило к неприятным последствиям для этих томов. Одна из аварий, к которым это привело, подробно описывается в разд. 11.4.2.

10.6.5. Сервисные функции

Набор сервисных функций, доступных драйверу, обычно представляет собой подмножество стандартной библиотеки того языка высокого уровня, на котором обычно пишутся драйверы. В большинстве современных ОС это С.

При выборе этого подмножества используется простой критерий: удаляются или заменяются на более или менее ограниченные все функции, которые так или иначе содержат в себе системные вызовы. Так, функции `memcp` или `sprint` вполне можно оставить, `malloc` придется заменить на эквивалент (в ядре Linux эта функция называется `kmalloc`), а `fwrite` драйверу вряд ли понадобится, особенно если учесть, что работа многих драйверов начинается до того, как будет смонтирована хоть одна файловая система.

Важную роль среди сервисных функций занимают операции, часто исполняемые одной командой, но такой, которую компиляторы ЯВУ в обычных условиях не генерируют. Это, прежде всего, операции обращения к регистрам ввода/вывода в машинах с отдельным адресным пространством ввода/вывода, а также команды разрешения и запрещения прерываний. На С такие операции реализуются в виде макроопределений, содержащих ассемблерную вставку.

Правила кодирования драйверов во многих ОС требуют, чтобы даже на машинах с единственным адресным пространством обращения к регистрам устройств происходили посредством макросов. Такой код может быть легко портирован на процессор с отдельным адресным пространством. В наше время, когда одни и те же устройства и одни и те же периферийные шины подключаются к различным процессорам, портирование драйверов между различными процессорами осуществляется весьма часто.

При портировании драйвера разработчик должен также принимать во внимание различия в порядке байтов устройства и текущего процессора. Для приведения этого параметра в соответствие обычно предоставляются функции или макросы перестановки байтов как в одном слове, так и в блоках значительного размера.

10.7. Асинхронная модель ввода/вывода с точки зрения приложений

В разд. 10.5.1, обсуждая асинхронную модель драйвера, мы задались вопросом: должна ли задача, сформировав запрос на ввод/вывод, дожидаться его завершения? Ведь система, приняв запрос, передает его асинхронному драйверу, который инициирует операцию на внешнем устройстве и освобождает процессор. Сама система не ожидает завершения запроса. Так должна ли пользовательская задача ожидать его?

Если было запрошено чтение данных, то ответ, на первый взгляд, очевиден: должна. Ведь если данные запрошены, значит, они сейчас будут нужны программе.

Однако можно выделить буфер для данных, запросить чтение, потом некоторое время заниматься чем-то полезным, но не относящимся к запросу, и лишь в точке, когда данные действительно будут нужны, спросить систему: а готовы ли данные? Если готовы, то можно продолжать работу. Если нет, то придется ждать (рис. 10.9).

В многих приложениях, особенно интерактивных или работающих с другими устройствами-источниками событий, асинхронное чтение оказывается единственным приемлемым вариантом, поскольку оно позволяет задаче одновременно осуществлять обмен с несколькими источниками данных и таким образом повысить пропускную способность и/или улучшить время реакции на событие.

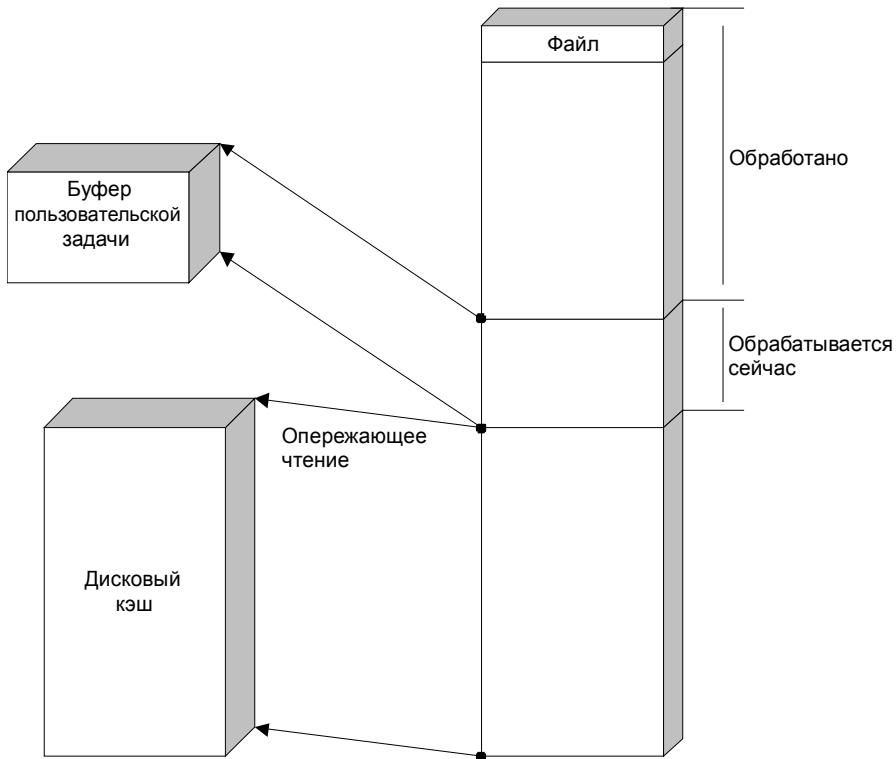


Рис. 10.9. Опережающее чтение

При записи, казалось бы, нет необходимости дожидаться физического завершения операции. При этом мы получаем режим, известный как *отложенная запись* (*lazy write* — "ленивая" запись, если переводить дословно). Однако такой режим создает две специфические проблемы.

Во-первых, программа должна знать, когда ей можно использовать буфер с данными для других целей. Если система копирует записываемые данные из пользовательского адресного пространства в системное, то эта же проблема возникает внутри ядра; внутри ядра проблема решается использованием многобуферной схемы, и все относительно просто. Однако копирование приводит к дополнительным затратам времени и требует выделения памяти под буферы. Наиболее остро эта проблема встает при работе с дисковыми и сетевыми устройствами, с которыми система обменивается большими объемами данных (а сетевые устройства еще и могут генерировать данные неожиданно). Проблема управления дисковыми буферами подробнее обсуждается в разд. 10.7.

В большинстве современных вычислительных систем общего назначения накладные расходы, обусловленные буферизацией запросов, относительно невелики или, по крайней мере, считаются приемлемыми. Но в системах реального времени и/или встраиваемых контроллерах, где время и объем оперативной памяти жестко ограничены, эти расходы оказываются серьезным фактором.

Если же вместо системных буферов используется отображение данных в системное адресное пространство (системы с открытой памятью можно считать вырожденным случаем такого отображения), то ситуация усложняется. Пользовательская задача должна иметь возможность узнать о физическом окончании записи, потому что только после этого буфер действительно свободен. Фактически, программа должна самостоятельно реализовать многобуферную схему или искать другие выходы.

Во-вторых, программа должна дождаться окончания операции, чтобы узнать, успешно ли она закончилась. Часть ошибок, например, попытку записи на устройство, физически не способное выполнить такую операцию, можно отловить еще во время предобработки, однако аппаратные проблемы могут быть обнаружены только на фазе исполнения запроса.

Многие системы, реализующие отложенную запись, при обнаружении аппаратной ошибки просто устанавливают флаг ошибки в блоке управления устройством. Программа предобработки, обнаружив этот флаг, отказывается выполнять следующий запрос. Таким образом, прикладная программа считает ошибочно завершившийся запрос успешно выполнившимся и обнаруживает ошибку лишь при одной из следующих попыток записи, что не совсем правильно.

Иногда эту проблему можно игнорировать. Например, если программа встречает одну ошибку при записи, то все исполнение программы считается неуспешным и на этом заканчивается. Однако во многих случаях, например, в задачах управления промышленным или исследовательским оборудованием, программе необходимо знать результат завершения операции, поэтому простая отложенная запись оказывается совершенно неприемлемой.

Так или иначе, ОС, реализующая асинхронное исполнение запросов ввода/вывода, должна иметь средства сообщить пользовательской программе о физическом окончании операции и результате этой операции.

Синхронный и асинхронный ввод/вывод в RSX-11 и VMS

Например, в системах RSX-11 и VAX/VMS фирмы DEC для синхронизации используется флаг локального события (local event flag). Как говорилось в разд. 7.3.1, флаг события в этих системах представляет собой аналог двоичных семафоров Дейкстры, но с ним также может быть ассоциирована процедура

AST. Системный вызов ввода/вывода в этих ОС называется `QIO` (Queue Input/Output [Request] — установить в очередь запрос ввода/вывода) и имеет две формы: асинхронную `QIO` и синхронную `QIOW` (Queue Input/Output and Wait — установить запрос и ждать [завершения]). С точки зрения подсистемы ввода/вывода эти вызовы ничем не отличаются, просто при запросе `QIO` ожидание конца запроса выполняется пользовательской программой "вручную", а при `QIOW` выделение флага события и ожидание его установки делается системными процедурами пред- и постобработки.

В ряде систем реального времени, например, в OS-9 и RT-11, используются аналогичные механизмы.

Напротив, большинство современных ОС общего назначения не связываются с асинхронными вызовами и предоставляют прикладной программе чисто синхронный интерфейс, тем самым вынуждая ее ожидать конца операции.

Возможно, это объясняется идеальным влиянием ОС Unix. Набор операций ввода-вывода, реализованных в этой ОС, стал общепризнанным стандартом де-факто и основой для нескольких официальных стандартов. Например, набор операций ввода/вывода в MS DOS является прямой копией Unix; кроме того, эти операции входят в стандарт ANSI на системные библиотеки языка C и стандарт POSIX.

Современные системы семейства Unix разрешают программисту выбирать между отложенной записью по принципу Fire And Forget (выстрелил и забыл) и полностью синхронной, используя вызов `fcntl` с соответствующим кодом операции.

Если все-таки нужен более детальный контроль над порядком обработки запросов, разработчикам предлагается самостоятельно имитировать асинхронный обмен, создавая для каждого асинхронно исполняемого запроса свою нить. Эта нить тем или иным способом сообщает основной нити о завершении операции. Для этого чаще всего используются штатные средства межпоточного взаимодействия — семафоры и др. Грамотное использование нитей позволяет создавать интерактивные приложения с очень высоким субъективным временем реакции, но за это приходится платить усложнением логики программы. В Windows NT/2000/XP существует средство организации асинхронного ввода/вывода — так называемые *порты завершения* [операции] (*completion ports*). Однако это средство не поддерживается в Windows 95, поэтому большинство разработчиков избегают использования портов завершения.

Синхронная модель ввода/вывода проста в реализации и использовании и, как показал опыт систем семейства Unix и его идеальных наследников, вполне адекватна большинству приложений общего назначения. Однако, как уже было показано, она не очень удобна (а иногда и просто непригодна) для задач реального времени. Следует также иметь в виду, что имитация асинхронного

обмена с помощью нитей не всегда допустима: асинхронные запросы исполняются в том порядке, в котором они устанавливались в очередь, а порядок получения управления нитями в многозадачной среде не гарантирован. Поэтому, например, стандарт POSIX.4 требует поддержки наравне с нитями и средств асинхронного обмена с вызовом callback при завершении операции.

10.8. ДИСКОВЫЙ КЭШ

Функции и принципы работы дискового кэша существенно отличаются от общих алгоритмов кэширования, обсуждавшихся в разд. 5.5. Дело в том, что характер обращения к файлам обычно существенно отличается от обращений к областям кода и данных задачи. Например, компилятор С и макропроцессор ТЕХ рассматривают входные и выходные файлы как потоки данных. Входные файлы прочитываются строго последовательно и полностью, от начала до конца. Аналогично, выходные файлы полностью перезаписываются, и перезапись тоже происходит строго последовательно. Попытка выделить аналог рабочей области при таком характере обращений обречена на провал независимо от алгоритма, разве что рабочей областью будут считаться все входные и выходные файлы.

Тем не менее кэширование или, точнее, буферизация данных при работе с диском имеет смысл и во многих случаях может приводить к значительному повышению производительности системы. Если отсортировать механизмы повышения производительности в порядке их важности, мы получим следующий список:

1. Размещение в памяти структур файловой системы — каталогов, FAT или таблицы инодов (эти понятия подробнее обсуждаются в главе 11) и т. д. Это основной источник повышения производительности при использовании дисковых кэшей под MS/DR DOS.
2. Отложенная запись. Само по себе откладывание записи не повышает скорости обмена с диском, но позволяет более равномерно распределить по времени загрузку дискового контроллера.
3. Группировка запросов на запись. Система имеет пул буферов отложенной записи, который и называется дисковым кэшем. При поступлении запроса на запись система выделяет буфер из этого пула и ставит его в очередь к драйверу. Если за время нахождения буфера в очереди в то же место на диске будет произведена еще одна запись, система может дописать данные в имеющийся буфер вместо установки в очередь второго запроса. Это значительно повышает скорость, если запись происходит массивами, не кратными размеру физического блока на диске.

4. Собственно кэширование. После того как драйвер выполнил запрос, буфер не сразу используется повторно, поэтому какое-то время он содержит копию записанных или прочитанных данных. Если за это время произойдет обращение на чтение соответствующей области диска, система может отдать содержимое буфера вместо физического чтения.
5. Опережающее считывание. При последовательном обращении к данным чтение из какого-либо блока значительно повышает вероятность того, что следующий блок также будет считан. Теоретически опережающее чтение должно иметь тот же эффект, что и отложенная запись, т. е. обеспечивать более равномерную загрузку дискового канала и его работу параллельно с центральным процессором. На практике, однако, часто оказывается, что считанный с опережением блок оказывается никому не нужен, поэтому эффективность такого чтения заметно ниже, чем у отложенной записи.
6. Сортировка запросов по номеру блока на диске. По идеи, такая сортировка должна приводить к уменьшению времени позиционирования головок чтения/записи (см. разд. 9.6.1). Кроме того, если очередь запросов будет отсортирована, это облегчит работу алгоритмам кэширования, которые производят поиск буферов по номеру блока.

Кэширование значительно повышает производительность дисковой подсистемы, но создает ряд проблем, причем некоторые из них довольно неприятного свойства.

Первая из проблем — та же, что и у отложенной записи. При использовании отложенной записи программа не знает, успешно ли завершилась физическая запись. При работе с дисками один из основных источников ошибок — физические ошибки диска. Однако многие современные файловые системы поддерживают так называемый *hotfixing* (горячую починку) — механизм, обеспечивающий динамическую замену "плохих" логических блоков на "хорошие", что в значительной мере компенсирует эту проблему.

Вторая проблема гораздо серьезнее и тоже свойственна всем механизмам отложенной записи: если в промежутке между запросом и физической записью произойдет сбой всей системы, то данные будут потеряны. Например, пользователь сохраняет отредактированный файл и, не дождавшись окончания физической записи, выключает питание — содержимое файла оказывается потерянно или повреждено. Другая ситуация, до боли знакомая всем пользователям DOS/Windows 3.x/ 95: пользователь сохраняет файл, и в это время система зависает — результат тот же. Аналогичного результата можно достичь, не вовремя достав дискету или другой удалляемый носитель из привода (чтобы избежать этого, механика многих современных дисководов позволяет программно заблокировать носитель в приводе).

Очень забавно наблюдать, как пользователь, хотя бы раз имевший неприятный опыт общения с дисковым кэшем SMARTDRV, копирует данные с чужого компьютера на дискету. Перед тем как извлечь ее из дисковода, он оглядывается на хозяина машины и с опаской спрашивает: "У тебя там никаких кэшей нет?" В эпоху MS DOS мне доводилось наблюдать такое поведение у нескольких десятков людей.

Если откладывается запись не только пользовательских данных, но и модифицированных структур файловой системы, ситуация еще хуже: системный сбой может привести не только к потере данных, находившихся в кэше, но и к разрушению файловой системы, т. е. в худшем случае, к потере всех данных на диске.

Методы обеспечения целостности данных при системном сбое подробнее обсуждаются в *разд. 11.4*. Находившиеся в кэше данные при фатальном сбое гибнут всегда, но существуют способы избежать повреждения системных структур данных на диске без отказа от использования отложенной записи.

Третья проблема, связанная с дисковым кэшем, — это выделение памяти под него. Уменьшение кэша приводит к снижению производительности дисковой подсистемы, увеличение же кэша отнимает память у пользовательских процессов. В системах с виртуальной памятью это может привести к увеличению дисковой активности за счет увеличения объема подкачки, что ведет к снижению как дисковой, так и общей производительности системы. Перед администратором системы встает нетривиальная задача: найти точку оптимума. Положение этой точки зависит от следующих параметров:

- объема физической памяти;
- скорости канала обмена с диском;
- скорости центрального процессора;
- суммы рабочих наборов программ, исполняющихся в системе;
- интенсивности и характера обращений к диску.

При этом зависимость количества страничных отказов от объема памяти, доступной приложениям, имеет существенно нелинейный вид. Это же утверждение справедливо для связи между размером дискового кэша и соответствующей экономией обращений к диску. Таким образом, задача подбора оптимального размера кэша — это задача нелинейной оптимизации. Самое неприятное, что ключевой исходный параметр — характер обращений к диску — не количественный, а качественный; точнее сказать, его можно изменить лишь с помощью очень большого числа независимых количественных параметров.

Во многих ситуациях невозможно теоретически оценить положение оптимальной точки, и единственным способом оказывается эксперимент: прогон

типичной для данной машины смеси заданий при различных объемах кэша. При этом нужно иметь возможность различать дисковую активность, связанную с обращениями к файлам и со страничным обменом. Большинство современных ОС предоставляют для этой цели различные инструменты системного мониторинга. Чаще, однако, объем кэша выставляется на глаз, а к дополнительной настройке прибегают, только если производительность оказывается слишком низкой.

Возникает вполне естественное желание возложить подбор размера кэша на саму систему, т. е. менять размер кэша динамически в зависимости от рабочей нагрузки. Кроме упрощения работы администратора, такое решение имеет еще одно большое преимущество: система начинает "автомагически" подстраиваться под изменения нагрузки.

Но далеко не все так просто. Если объем памяти в системе превосходит потребности прикладных программ, то динамический дисковый кэш может формироваться по очень простому "остаточному" принципу — все, что не пригодилось приложениям, отдается под кэш. Однако оперативная память до сих пор относительно дорога и представляет собой дефицитный ресурс, поэтому наибольший практический интерес представляет ситуация, когда памяти не хватает даже приложениям, не говоря уже о кэше. Тем не менее и в этой ситуации кэш некоторого объема бывает нужен.

Разумной политикой была бы подстройка кэша в зависимости от количества страничных отказов: если число отказов становится слишком большим, система уменьшает кэш; если же число отказов мало, а идут интенсивные обращения к диску, система увеличивает кэш. Получается саморегулирующаяся система с отрицательной обратной связью. Однако, если вдуматься, то видно, что вместо одной произвольной переменной (объема статического кэша) мы вынуждены ввести как минимум три:

- количество страничных отказов, которое считается слишком большим;
- количество отказов, которое считается достаточно малым;
- величину, на которую следует увеличить или уменьшить кэш в этих случаях.

На практике часто также вводятся параметры, ограничивающие минимальный и максимальный размеры кэша.

Оптимальные значения этих переменных зависят практически от тех же самых параметров, что и объем статического кэша, но подбор значений экспериментальным путем оказывается значительно сложнее, потому что вместо одномерной нелинейной оптимизации мы вынуждены заниматься трехмерной нелинейной оптимизацией.

Кроме того, читатель, знакомый с теорией управления, должен знать, что неудачный подбор параметров у системы с отрицательной обратной связью может приводить к колебательному процессу вместо саморегуляции. В дискуссиях USENET news приводились примеры развития таких колебаний в динамическом кэше системы Windows NT при компиляции большого проекта в условиях недостатка памяти.

Вполне возможно, что низкая производительность Windows NT/2000/XP на машинах с небольшим количеством памяти объясняется вовсе не низким качеством реализации и даже не секретным говором между фирмой Microsoft и производителями оперативной памяти, а просто плохо сбалансированным динамическим кэшем.

10.9. Спулинг

Гигабайт тебе в спул.

Популярное ругательство

Термин *спулинг* (*spooling*) не имеет общепринятого русского аналога. В соответствии с программистским фольклором, слово это происходит от аббревиатуры Simultaneous Peripheral Operation Off-Line. Эту фразу трудно дословно перевести на русский язык; имеется в виду метод работы с внешними устройствами вывода (реже — ввода) в многозадачной ОС или многомашинной среде, при котором задачам создается иллюзия одновременного доступа к устройству. При этом, однако, задачи не получают к устройству прямого доступа, а работают в режиме *offline* (без прямого подключения). Выводимые данные накапливаются системой, а затем выводятся на устройство так, чтобы вывод различных задач не смешивался.

Видно, что этот метод работы отчасти напоминает простую отложенную запись, но основная задача здесь не только и не столько повышение производительности, сколько разделение доступа к медленному внешнему устройству.

Чаще всего спулинг применяется для работы с печатающими устройствами, а для промежуточного хранения данных используется диск. Многие почтовые системы применяют механизм, аналогичный спулингу: если получатель не готов принять письмо, или линия связи с получателем занята либо вообще разорвана, предназначеннное к отправке письмо помещается в очередь. Затем, когда соединение будет установлено, письмо отправляется.

Классический спулинг реализован в ОС семейства Unix. В этих ОС вывод задания на печать осуществляется командой `lpr`. Эта команда копирует предназначенные для печати данные в каталог `/usr/spool/lp`, возможно, пропуская их при этом через программу-фильтр. Каждая порция данных помещается в отдельный файл. Имена файлов генерируются так, чтобы имя каждого вновь

созданного файла было "больше" предыдущего при сравнении ASCII-кодов. За счет этого файлы образуют очередь.

Системный *процесс-демон* (*daemon*) `lpd` (или `lpshed` в Unix System V) периодически просматривает каталог. Если там что-то появилось, а печатающее устройство свободно, демон копирует появившийся файл на устройство. По окончании копирования он удаляет файл, тем или иным способом уведомляет пользователя об окончании операции (в системах семейства Unix чаще всего используется электронная почта) и вновь просматривает каталог. Если там по-прежнему что-то есть, демон выбирает первый по порядку запрос и также копирует его на устройство.

Тот же механизм используется почтовой системой Unix — программой `sendmail`, только вместо каталога `/usr/spool/lp` используется `/usr/spool/mail`.

Этот механизм очень прост, но имеет один специфический недостаток: демон не может непосредственно ожидать появления файлов в каталоге, как можно было бы ожидать установки семафора или другого флага синхронизации. Если бы демон непрерывно сканировал каталог, это создавало бы слишком большую и бесполезную нагрузку системы. Поэтому демон пробуждается через фиксированные интервалы времени; если за это время ничего в очереди не появилось, демон засыпает вновь. Такой подход также очень прост, но увеличивает время прохождения запросов: запрос начинает исполняться не сразу же после установки, а лишь после того, как демон в очередной раз проснется.

В OS/2 и Win32 спулинг организован отчасти похожим образом с той разницей, что установка запроса в очередь может происходить не только командой `PRINT`, но и простым копированием данных на псевдоустройство `LPT[1-9]`. В отличие от систем семейства Unix как программа `PRINT`, так и псевдоустройства портов активизируют процесс спулинга непосредственно при установке запроса. Графические драйверы печатающих устройств в этих системах также используют спул вместо прямого обращения к физическому порту.

Novell Netware предоставляет специальный механизм для организации спулинга — очереди запросов. Элементы очереди в этом случае также хранятся на диске, но прикладные программы вместо просмотра каталога могут пользоваться системными функциями `GetNextMessage` и `PutMessage`. Вызов `GetNextMessage` блокируется, если очередь пуста; таким образом, нет необходимости ожидать пробуждения демона или специальным образом активизировать его — демон сам пробуждается при появлении запроса. Любопытно, что почтовая система Mercury Mail для Novell Netware может использовать для промежуточного хранения почты как очередь запросов, так и выделенный каталог в зависимости от конфигурации.

Вопросы для самопроверки

1. Перечислите причины, по которым желательно работать с устройствами через драйверы.
2. Почему считается невозможным реализовать универсальные драйверы, которые могли бы использоваться в различных ОС?
3. Как вы думаете, какие преимущества могла бы дать запись драйверов в ПЗУ, установленное на плате контроллера устройства? Какие недостатки у такого решения? Все ли преимущества и недостатки перечислены в книге? В частности, нельзя ли было бы использовать такие ПЗУ для внедрения троянских программ в систему, к которой подключается устройство?
4. Перечислите основные функции драйвера в системах семейства Unix. Если вы знакомы с интерфейсом драйвера каких-либо других ОС, постараитесь найти пересечения и отличия.
5. Почему многие ОС используют драйверы разных типов с различными интерфейсами?
6. Для чего может быть необходимо использование многоуровневых драйверов? Попробуйте привести примеры ситуаций, в которых многоуровневые драйверы полезны или необходимы, и которые не упомянуты в этой книге.
7. Опишите типичную архитектуру драйвера периферийного устройства.
8. Какие преимущества дает асинхронное исполнение запросов ввода/вывода? Какие недостатки?
9. В каких пределах можно реализовать асинхронное исполнение запросов ввода/вывода, сохраняя синхронный интерфейс у соответствующих системных вызовов?



ГЛАВА 11

Файловые системы

Работа программиста состоит в том, чтобы в определенном порядке намагнитить участки на поверхности врачающегося диска.

Одним из первых внешних устройств после клавиатуры и телевизора, которые перечисляются в любом руководстве по персональным компьютерам для начинающих, является магнитный диск. Вместо магнитного диска в наше время может использоваться и какая-то другая энергонезависимая память, например, флэш (ЭСППЗУ с блочным стиранием) или файловый сервер, но наличие такой памяти является очень важным. Ведь вы же не будете набирать вашу программу каждый раз при новом включении компьютера. Правда, на 16-разрядных машинах такое еще было возможным; мне доводилось слышать легенды о людях, которые могли по памяти набрать на консольном мониторе PDP-11 тетрис. Но для современных прикладных программ, размеры загрузочных модулей которых измеряются сотнями мегабайт, это невозможно.

Понятно также, что недостаточно иметь возможность просто запомнить программу и данные. Ведь вы можете работать с несколькими программами или над несколькими проектами одновременно. Ясно, что записывать на бумажке, в какое место вашей энергонезависимой памяти вы что-то сохранили, по меньшей мере неудобно. Поэтому естественно желание создать специализированную программу, которая будет как-то структурировать сохраненные данные. Именно эту работу по структурированию пользовательских данных берет на себя модуль ОС, называемый файловым менеджером. Дисковые операционные системы (ДОС) состоят по сути только из файлового менеджера и загрузчика бинарных модулей.

Понятие файла вводится в самом начале любого курса компьютерного ликбеза, но мало в каком курсе дается его внятное определение. Слово *файл* (*file*) дословно переводится с английского как папка или подшивка, но такой пере-

вод почти не добавляет ясности. Одно из наилучших определений, известных автору, звучит так: "Файл — это совокупность данных, доступ к которой осуществляется по ее имени".

Файл, таким образом, противопоставляется другим объектам, доступ к которым осуществляется по их адресу, например, записям внутри файла или блокам на диске.

Примечание

ОС семейства Unix трактуют понятие файла более широко — там файлом называется любой объект, имеющий имя в файловой системе. Однако файлы, не являющиеся совокупностями данных (каталоги, внешние устройства, псевдоустройства, именованные программные каналы, семафоры Xenix), часто называют не простыми файлами, а "специальными".

Из разд. 9.6 нам известно, что магнитный диск или другое устройство памяти (кроме, пожалуй, файлового сервера) чаще всего организует доступ к данным не по их именам, а все-таки по адресам, например, по номеру сектора, дорожки и поверхности диска. Поэтому, если система хочет предоставлять доступ по именам, она должна хранить таблицу преобразования имен в адреса — *директорию* (*directory*) или, как чаще говорят по-русски, *каталог*. В каталоге хранится имя файла и другая информация о файле, такая как его размер и местоположение на диске. Как правило, хранят также дату создания файла, дату его последней модификации, а в многопользовательских системах — идентификатор хозяина этого файла и права доступа к нему для других пользователей. Во многих файловых системах эта информация хранится не в самом каталоге, а в специальной структуре данных — *иноде*, *метафайле* и т. д. В этом случае запись в каталоге содержит только имя и указатель на управляющую структуру файла.

Большинство современных операционных систем позволяет делать вложенные каталоги — файлы, которые сами являются каталогами. В таких системах файл задается *полным* или *путевым именем* (*path name*), состоящим из цепочки имен вложенных каталогов и имени файла в последней из них.

Совокупность каталогов и других *метаданных*, т. е. системных структур данных, отслеживающих размещение файлов на диске и свободное дисковое пространство, называется *файловой системой* (ФС). Иногда на диске размещается только одна файловая система. Современные ОС часто позволяют размещать на одном физическом диске несколько файловых систем, выделяя каждой из них фиксированную непрерывную часть диска. Такие части диска называются *разделами* (*partition*) или *слайсами* (*slice*). Обычно разбиение диска на части производится на уровне драйвера диска, поэтому общее название частей — *логические диски*. С другой стороны, многие современные ОС могут объединять несколько логических дисков в один. Обычно это дела-

ется не на уровне ФС и даже не на уровне драйвера, а с помощью промежуточного слоя. В ряде современных ОС — в OS/2, Linux, Solaris — этот слой называется LVM (Logical Volume Manager — менеджер логических томов). При этом файловая система может оказаться распределена по нескольким физическим дискам или по несмежным участкам одного.

Далее мы будем называть пространство, занимаемое файловой системой, *томом* (volume).

11.1. Файлы с точки зрения пользователя

Прежде чем рассматривать структуры файловых систем, давайте сначала выясним, какие же операции над файлами и их именами обычно предоставляются. По аналогии с адресным пространством, иногда употребляют термин *пространство имен*, характеризующий совокупность всех допустимых имен файлов. Структура пространства имен зависит как от операционной, так и от файловой системы. Структура каталогов ФС накладывает ограничения на длину имен файлов и символы, которые могут употребляться в именах. ОС может устанавливать собственное ограничение на длину имени файла.

Как правило, ограничения на длину имени на уровне ОС обусловлены требованием совместимости со старым программным обеспечением: если в спецификациях системных вызовов сказано, что имя файла не может содержать более 12 символов, разработчики программного обеспечения будут выделять под буфер для хранения имени файла именно столько места. Такая программа без перекомпиляции не сможет работать с именами файлов большей длины и с каталогами, содержащими такие имена.

11.1.1. Монтирование файловых систем

Прежде чем ОС сможет использовать файловую систему, она должна выполнить над этой системой операцию, называемую *монтированием* (*mount*). В общем случае операция монтирования включает следующие шаги:

1. Проверку типа монтируемой ФС.
2. Проверку целостности ФС.
3. Считывание системных структур данных и инициализацию соответствующего модуля файлового менеджера (драйвера файловой системы).
4. В некоторых случаях — модификацию ФС с тем, чтобы указать, что она уже смонтирована. При этом устанавливается так называемый *флаг загрязнения* (*dirty flag*). Смысл этой операции будет объяснен в разд. 11.4.
5. Включение новой файловой системы в общее пространство имен (рис. 11.1). В различных системах это делается разными способами.

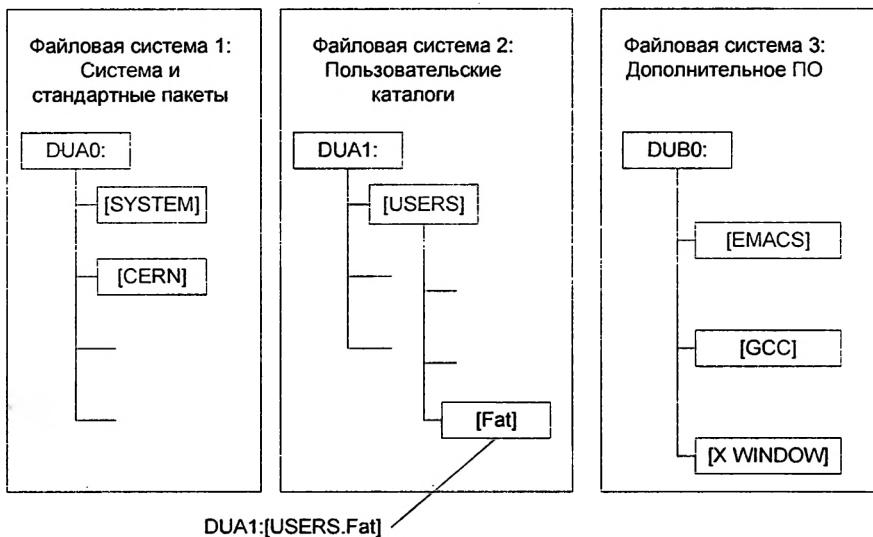


Рис. 11.1. Пространство имен ОС с несколькими ФС

Многие пользователи MS/DR DOS никогда не сталкивались с понятием монтирования. Дело в том, что эта система (как и многие другие ДОС, например RT-11) выполняет упрощенную процедуру монтирования при каждом обращении к файлу. Упрощения состоят в пропуске шагов 1 и 2 и отсутствии шага 4 (ФС MS/DR DOS устойчива к сбоям).

ДОС, как правило, помещают в пространство имен все доступные блочные устройства, не выполняя полной процедуры монтирования. Если какое-то из этих устройств не содержит ФС известного типа, то система будет возмущаться при обращениях к такому устройству, но не удалит его из списка доступных ФС. А иногда даже не будет возмущаться — попробуйте поставить в дисковод машины под управлением MS/DR DOS дискету, не содержащую файловой системы (например, созданную программой tar), и в командной строке набрать `DIR A:`. Скорее всего, вы увидите несколько экранов мусора, но ни одного сообщения об ошибке!

Если мы монтируем ФС, размещенную на удаленной машине (*файловом сервере*), то шаги 1 и 2 заменяются на установление соединения этим сервером. В системах семейства CP/M при работе с файловыми серверами Novell Netware монтирование серверных файловых систем производится командой `MAP`, а с файловыми серверами, поддерживающими протокол SMB, — командой `NET USE`.

Обычно имя файла в подмонтированной файловой системе имеет вид `ИМЯ_ФС:имена\каталогов\имя.файл`. При этом вместо разделителей `:` и `\` могут использоваться другие символы.

Имена файловых систем в RT-11, RSX-11, VMS

В RT-11, RSX-11 и VMS в качестве имени файловой системы используется имя физического устройства, на котором размещена ФС. Если применяется DECNet, перед именем устройства можно поместить имя узла сети, на котором это устройство находится. Полное имя локального файла в VMS выглядит так: DUA0:[USERS.FAT_BROTHER.WORK]test.exe. При этом DUA0 означает дисковое устройство 0, присоединенное к дисковому контроллеру A: Disk Unit A [device #] 0, [USERS.FAT_BROTHER.WORK] означает каталог WORK в каталоге FAT_BROTHER в каталоге USERS.

Имена файловых систем в ОС семейства CP/M

В системах семейства CP/M имена файловых систем обозначаются буквами латинского алфавита, а сами файловые системы часто почему-то называются "драйвами". При некотором желании можно использовать в качестве имен ФС также символы 'Г' и 'Г'. Устройства А: и В: — это всегда приводы гибких дисков, устройство С: — обычно первый жесткий диск или первый раздел на первом жестком диске.

Меня всегда интересовал вопрос: "Что будет делать пользователь, когда у него кончатся доступные буквы алфавита?" При использовании только локальных дисков такая ситуация кажется маловероятной, но при подключении к нескольким файловым серверам количество используемых файловых систем резко возрастает...

В OS/2 и Windows for Workgroups эта проблема решена использованием так называемых UNC-имен (Universal Naming Convention — универсальное соглашение об именах), задающих имя файла в виде \\NODE\SHARE\PATH\FILE.NAM (рис. 11.2), где NODE — имя сетевого узла, SHARE — имя разделяемого ресурса на этом узле (это может быть не только разделяемый каталог, но и принтер, а в OS/2 также и модем), а PATH\FILE.NAM — путь к файлу относительно разделяемого каталога. К сожалению, далеко не все старые (и даже многие не очень уж старые) программы понимают такие имена. Например, даже стандартный командный процессор системы Windows NT не может выполнить команду cd \\NODE\SHARE\DIR (проверялось на NT 4.0 sp4-6, 2000 sp 1 и XP).

Структура пространства имен в Unix

В операционных системах семейства Unix смонтированные ФС выглядят как каталоги единого дерева (строго говоря, структура каталогов в UNIX не обязана являться деревом; но об этом см. разд. 11.3). Данное дерево начинается с корневого каталога, выделенной ФС, называемой *корневой (root)*. Администратор системы может подмонтировать новую ФС к любому каталогу, находящемуся на любом уровне дерева (рис. 11.3). Такой каталог после этого называют *точкой монтирования (mount point)*, но данное выражение отражает только текущее состояние каталога. После того как мы размонтируем ФС, мы сможем использовать этот каталог как обычный, и наоборот, мы можем сделать точкой монтировки любой каталог.

Такой подход имеет неочевидное, на первый взгляд, но серьезное преимущество перед раздельными пространствами имен для разных физических файловых систем. Преимущество состоит в том, что пространство имен оказывается не связанным с физическим размещением файлов. Следовательно, администра-

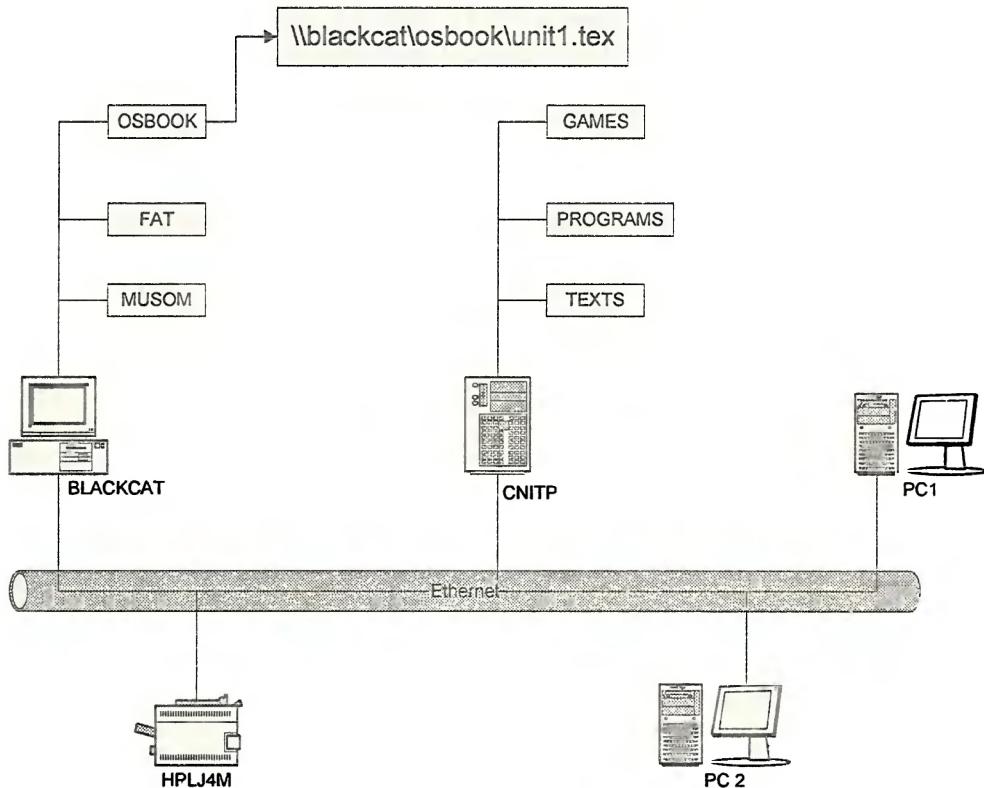


Рис. 11.2. UNC-имена

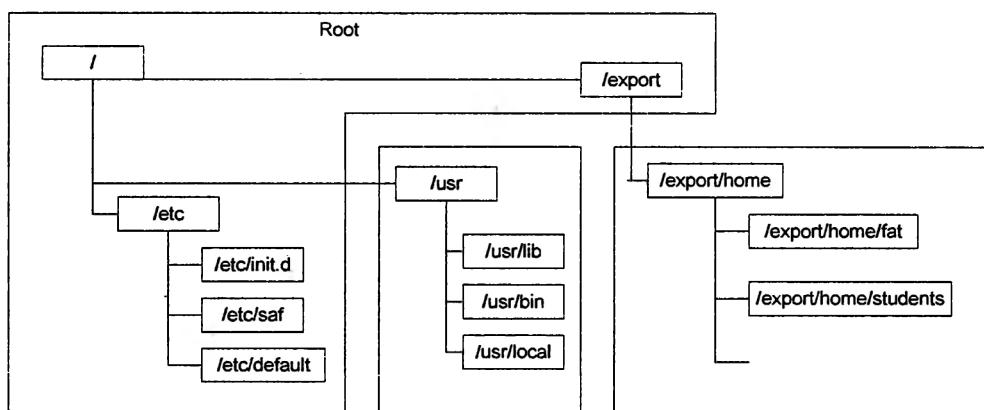


Рис. 11.3. Структура пространства имен в Unix

ратор может поддерживать неизменную структуру дерева каталогов, перемещая при этом отдельные ветви по дискам ради более эффективного использования дискового пространства или даже просто ради удобства администрирования.

По традиции все Unix-системы имеют примерно одинаковую структуру дерева каталогов: системные утилиты находятся в каталоге `/bin`, системные библиотеки — в каталоге `/lib`, конфигурационные файлы — в каталоге `/etc` и т. д. Например, база данных об именах пользователей всегда находится в файле `/etc/passwd`.

Точки монтирования реализованы и в некоторых ОС, не относящихся к семейству Unix. Так, свободно распространяемый продукт TVFS (Toronto Virtual File System) для OS/2 позволяет объединять несколько различных файловых систем (как локальных, так и сетевых) в один логический диск, монтируя реальные ФС в каталоги виртуальной. Аналогичная функциональность была реализована в Windows 2000, но почему-то только в поставке Advanced Server.

11.1.2. Формат имен файлов

В различных ФС допустимое имя файла может иметь различную длину, и в нем могут использоваться различные наборы символов. Так, в RT-11 и RSX-11 имена файлов состоят из символов кодировки RADIX-50 и имеют длину 9 символов: 6 символов — собственно имя, а 3 — расширение. При этом имя имеет вид "XXXXXX.XXX", но символ '.' не является частью имени — это просто знак препинания. Предполагается, что расширение должно соответствовать типу данных, хранящихся в файле: SAV будет именем абсолютного загружаемого модуля, FOR — программы на Фортране, CRH — "файлом информации о системном крахе", как было написано в одном переводе руководства (попросту говоря, это посмертная выдача ОС, по которой можно попытаться понять причину аварии).

В CP/M и ее потомках MS DOS и DR DOS, а также в VMS имена файлов хранятся в 8-битной ASCII-кодировке, но почему-то разрешено использование только букв верхнего регистра, цифр и некоторых печатаемых символов. При этом в системах линии CP/M имя файла имеет 8 символов плюс 3 символа расширения, а в VMS как имя, так и расширение могут содержать более 32 символов. Все перечисленные системы используют нечувствительный к регистру букв поиск в каталогах: имена file.c, File.C и FILE.C считаются одним и тем же именем.

Ограничения на формат имени в MS DOS

Любопытно, что MS/DR DOS при поиске в каталоге переводят в верхний регистр имя, заданное пользователем, но оставляют без изменений имя, считанное из каталога. Строго говоря, это ошибка: если мы создадим имя файла, содержащее буквы нижнего регистра, то ни одна программа не сможет открыть или переименовать такой файл.

Мне довелось столкнуться с такой проблемой при попытке прочитать дискету, записанную ОС ТС (Экспериментальная UNIX-подобная ОС для Паскаль-машины N9000). Проблему удалось решить только с помощью шестнадцатеричного дискового редактора прямым редактированием имен в каталогах. Возможно, существует и более элегантное решение, но мне не удалось его найти.

Использовать конструкцию `*.*` бесполезно, потому что, в действительности, операции над файлами, заданными таким образом, состоят из двух операций: `FindFirst/FindNext`, которая возвращает [следующее] имя файла, соответствующее шаблону, и `Open`. `FindFirst/FindNext` возвращает недопустимое имя файла и `Open` не может использовать его. Программа `CHKDSK` не возражает против имен файлов в нижнем регистре. Строго говоря, это тоже ошибка. Все остальные способы так или иначе сводятся к прямой (в обход ДОС) модификации ФС.

Кроме того, любопытных эффектов можно достичь, попытавшись создать файл с именем, содержащим русские буквы.

Имена файлов в ОС семейства Unix

Наибольшим либерализмом в смысле имен отличаются ОС семейства Unix, в которых имя файла может состоять из любых 8-битных символов, в том числе символов кодировки ASCII, какого-то из национальных алфавитов или даже многобайтовых кодов UTF-8, кроме символов ASCII (NULL) и '/'. Последнее требование обусловлено тем, что символ ASCII (NULL) является признаком окончания имени (как в интерфейсе системных вызовов, так и при хранении данных в каталоге), а '/' — разделителем между именем каталога и именем файла. Никакого разделения на имя и расширение нет, и хотя имена файлов с программой на языке С заканчиваются ".c", а объектных модулей — ".o", точка здесь является частью имени. Вы можете создать файл с именем "gcc-2.5.8.tar.gz". В UNIX SVR3 длина имени файла ограничена 14 символами, а в BSD UNIX, Linux и SVR4 — только длиной блока на диске, т. е. 512 байтами или более.

Возможность использовать в именах неалфавитные символы типа перевода каретки или ASCII EOT (End Of Transmission) кажется опасным излишеством. На самом деле:

- это не излишество, а, скорее, упрощение — из процедур, работающих с именами, удалена проверка символа на "допустимость";
- оно не столь уж опасно: такой файл всегда можно переименовать.

В некоторых случаях процесс набора имени файла в командной строке превращается в нетривиальное упражнение, потому что `shell` (командный процессор) рассматривает многие неалфавитные символы как команды. Но надо отметить, что, правильно используя кавычки и символ '\', пользователь может передать команде аргумент, содержащий любые символы ASCII, кроме '\000'.

Длинные имена файлов в ОС семейства CP/M

В последнее время в ОС стало модным поддерживать длинные имена файлов. Отчасти это, возможно, связано с тем, что производители ПО для персональных компьютеров осознали, что системы семейства Unix являются потенциаль-

но опасными конкурентами, а длинные имена файлов традиционно считаются одним из преимуществ этого семейства.

Например, OS/2, использующая файловую систему *HPFS* (*High Performance File System* — высокопроизводительная файловая система), поддерживает имена файлов длиной до 256 символов, содержащие печатаемые символы и пробелы. Точка считается частью имени, как и в UNIX, и можно создавать имена, содержащие несколько точек. Аналогичную структуру имеют имена в *NTFS*, используемой в Windows NT/2000/XP, *VFAT* (реализация файловой системы *FAT16*, используемая в Windows 95/98/ME) и *FAT32*.

Описанные ОС при поиске файла приводят к одному регистру все алфавитные символы в имени. С одной стороны, это означает дополнительное удобство для пользователя — при наборе имени не нужно заботиться о регистре букв, с другой — пользователь не может создать в одном каталоге файлы "text.txt" и "Text.txt". Из-за этого, например, нельзя использовать принятые в UNIX соглашения о том, что файл на языке С имеет расширение "c", а на языке C++ — "C".

Главная же проблема, возникающая при работе с нечувствительными к регистру именами, — это преобразование регистра в именах, использующих национальные алфавиты: русский, греческий, японскую слоговую азбуку и т. д. Файловая система, поддерживающая такие имена, должна учитывать языковые настройки ОС, что создает много сложностей, в том числе и при считывании удаленных носителей, записанных в одной стране, где-нибудь за границей. В системах семейства Win32 эта проблема решена за счет хранения имен в формате Unicode.

Некоторые ОС, например, RSX-11 и VMS, поддерживают также номер версии файла. В каталоге может существовать несколько версий файла с одним именем; если номер версии при открытии файла не задается, то открывается последняя версия.

Версии файла очень удобны при разработке любых объектов, от программ или печатных плат до книг: если вам не понравились изменения, внесенные вами в последнюю версию, вы всегда можете откатиться назад. Ныне функцию хранения предыдущих версий изменяемых файлов и управляемого отката к ним реализуют специальные приложения, *системы управления версиями* (*version control system*) (RCS, CVS и др.).

11.1.3. Операции над файлами

Большинство современных ОС рассматривают файл как неструктурированную последовательность байтов переменной длины. В стандарте POSIX над файлом определены следующие операции.

`int open(char * fname, int flags, mode_t mode)`

Эта операция "открывает" файл, устанавливая соединение между программой и файлом. При этом программа получает "ручку" или *дескриптор*

файла — целое число, идентифицирующее данное соединение. Фактически это индекс в системной таблице открытых файлов для данной задачи. Все остальные операции используют этот индекс для ссылки на файл. Параметр `char * fname` задает имя файла. `int flags` — это битовая маска, определяющая режим открытия файла. Файл может быть открыт только для чтения, только для записи и для чтения и записи; кроме того, можно открывать существующий файл, а можно пытаться создать новый файл нулевой длины. Необязательный третий параметр `mode` используется только при создании файла и задает атрибуты этого файла.

□ `off_t lseek(int handle, off_t offset, int whence)`

Эта операция перемещает указатель чтения/записи в файле. Параметр `offset` задает количество байтов, на которое нужно сместить указатель, а параметр `whence` — начало отсчета смещения. Предполагается, что смещение можно отсчитывать от начала файла (`SEEK_SET`), от его конца (`SEEK_END`) и от текущего положения указателя (`SEEK_CUR`). Операция возвращает положение указателя, отсчитываемое от начала файла. Таким образом, вызов `lseek(handle, 0, SEEK_CUR)` возвратит текущее положение указателя, не передвигая его.

□ `int read(int handle, char * where, size_t how_much)`

Операция чтения из файла. Указатель `where` задает буфер, куда нужно поместить прочитанные данные; третий параметр указывает, сколько данных надо считать. Система считывает требуемое число байтов из файла, начиная с указателя чтения/записи в этом файле, и перемещает указатель к концу считанной последовательности. Если файл кончился раньше, считывается столько данных, сколько оставалось до его конца. Операция возвращает количество считанных байтов. Если файл открывался только для записи, вызов `read` возвратит ошибку.

□ `int write(int handle, char * what, size_t how_much)`

Операция записи в файл. Указатель `what` задает начало буфера данных; третий параметр указывает, сколько данных надо записать. Система записывает требуемое число байтов в файл, начиная с указателя чтения/записи в этом файле, заменяя хранившиеся в этом месте данные, и перемещает указатель к концу записанного блока. Если файл кончился раньше, его длина увеличивается. Операция возвращает количество записанных байтов. Если файл открывался только для чтения, вызов `write` возвратит ошибку.

□ `int ioctl(int handle, int cmd, ...)`

`int fcntl(int handle, int cmd, ...)`

Дополнительные операции над файлом. Первоначально, по-видимому, предполагалось, что `ioctl` — это операции над самим файлом, а `fcntl` —

это операции над дескриптором открытого файла, но потом историческое развитие несколько перемешало функции этих системных вызовов. Стандарт POSIX определяет некоторые операции как над дескриптором, например дублирование (в результате этой операции мы получаем два дескриптора, связанных с одним и тем же файлом), так и над самим файлом, например, операцию `truncate` — обрезать файл до заданной длины. В большинстве версий Unix операцию `truncate` можно использовать и для вырезания данных из середины файла. При считывании данных из такой вырезанной областичитываются нули, а сама эта область не занимает физического места на диске.

Важной операцией является блокировка участков файла. Стандарт POSIX предлагает для этой цели библиотечную функцию, но в системах семейства Unix эта функция реализована через вызов `fcntl`.

Большинство реализаций стандарта POSIX предлагает и свои дополнительные операции. Так, в Unix SVR4 этими операциями можно устанавливать синхронную или отложенную запись (подробнее понятие отложенной записи обсуждается в разд. 10.7) и т. д.

- `caddr_t mmap(caddr_t addr, size_t len, int prot, int flags, int handle, off_t offset)`

Отображение участка файла в виртуальное адресное пространство процессора. Параметр `prot` задает права доступа к отображеному участку: на чтение, запись и исполнение. Отображение может происходить на заданный виртуальный адрес, или же система может выбирать адрес для отображения сама.

Еще две операции выполняются уже не над файлом, а над его именем: это операции переименования и удаления файла. В некоторых системах, например в системах семейства Unix, файл может иметь несколько имен, и существует только системный вызов для удаления имени. Файл удаляется при удалении последнего имени.

Видно, что набор операций над файлом в этом стандарте очень похож на набор операций над внешним устройством. И то, и другое рассматривается как неструктурированный поток байтов. Для полноты картины следует сказать, что основное средство межпроцессной коммуникации в системах семейства Unix (труба) также представляет собой неструктурированный поток данных. Идея о том, что большинство актов передачи данных может быть сведено к байтовому потоку, довольно стара, но Unix была одной из первых систем, где эта идея была приближена к логическому завершению.

Примерно та же модель работы с файлами принята в CP/M, а набор файловых системных вызовов MS DOS фактически скопирован с вызовов Unix v7. В свою очередь, OS/2 и Windows NT/2000/XP унаследовали принципы работы с файлами непосредственно от MS DOS.

В системах, не имеющих Unix в родословной, может использоваться несколько иная трактовка понятия файла. Чаще всего файл трактуется как набор записей (рис. 11.4). Обычно система поддерживает записи как постоянной длины, так и переменной. Например, текстовый файл интерпретируется как файл с записями переменной длины, а каждой строке текста соответствует одна запись. Такова модель работы с файлами в RSX-11/VMS и в ОС линии OS/360—MVS—z/OS фирмы IBM.

Практика систем с неструктурированными файлами показала, что хотя структурированные файлы часто бывают удобны для программиста, необязательно встраивать поддержку записей в ядро системы. Это вполне можно сделать и на уровне библиотек. К тому же структурированные файлы сами по себе не решают серьезной проблемы, полностью осознанной лишь в 80-е годы при разработке новых моделей взаимодействия человека с компьютером.

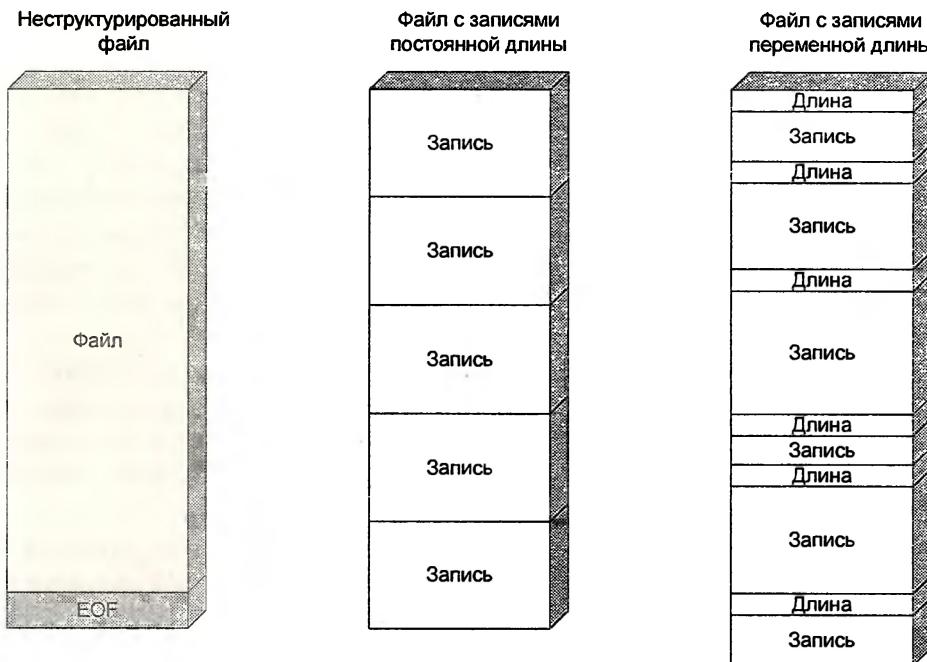


Рис. 11.4. Неструктурированный файл и файлы как наборы записей

11.1.4. Тип файла

Легко понять, что структурированные файлы предоставляют системе и программисту информацию о структуре хранящихся данных, но не дают никаких сведений о форме представления и смысле этих данных.

Например, с точки зрения системы, исходный текст программы на языке С и документ в формате L_AT_EX совершенно идентичны: и то, и другое представляет собой текстовый файл (или, в RSX-11/VMS, файл с записями переменной длины). Однако, если мы попытаемся подать наш документ на вход С-компиллятора, мы получим множество синтаксических ошибок и никакого полезного результата.

Этот пример показывает, что во многих случаях желательно связать с файлом — неважно, структурированный ли это файл или байтовый поток — какую-то метаинформацию: в каком формате хранятся данные, какие операции над ними допустимы, а иногда и сведения о том, кому и зачем эти данные нужны.

По-видимому, наиболее общим решением этой проблемы был бы объектно-ориентированный подход, в котором файл данных рассматривается как объект, а допустимые операции — как методы этого объекта. Ни в одной из известных автору ОС эта идея в полной мере не реализована, но пользовательские интерфейсы многих современных ОС предоставляют возможность ассоциировать определенные действия с файлами различных типов.

Так, например, Explorer — пользовательская оболочка Windows 95, Windows NT 4.0 и последующих версий Windows — позволяет связать ту или иную программу с файлами, имеющими определенное расширение, например, программу MS Word с файлами, имеющими расширение DOC. Когда пользователь нажимает левую кнопку мыши на значке, представляющем такой файл, то автоматически запускается MS Word. Эти же ассоциации доступны и из командной строки — можно напечатать start Доклад.DOC, и опять-таки запустится MS Word.

Такое связывание очень просто в реализации и сделано не только в Explorer, но и в простых текстовых оболочках вроде Norton Commander или Far Manager. От ОС при этом требуется только дать возможность каким-то образом различать типы файлов.

Первые попытки ассоциировать с файлом признак типа были сделаны еще в 60-е годы XX века. При этом идентификатор типа добавлялся к имени файла в виде короткой, но мнемонической последовательности символов — *расширения* (*extension*). В большинстве современных ОС расширение отделяется от имени символом ".", но проследить истоки этой традиции мне не удалось. При этом, например, файлы на языке С будут иметь расширение "c", на C++ — "C", а документы в формате L_AT_EX — "tex".

В ОС семейства Unix имя файла может содержать несколько символов ".", и, таким образом, файл может иметь несколько каскадированных расширений. Например, файл "main.C" — это программа на языке C++; "main.C.gz" — это

программа на языке C++, упакованная архиватором GNU Zip с целью сэкономить место; "main.C.gz.crypt" — это программа, которую упаковали и потом зашифровали, чтобы никто посторонний не смог ее прочитать; наконец, "main.C.gz.crypt.uue" — это упакованная и зашифрованная программа, преобразованная в последовательность печатаемых символов кода ASCII, например, для пересылки по электронной почте.

В принципе, расширения являются вполне приемлемым и во многих отношениях даже очень удобным способом идентификации типа файла. Одно из удобств состоит в том, что для использования этого метода не нужно никаких или почти никаких усилий со стороны ОС: просто программы договариваются интерпретировать имя файла определенным образом.

Пример 11.1. Командная строка компилятора

```
cc main.C c-code.c asm-code.s obj-code.o\ library1.a library2.so -o  
program
```

Например, стандартный драйвер компилятора в системах семейства Unix — программа cc — определяет тип файла именно по расширению. Командная строка, приведенная в примере 11.1, будет интерпретироваться следующим образом:

- main.C — текст на языке C++. Его нужно обработать препроцессором и откомпилировать компилятором C++, а затем передать то, что получится, редактору связей. Большинство компиляторов в Unix генерируют код на ассемблере, т. е. вывод компилятора еще нужно пропустить через ассемблер;
- c-code.c — текст на языке С. Он обрабатывается так же, как и программа на C++, только вместо компилятора C++ используется компилятор С;
- asm-code.s — программа на языке ассемблера. Ее нужно обработать ассемблером и получить объектный модуль;
- obj-code.o — объектный модуль, который непосредственно можно передавать редактору связей;
- library1.a — объектная библиотека, которую нужно использовать для разрешения внешних ссылок наравне со стандартными библиотеками;
- library2.so — разделяемая библиотека, которую надо связать с создаваемым динамическим модулем.

Многие ОС, разработанные в 70-е годы, такие как RT-11, RSX-11, VAX/VMS, CP/M, навязывают программисту разделение имени на собственно имя и расширение, интерпретируя точку в имени файла как знак препинания. В таких системах имя может содержать только одну точку и соответственно

иметь только одно расширение. Напротив, в ОС нового поколения — OS/2, Windows NT и даже в Windows 95 — реализована поддержка имен файлов свободного формата, которые могут иметь несколько каскадированных расширений, как и в Unix.

Однако никакие средства операционной системы не могут навязать прикладным программам правила выбора расширения для файлов данных. Это приводит к неприятным коллизиям. Например, почти все текстовые процессоры от Лексикона до Word 2007 включительно используют расширение файла .doc (сокращение от document), хотя форматы файлов у различных процессоров и даже у разных версий одного процессора сильно различаются.

Другая проблема связана с исполняемыми загрузочными модулями. Обычно система использует определенное расширение для исполняемых файлов.

Так, VMS и системы семейства CP/M используют расширение .exe: сокращение от executable (исполняемый). Однако по мере развития системы формат загрузочного модуля может изменяться. Так, например, OS/2 v3.0 поддерживает по крайней мере шесть различных форматов загрузочных модулей:

- 16-разрядные сегментированные загрузочные модули: формат OS/2 1.x;
- 32-разрядные загрузочные модули, использующие "плоскую" (flat) модель памяти: формат OS/2 2.x (LE — Linear Executable);
- 32-разрядные модули нового формата, использующие упаковку кода и данных: формат OS/2 3.x (LX — Linear executable eXtended);
- exe- и com-модули DOS;
- загрузочные модули Win16 (NE);
- загрузочные модули Win32s (PE — Portable Executable).

Для исполнения последних трех типов программ OS/2 создает виртуальную машину, работающую в режиме совместимости с 8086. Эта задача запускает копию ядра DOS и, если это необходимо, копию MS Windows, которые уже выполняют загрузку программы. Загрузочные модули всех трех "родных" форматов загружаются системой непосредственно. Так или иначе, загрузчик должен уметь правильно распознавать все форматы. При этом он не может использовать расширение файла: файлы всех перечисленных форматов имеют одинаковое расширение exe.

Похожая ситуация имеет место в системах семейства Unix, где бинарные загрузочные модули и командные файлы вообще не имеют расширения. При этом большинство современных версий системы также поддерживает несколько различных исторически сложившихся форматов загрузочного модуля.

Разработчики Unix столкнулись с этой проблемой еще в 70-е годы XX века. В качестве решения они предложили использовать *магические числа* (*magic numbers*)

number) или *сигнатуры* (*signature* — подпись) — соглашение о том, что файлы определенного формата содержат в начале определенный байт или последовательность байтов. Первоначально это были численные коды; файл /etc/magic содержал коды, соответствующие известным типам файлов. Позднее в качестве магических чисел стали использоваться длинные текстовые строки. Так, например, изображения в формате CompuServe GIF 87a должны начинаться с символов GIF87a.

Легко понять, что магические числа ничуть не лучше расширений, а во многих отношениях даже хуже. Например, пользователь, просмотрев содержимое каталога, не может сразу узнать типы содержащихся в нем файлов. Еще хуже ситуация, когда расширение файла не соответствует его реальному типу. Это будет вводить в заблуждение не только пользователя, но и некоторые программы, полагающиеся при определении формата на расширение вместо магического числа.

С длинными мнемоническими текстовыми строками связана еще одна забавная проблема, которая может иметь неприятные последствия. Например, текстовый файл следующего содержания:

GIF87a — это очень плохой формат хранения изображений.

Такой файл будет воспринят некоторыми программами как изображение в формате CompuServe GIF 87a, каковым он, безусловно, не является.

Оригинальное развитие идея магических чисел получила в современных системах семейства Unix, где сигнтура вида #!/bin/sh означает, что данный файл представляет собой интерпретируемую программу, интерпретатор которой хранится в файле /bin/sh (в данном случае это стандартный командный процессор).

Пытаясь как-то решить проблему идентификации типа файла, разработчики Macintosh отказались как от расширений, так и от магических чисел. В MacOS каждый файл состоит из двух частей или ветвей (*forks*): *ветви данных* (*data fork*) и *ветви ресурсов* (*resource fork*). Кроме идентификатора типа файла, ветвь ресурсов хранит информацию о:

- значке, связанном с этим файлом;
- расположении значка в открытой папке (folder);
- программе, которую нужно запустить при "открытии" этого файла.

Еще дальше в этом же направлении пошли разработчики системы OS/2. В этой системе с каждым файлом связан набор *расширенных атрибутов* (*extended attributes*). Атрибуты имеют вид "Имя:Значение". При этом значение может быть как текстовой строкой, так и блоком двоичных данных произвольного формата и размера. Некоторые расширенные атрибуты использу-

ются оболочкой Workplace Shell (WPS) как эквивалент ветви ресурсов в Mac OS: для идентификации типа файла, связанного с ним значка и размещения этого значка в открытом окне. Тип файла идентифицируется текстовой строкой. Например, программа на языке С идентифицируется строкой с code. Это резко уменьшает вероятность конфликта имен типов — ситуацию, довольно часто возникающую при использовании расширений или магических чисел.

Другие атрибуты могут применяться для иных целей. Например, LAN Server — файловый сервер фирмы IBM — использует расширенные атрибуты для хранения информации о владельце файла и правах доступа к нему. Некоторые текстовые редакторы применяют расширенные атрибуты для хранения положения курсора при завершении последней сессии редактирования, так что пользователь всегда попадает в то место, где он остановился в прошлый раз.

Кроме того, расширенные атрибуты могут использоваться и для хранения сведений о назначении файла. В OS/2 существуют предопределенные расширенные атрибуты с именами Subject (тема), Comment (комментарии) и Key phrases (ключевые фразы), которые могут, например, использоваться для поиска документов, относящихся к заданной теме. К сожалению, такой поиск возможен, только если создатель документа позаботился о присвоении этим атрибутам правильных значений.

Ряд сетевых протоколов — HTTP, NNTP, SMTP — допускает снабжение передаваемых данных метаинформацией о том, что это за данные, в каком формате они передаются и что с ними можно делать. Метаинформация передается в том же потоке, что и сами данные, но перед данными, в так называемом заголовке (header). В NNTP и SMTP для этого используются необязательные поля MIME (Multipurpose Internet Mail Extension — многоцелевое расширение почты Internet). В заголовках HTTP 1.1 соответствующие поля обязательны.

11.2. Простые файловые системы

Наиболее простой файловой системой можно считать структуру, созданную архиватором системы UNIX — программой tar (Tape ARchive — архив на [магнитной] ленте). Этот архиватор просто пишет файлы один за другим, помещая в начале каждого файла заголовок с его именем и длиной (рис. 11.5). Аналогичную структуру имеют файлы, создаваемые архиваторами типа arj; в отличие от них, tar не упаковывает файлы.

Для поиска какого-то определенного файла вы должны прочитать первый заголовок; если это не тот файл, то отмотать ленту до его конца, прочитать новый заголовок и т. д. Это не очень удобно, если мы часто обращаемся к

отдельным файлам, особенно учитывая то, что цикл "перемотка — считывание — перемотка" у большинства лентопротяжных устройств происходит намного медленнее, чем простая перемотка. Изменение же длины файла в середине архива или его стирание вообще превращается в целую эпопею. Поэтому tar используется для того, чтобы собрать файлы с диска в некую единую сущность, например, для передачи по сети или для резервного копирования, а для работы файлы обычно распаковываются на диск или другое устройство с произвольным доступом.

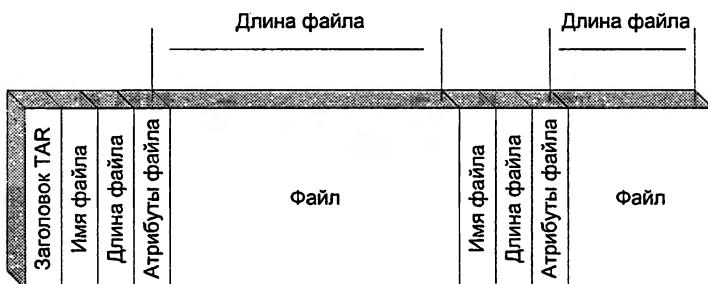


Рис. 11.5. Структура архива tar

Для того чтобы не заниматься при каждом новом поиске просмотром всего устройства, удобнее всего разместить каталог в определенном месте, например в начале ленты. Наиболее простую из знакомых автору файловую систему такого типа имеет ОС RT-11. Это единственная известная автору файловая система, которая с одинаковым успехом применялась как на лентах, так и на устройствах с произвольным доступом. Все более сложные ФС, обсуждаемые далее, используются только на устройствах произвольного доступа. В наше время наиболее распространенный тип запоминающего устройства произвольного доступа — это магнитный или оптический диск, поэтому далее в тексте мы будем называть такие устройства просто дисками — для краткости, хотя такое сокращение и не вполне корректно.

В этой ФС, как и во всех обсуждаемых далее, место на диске или ленте выделяется блоками. Размер блока, как правило, совпадает с аппаратным размером сектора (512 байт у большинства дисковых устройств), однако многие ФС могут использовать *логические блоки*, состоящие из нескольких секторов (так называемые *кластеры*).

Использование блоков и кластеров вместо адресации с точностью до байта обусловлено двумя причинами. Во-первых, у большинства устройств произвольного доступа доступ произволен лишь с точностью до сектора, т. е. нельзя произвольно считывать или записывать любой байт — нужно считывать или записывать весь сектор целиком. Именно поэтому в системах семейства Unix такие устройства называются блочными (block-oriented).

Во-вторых, использование крупных адресуемых единиц позволяет резко увеличить адресуемое пространство. Так, используя 16-битный указатель, с точностью до байта можно адресовать всего 64 Кбайт, но если в качестве единицы адресации взять 512-байтовый блок, то объем адресуемых данных сможет достичь 32 Мбайт; если же использовать кластер размером 32 Кбайт, то можно работать с данными объемом до 2 Гбайт. Аналогично, 32-битовый указатель позволяет адресовать с точностью до байта 4 Гбайт данных, т. е. меньше, чем типичный современный жесткий диск; но если перейти к адресации по блокам, то адресное пространство вырастет до 2 Тбайт, что уже вполне приемлемо для большинства современных запоминающих устройств.

Таким образом, адресация блоками и кластерами позволяет использовать в системных структурах данных короткие указатели, что приводит к уменьшению объема этих структур и к снижению накладных расходов. Под накладными расходами в данном случае подразумевается не только освобождение дискового пространства, но и ускорение доступа: структуры меньшего размера быстреечитываются, в них быстрее производится поиск и т. д. Однако увеличение объема кластера имеет оборотную сторону — оно приводит к внутренней фрагментации (см. разд. 4.2).

Ряд современных файловых систем использует механизм, по-английски называемый *block suballocation*, т. е. размещение частей блоков. В этих ФС кластеры имеют большой размер, но есть возможность разделить кластер на несколько блоков меньшего размера и записать в эти блоки "хвосты" от нескольких разных файлов (рис. 11.6). Это, безусловно, усложняет ФС, но позволяет одновременно использовать преимущества, свойственные и большим, и маленьким блокам. Поэтому ряд распространенных ФС, например файловая система Novell Netware 4.1 и FFS (известная также как UFS и Berkley FS), используемая во многих системах семейства Unix, применяет этот механизм.

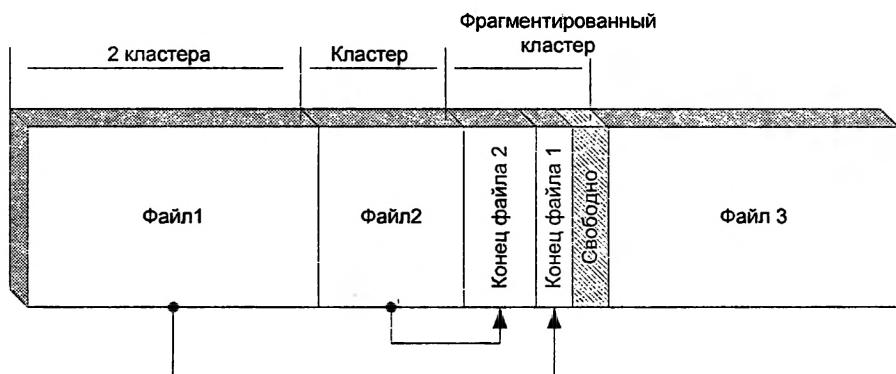


Рис. 11.6. Субаллокация блоков

Субаллокация требует от файловой системы поддержания запаса свободных блоков на случай, если пользователю потребуется увеличить длину одного из файлов, "хвост" которого был упакован во фрагментированный блок. Учебные пособия по Netware рекомендуют поддерживать на томах с субаллокацией не менее тысячи свободных кластеров, но не предоставляют штатных методов обеспечения этого требования. Напротив, UFS показывает свободное место на диске с учетом "неприкосновенного" резерва свободного пространства, так что нулевое показываемое свободное пространство соответствует 5% или 10% физического свободного места. Объем этого резерва настраивается утилитами конфигурации ФС.

Но вернемся к простым файловым системам. В RT-11 каждому файлу выделяется непрерывная область на диске. Благодаря этому в каталоге достаточно хранить адрес первого блока файла и его длину, также измеренную в блоках. В RT-11 поступили еще проще: порядок записей в каталоге совпадает с порядком файлов на диске, и началом файла считается окончание предыдущего файла. Свободным участкам диска тоже соответствует запись в каталоге (рис. 11.7). При создании файла система ищет первый свободный участок подходящего размера.

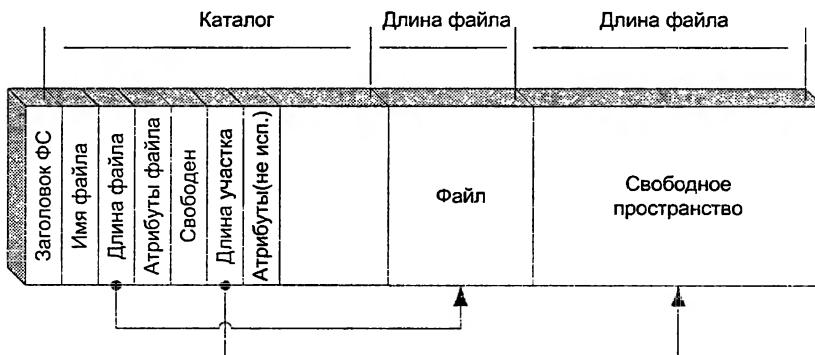


Рис. 11.7. Структура файловой системы RT-11

Фактически эта структура отличается от формата tar только тем, что каталог вынесен в начало диска, и существует понятие свободного участка внутри области данных. Эта простая организация имеет очень серьезные недостатки.

При создании файла программа должна указать его длину. Часто это бывает затруднительно. Особенно неудобно увеличивать размер уже созданного файла. Точнее, это просто невозможно: вместо удлинения старого файла приходится создавать новый файл нужной длины и копировать содержимое старого файла в него.

При хаотическом создании и удалении файлов возникает проблема фрагментации свободного пространства. Для ее решения существует специальная программа SQUEESE (сжать [диск]), которая переписывает файлы так, чтобы объединить все свободные фрагменты (рис. 11.8). Эта программа требует много времени, особенно для больших дисковых томов, и потенциально опасна: если при ее исполнении произойдет сбой системы (а с машинами третьего поколения такое случалось по нескольку раз в день), то значительная часть данных будет необратимо разрушена.

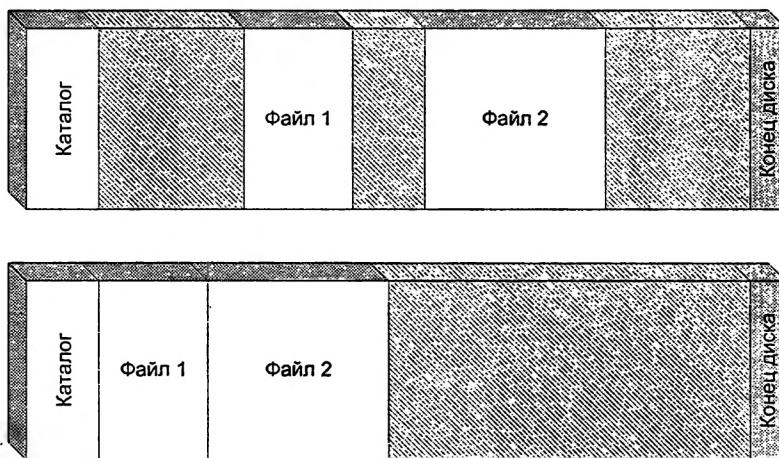


Рис. 11.8. Дефрагментация диска в RT-11

Сочетание этих недостатков привело к тому, что такие ФС имели успех лишь в мини- и микрокомпьютерах с очень малым объемом ОЗУ, в которых было сложно разместить код, работающий с более сложной ФС. Когда память подешевела, а адресное пространство процессоров расширилось, на жестких дисках быстро получили распространение более совершенные файловые системы.

В 90-е годы XX века аналогичная структура ФС неожиданно получила распространение на устройствах однократной записи — оптических компакт-дисках и DVD.

Файловая система CDFS (ISO 9660)

Отличия файловой системы ISO 9660 от приведенной на рис. 11.7 невелики, но заслуживают обсуждения.

Файлы ISO 9660 занимают непрерывное пространство на диске. В каталоге хранятся имя файла, его атрибуты (в первую очередь, дата создания), начало (номер первого сектора) и длина в байтах.

Увеличение длины файла не допускается. Не предусмотрено также механизма описания свободного пространства внутри области данных; однако ФС может

занимать не весь диск, при этом пространство от последнего файла ФС и до конца диска считается свободным.

Важное отличие от рис. 11.7 состоит в том, что каталог вовсе не обязательно размещается в начале диска. Файловая система может содержать несколько каталогов. Каждый каталог является файлом — как в том смысле, что он занимает собственную непрерывную область диска, так и в том смысле, что на него должна существовать ссылка из какого-то другого каталога. Из второго требования существует исключение — корневые каталоги. Ссылки на корневые каталоги размещаются в системной области диска (секторах с 0 по 15).

Корневых каталогов на диске может быть несколько. Это позволяет создать несколько параллельных иерархий каталогов с различными форматами имен и различными атрибутами файлов.

По стандарту, диск обязан иметь стандартную иерархию каталогов с именами в формате ISO 9660. Имена состоят из символов ASCII и разделяются на имя (до 32 символов), расширение (также до 32 символов) и версию. Расширение отделяется от имени символом '.' (ASCII 0x2e), версия — символом ';' (ASCII 0x3b). Алфавитные символы должны храниться в верхнем регистре; при открытии файла, имя также переводится в верхний регистр. Формат имен, таким образом, в основном соответствует формату имен в файловых системах VAX/VMS. Большинство современных программ для записи *CD-ROM* ограничивают длину имени восемью символами, а расширение — тремя, и запрещают использование версий. Соответствующая этим требованиям иерархия каталогов ISO 9660 может быть полностью доступна из-под MS/DR DOS.

Распространены такие альтернативные структуры каталогов, как:

- HFS — формат имен соответствует правилам файловой системы Macintosh. Допускаются блоки расширенных атрибутов (resource forks), однако сами эти атрибуты должны размещаться в том же непрерывном блоке дискового пространства, что и сам файл. При этом каталог ISO 9660 содержит длину файла без учета его расширенных атрибутов;
- Rockridge — Unix-совместимые имена и атрибуты файлов. Имена имеют длину до 512 байт и могут содержать любые восьмибитные символы, кроме ASCII (NULL) и '/' (ASCII 0x2f); имена файлов учитывают регистр алфавитных символов. С файлами связаны три даты: создания, последней модификации и последнего доступа (разумеется, на компакт-диске третья дата не может содержать актуальное значение). Кроме этих дат, каждый файл имеет идентификаторы хозяина и группы и 12-битную маску прав доступа. Поддерживаются жесткие и символические связи;
- Joliet — Windows-совместимая иерархия каталогов. Имена файлов содержат до 256 символов Unicode и сохраняют регистр алфавитных символов, но не учитывают его при поиске файлов.

Существует также несколько менее распространенных форматов каталогов, например HPFS (OS/2-совместимые имена и атрибуты), в котором допускаются расширенные атрибуты и имена файлов могут содержать до 256 символов любой однобайтовой кодировки национального алфавита; при поиске файлов регистр алфавитных символов игнорируется, но что считать алфавитным символом, определяется настройками ОС. Из-за этого диск, записанный в одной кодировке (например, европейской ISO 8859), может не прочитаться в системе, которая настроена на другую кодировку (например, российская CP 866). Справа

ведливости ради надо отметить, что проблемы возникнут только при доступе к файлам, имена которых содержат символы, не входящие в ASCII. Так или иначе, современные версии OS/2 используют Joliet вместо этого формата.

Диск может иметь все эти иерархии каталогов сразу.

Кроме того, несколько иерархий каталогов могут образовываться на диске, если его запись происходила в несколько приемов (так называемые *мультисессионные диски, multisession CD*). При этом неизбежно происходит расширение каталогов. Поскольку не позволяет файлам и каталогам занимать несмежные участки диска, это предполагает отбрасывание старого каталога и создание нового, который содержит ссылки как на старые, так и на новые файлы. Некоторые из старых файлов при этом также могут быть отброшены (CD-ROM представляет собой устройство однократной записи, поэтому переиспользовать пространство, занятное отброшенными файлами и каталогами, невозможно). Поскольку расширяться могут как корневой, так и вложенные каталоги, это приводит к созданию двух независимых, но, как правило, имеющих непустое пересечение иерархий.

Для переноса корневого каталога необходимо перезаписать заголовок ФС; мы уже отмечали, что на CD-ROM это невозможно — поэтому и было введено понятие множественных сессий, каждая из которых имеет собственный заголовок. Каждая сессия занимает непрерывную область на диске, но может ссылаться на файлы и каталоги предыдущих сессий. Современные драйверы ISO 9660 всегда используют заголовок ФС из последней сессии и, соответственно, всегда работают только с последней версией иерархии каталогов.

Необходимость объединять модификации ФС в сессии доставляет много недостатков пользователям — из-за этого CD-ROM невозможно использовать как "нормальную" модифицируемую ФС. Пожалуй, единственный корректный способ создать иллюзию прозрачного доступа — это вносить изменения в кэш на жестком диске, а потом, по специальному запросу, формировать образ ISO и сбрасывать его на CD. Так поступают некоторые программы записи CD для Windows и OS/2, наиболее известная из которых — RSJ CD Writer, а также штатное средство записи CD, поставляемое в составе Windows XP.

Поставщики ПО и операционных систем долго лоббировали развитие стандарта ISO 9660, которое позволило бы сделать из него "нормальную" ФС, допускающую многократное изменение файлов и каталогов без отбрасывания немодифицированных частей этих объектов; особенно усилилось это давление после распространения многократно записываемых дисков CD-RW. Разумеется, это возможно, только если мы позволим файлам и каталогам занимать несмежные области диска. Это предполагалось третьим уровнем стандарта 9660, который так и не был толком специфицирован и никем не поддерживается.

Развитие черновых вариантов спецификаций третьего уровня ISO 9660 привело к стандарту ISO/IEC 13490, который тоже не имел коммерческого успеха и, наконец, к тому, что мы знаем под названиями UDF (Universal Disk Format — универсальный формат диска), ISO 13346 или ECMA 167.

UDF является стандартом для дисков DVD и поддерживается широким спектром ОС на почти столь же широком спектре устройств, в том числе на устройствах однократной записи (CD и CD-ROM), многократной записи с очисткой (CD-RW, DVD-RW, флеш-память) и полноценной многократной записи (DVD-R+, магнитооптические и даже магнитные диски). UDF по структуре близок к тому,

что далее в этой главе описывается под названием "сложные файловые системы" (разд. 11.3); в частности, он допускает размещение файлов в несмежных участках диска и хранит на диске битовую карту свободных блоков.

Интересно отметить, что спецификации UDF и CDFS допускают их параллельное создание на одном и том же диске. В частности, образы таких ФС могут быть созданы программой mkisofs. Довольно часто такая структура используется на фабричных DVD. При этом структуры данных UDF воспринимаются CDFS как еще одна иерархия каталогов. Разумеется, такая гибридная ФС должна соответствовать требованиям ISO 9660, т. е. все файлы должны размещаться в непрерывных областях диска, поэтому и возможности модификации такой ФС оказываются крайне ограничены, в силу чего это имеет смысл лишь на носителях с однократной записью. Зато такой диск может быть прочитан ОС или устройствами, которые поддерживают только одну из этих файловых систем. Итак, ранее мы видели, что для обеспечения удобной работы с устройствами постоянной памяти необходимо позволить файлам занимать несмежные области диска.

В частности, очевидно, что уже упоминавшаяся субаллокация возможна только в условиях, когда файлы не обязаны размещаться в непрерывных областях. Наиболее простым решением было бы хранить в конце каждого блока файла указатель на следующий, т. е. превратить файл в связанный список блоков (рис. 11.9). При этом, естественно, в каждом блоке будет храниться не 512 байт данных, а на 4—8 байт меньше. При строго последовательном доступе к файлу такое решение было бы вполне приемлемым, но при попытках реализовать произвольный доступ возникнут неудобства. Возможно, поэтому, а может, и по каким-то исторически сложившимся причинам такое решение не используется ни в одной из известных мне ФС.

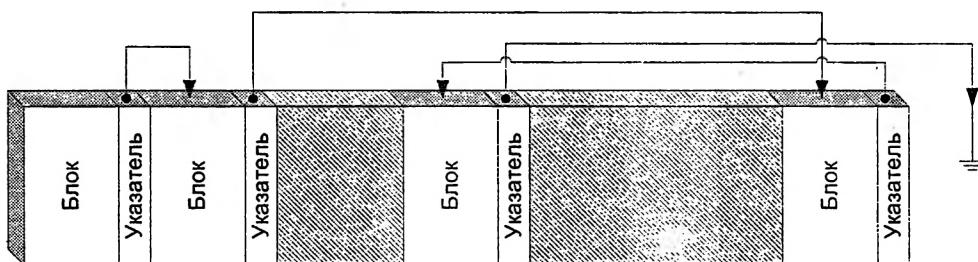


Рис. 11.9. Файл в виде односвязного списка блоков

Отчасти похожее решение было реализовано в MS DOS и DR DOS. Эти системы создают на диске таблицу, называемую *FAT* (*File Allocation Table*, таблица размещения файлов). В этой таблице каждому блоку, предназначенному для хранения данных, соответствует 12-битовое значение. Если блок свобо-

ден, то значение будет нулевым. Если же блок принадлежит файлу, то значение равно адресу следующего блока этого файла. Если это последний блок в файле, то значение — 0xFFFF (рис. 11.10). Существует также специальный код для обозначения плохого (bad) блока, не читаемого из-за дефекта физического носителя. В каталоге хранится номер первого блока и длина файла, измеряемая в байтах.

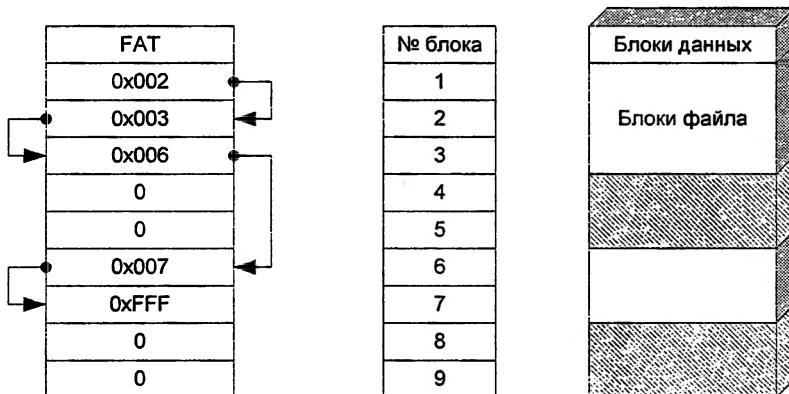


Рис. 11.10. Структура файловой системы FAT

Емкость диска при использовании 12-битовой FAT оказывается ограничена 4096 блоками (2 Мбайт), что приемлемо для дискет, но совершенно не годится для жестких дисков и других устройств большой емкости. На таких устройствах DOS использует FAT с 16-битовыми элементами. На еще больших (более 32 Мбайт) дисках DOS выделяет пространство не блоками, а кластерами из нескольких блоков. Эта файловая система так и называется — FAT.

Она очень проста и имеет одно серьезное достоинство: врожденную устойчивость к сбоям (*fault tolerance*), но об этом далее. В то же время у нее есть и ряд серьезных недостатков.

Первый недостаток состоит в том, что при каждой операции над файлами система должна обращаться к FAT. Это приводит к частым перемещениям головок дисковода, и в результате — к резкому снижению производительности. Действительно, исполнение программы архиватора под MS DOS и под DOS-эмulationом систем Unix или OS/2 различается по скорости почти в 1,5 раза. Особенно это заметно при архивировании больших каталогов.

Использование дисковых кэшей, а особенно помещение FAT в оперативную память, существенно ускоряет работу, хотя обычно FAT кэшируется только для чтения ради устойчивости к сбоям. При этом мы сталкиваемся со специфической проблемой: чем больше диск, тем больше у него FAT, соответст-

венно, тем больше нужно памяти. У тома Novell Netware 3.12 размером 1,115 Гбайт с размером кластера 4 Кбайт размер FAT достигает мегабайта (легко понять, что Netware использует FAT с 32-разрядными элементами. При 16-разрядном элементе FAT дисковый том такого объема с таким размером кластера просто невозможен). При монтировании такого тома Netware занимает под FAT и кэш каталогов около 6 Мбайт памяти.

Для сравнения, Netware 4 использует субаллокацию, поэтому можно относительно безнаказанно увеличивать объем кластера и нет необходимости делать кластер таким маленьким. Для дисков такого объема Netware 4 устанавливает размер кластера 16 Кбайт, что приводит к уменьшению всех структур данных в 4 раза. Понятно, что для MS DOS, которая умеет адресовать всего 1 Мбайт (1088 Кбайт, если разрешен HMA), хранить такой FAT в памяти целиком просто невозможно.

Разработчики фирмы Microsoft пошли другим путем: они ограничили разрядность элемента FAT 16 битами. При этом размер таблицы не может превышать 128 Кбайт, что вполне терпимо. Зато вся файловая система может быть разбита только на 64 Кбайт блоков. В старых версиях MS DOS это приводило к невозможности создавать файловые системы размером более 32 Мбайт.

В версиях старше 3.11 появилась возможность объединять блоки в кластеры. Например, на дисках размером от 32 до 64 Мбайт кластер будет состоять из 2 блоков и иметь размер 1 Кбайт. На дисках размером 128—265 Мбайт кластер будет уже размером 4 Кбайта и т. д.

Windows 95 использует защищенный режим процессора x86, поэтому адресное пространство там гораздо больше одного мегабайта. Разработчики фирмы Microsoft решили воспользоваться этим и реализовали версию FAT с 32-битовым элементом таблицы — так называемый FAT32. Однако дисковый кэш Windows 95 не стремится удержать весь FAT в памяти; вместо этого FAT кэшируется на общих основаниях, наравне с пользовательскими данными. Поскольку работа с файлами большого (больше одного кластера) объема требует прослеживания цепочки элементов FAT, а соответствующие блоки таблицы могут не попадать в кэш, то производительность резко падает. В сопроводительном файле Microsoft признает, что производительность FAT32 на операциях последовательного чтения и записи может быть в полтора раза ниже, чем у кэшированного FAT16.

В заключение можно сказать, что при использовании FAT на больших дисках мы вынуждены делать выбор между низкой производительностью, потребностями в значительном объеме оперативной памяти или большим размером кластера, который приводит к существенным потерям из-за внутренней фрагментации.

Для эффективного управления большими объемами данных необходимо что-то более сложное, чем FAT.

11.3. "Сложные" файловые системы

Структуры "сложных" файловых систем отличаются большим разнообразием, однако можно выделить несколько общих принципов.

Обычно файловая система начинается с заголовка или, как это называется в системах семейства Unix, *суперблока (superblock)*. Суперблок хранит информацию о размерах дискового тома, отведенного под ФС, указатели на начало системных структур данных и другую информацию, зависящую от типа ФС. Например, для статических структур может храниться их размер. Часто суперблок содержит также магическое число — идентификатор типа файловой системы. Аналог суперблока существует даже в FAT — это так называемая *загрузочная запись (boot record)*.

Практически все современные ФС разделяют список свободных блоков и структуры, отслеживающие размещение файлов. Чаще всего вместо списка свободных блоков используется битовая карта, в которой каждому блоку соответствует один бит: занят/свободен. В свою очередь, список блоков для каждого файла обычно связан с описателем файла. На первый взгляд, эти описатели кажутся естественным хранить в каталоге, но их, как правило, выносят в отдельные области, часто собранные в специальные области диска, — таблицу инодов, метафайл и т. д. Такое решение уменьшает объем каталога и, соответственно, ускоряет поиск файла по имени. К тому же многие ФС сортируют записи в каталоге по имени файла, также с целью ускорения поиска. Понятно, что сортировка записей меньшего размера происходит быстрее.

В файловой системе HPFS, используемой в OS/2 и Windows NT 3.51 (драйвер HPFS от NT 3.51 после некоторой доработки напильником также функционирует под NT 4.0 и Windows 2000, но не включен в стандартную поставку этих версий ОС и не поддерживается компанией Microsoft), каждая запись в каталоге содержит имя файла и указатель на *fnode* (файловую запись). Каталоги в этой ФС организованы в виде В-деревьев и отсортированы по именам файлов.

В-деревья и их варианты

В-дерево или блочное дерево — структура данных, оптимизированная для хранения логически отсортированных данных на дисковых носителях. В действительности, это собирательное название для довольно-таки обширного семейства структур. Некоторые типы В-деревьев имеют собственные названия, например В+-деревья, В*-деревья.

Простое В-дерево представляет собой двоичное дерево поиска, в котором каждый узел дерева содержит не один элемент данных (как в простых двоичных деревьях), а целый массив таких элементов. Размер узла при этом соответствует физическому блоку диска или логическому кластеру ФС; узлы В-деревьев также называют блоками.

Как и в простых двоичных деревьях, при размещении элементов в узлах дерева выполняется условие сортировки. Если текущий блок дерева содержит элементы, соответствующие буквам латинского алфавита с M по Q, то элемент, обозначенный буквой K, будет расположен в левом поддереве текущего блока, а обозначенный буквой S — в правом поддереве. Скорость поиска в таком дереве пропорциональна глубине дерева; если дерево сбалансированное (если длина всех его ветвей приблизительно одинакова), то его глубина пропорциональна логарифму количества узлов.

Вставка элемента в В-дерево также выполняется за логарифмическое время. Система должна найти узел, в который новый элемент должен был бы попасть при соблюдении условия сортировки. Если этот узел заполнен, система все равно вставляет элемент в него; при этом выталкивается один из ранее хранившихся в этом узле элементов, первый либо последний. Затем этот элемент вставляется в левое или правое поддерево текущего узла. Легко понять, что первый элемент должен вставляться в левое поддерево, а правый — в правое.

Большинство реализаций В-дерева выбирают первый или последний элемент в зависимости от количества элементов в поддеревьях; благодаря этому, вставка элемента обычно не требует перебалансировки дерева и осуществляется за логарифмическое время.

Однако удаление элемента может превращаться в проблему. Большинство реализаций В-деревьев допускают разреженные, т. е. не до конца заполненные, узлы. Так, если узел допускает хранение пяти элементов с M по Q, а у нас есть только элементы M, P и Q, то узел дерева будет содержать только три элемента. Однако, если мы удалим все элементы из узла, может потребоваться глобальная перебалансировка дерева. Обычно она производится аналогично тому, как перебалансируются двоичные AVL-деревья: если левое поддерево узла содержит меньше элементов, чем правое, то дерево "переподвешивают" за старший из узлов правого поддерева [Кормен/Лейзерсон/Ривест 2000].

В-деревья и их варианты используются во многих современных ФС для хранения каталогов и структур данных, описывающих размещение файлов на диске. Использование деревьев ускоряет поиск в каталоге и произвольный доступ к файлу, однако ряд операций требует многократной перебалансировки дерева. Так, при записи длинного файла происходит многократное добавление блоков или экстентов в конец этого файла; при этом, если не делать перебалансировку, правая ветвь дерева окажется намного больше левой. Такая же проблема возникает при создании или удалении большого количества файлов в одном каталоге, например при распаковке архива или при перемещении иерархии каталогов. Она усугубляется тем, что многие распространенные утилиты для управления файлами — Far, File Commander, Windows Explorer, Midnight Commander — работают с группами файлов в соответствии с лексикографическим порядком их имен.

Кроме простых В-деревьев, в некоторых ФС используются более сложные структуры.

Так, в B+-деревьях узлы делятся на два типа: блоки данных и индексные блоки. Индексные блоки содержат указатели на другие индексные блоки или блоки данных. Блоки данных содержат только данные. Оба типа блоков объединены в деревовидную структуру, в которой блоки данных всегда являются листьями. B+-деревья обычно не двоичные, а сильно ветвящиеся — индексные блоки

имеют не два поддерева, а по поддереву на каждый из хранящихся в блоке элементов (рис. 11.11). Элемент индексного блока, разумеется, соответствует не элементу данных, а диапазону индексов, которые охватываются соответствующей ветвью дерева.

Для упрощения последовательного просмотра все блоки данных B+-дерева связаны в линейный список.

Таким образом, все B+-дерево в целом может рассматриваться как динамический отсортированный массив, который может расширяться с добавлением элементов в произвольные места массива. Сильное ветвление, как правило, снижает количество перебалансировок при вставке в дерево большого количества элементов, даже если эти элементы поступают в отсортированном порядке.

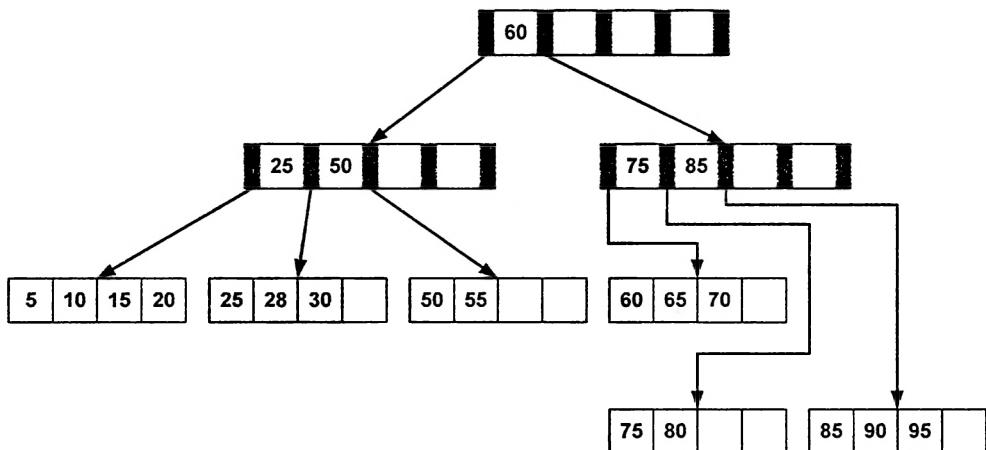


Рис. 11.11. B+-дерево

Файловая запись занимает один блок на диске и содержит список так называемых *extent* ("расширений") — (у этого термина нет приемлемого русского аналога). В переводах документации по OS/360 (ОС ЕС) так и писали: экстент). Каждый экстент описывает непрерывную последовательность дисковых блоков, выделенных файлу. Он задает начальный блок такой последовательности и ее длину. Вместо списка свободных блоков используется битовая карта диска, в которой каждому блоку соответствует один бит: занят/свободен.

Нужно отметить, что идея экстентов далеко не нова: аналогичная структура используется в некоторых версиях Unix с начала 80-х годов, а истоки этой идеи теряются в глубине 60-х годов XX века.

Файловая запись обычно размещается перед началом первого экстента файла, хотя это и не обязательно. Она занимает один блок (512 байт) и может содержать до десяти описателей экстентов (рис. 11.12). Кроме того, она содер-

жит информацию о времени создания файла, его имени и расширенных атрибутах (см. разд. 11.1.4). Если для списка экстентов или расширенных атрибутов места не хватает, то для них также выделяются экстенты. В этом случае экстенты размещаются в виде В-дерева для ускорения поиска. Максимальное количество экстентов в файле не ограничено ничем, кроме размера диска.

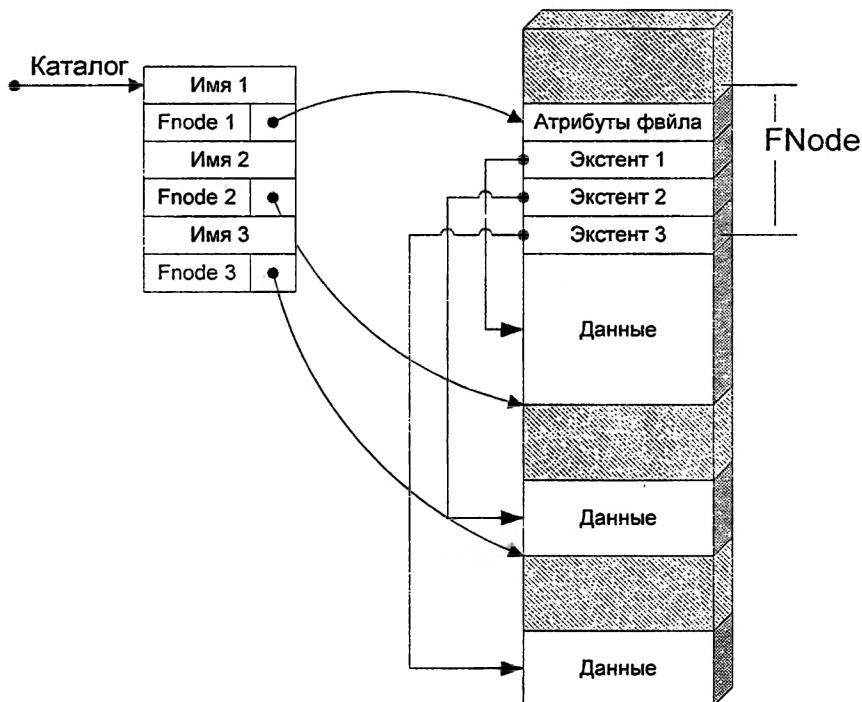


Рис. 11.12. Каталог и файловая запись в HPFS

Пользовательская программа может указать размер файла при его создании. В этом случае система сразу попытается выделить место под файл так, чтобы он занимал как можно меньше экстентов. Если программа не сообщила размера файла, используется значение по умолчанию. Фактически, HPFS размещает место под файл по принципу *worst fit* (наименее подходящего), начиная выделение с наибольшего непрерывного участка свободного пространства. В результате фрагментированными оказываются только файлы, длина которых увеличивалась многократно или же те, которые создавались при почти заполненном диске. При нормальной работе файл редко занимает больше 3—4 экстентов.

Еще одно любопытное следствие применения стратегии *worst fit* заключается в том, что пространство, освобожденное стертым файлом, обычно использу-

ется не сразу. Отмечались случаи, когда файл на активно используемом диске удавалось восстановить через несколько дней после стирания.

Экстенты открытых файлов и карта свободных блоков во время работы размещаются в ОЗУ, поэтому производительность такой ФС в большинстве ситуаций намного (в 1,5—2 раза и более) выше, чем у FAT с тем же объемом кэша, при вполне приемлемых потребностях в памяти и размере кластера 512 байт.

Видно также, что структура HPFS упрощает произвольный доступ к файлу: вместо прослеживания цепочки блоков нам нужно проследить цепочку экстентов, которая гораздо короче.

Более подробное описание структуры HPFS можно найти в статье [PC Magazine 1995]. Пользователи OS/2 считают целесообразным форматировать все разделы ёмкостью более 128 Мбайт под HPFS, поскольку при этом выигрываетя скорость и увеличивается эффективность использования дискового пространства; кроме того, исчезает необходимость в дефрагментации и появляется возможность создавать файлы и каталоги с длинными именами, не укладывающимися в модель 8.3, принятую в MS DOS.

За эти преимущества приходится платить неустойчивостью к сбоям (проблема устойчивости к системным сбоям обсуждается в разд. 11.4). В отличие от DOS, спонтанные разрушения системы с последующим зависанием в OS/2 случаются относительно редко, даже при запуске программ, заведомо содержащих ошибки (как при разработке или тестировании прикладного программного обеспечения). С другой стороны, на практике "неустойчивость" приводит лишь к тому, что после аварийной перезагрузки автоматически запускается программа восстановления ФС, что увеличивает время перезагрузки в несколько раз; реальный же риск потерять данные при сбое не выше, а, как показывает практика, даже существенно ниже, чем при использовании FAT, поэтому игра явно стоит свеч.

Наиболее интересна структура файловых систем в ОС семейства Unix. В этих ФС каталог не содержит почти никакой информации о файле. Там хранится только имя файла и номер его *инода* (i-node — по-видимому, сокращение от index node: индексная запись). Иноды всех файлов в данной ФС собраны в таблицу, которая так и называется: таблица инодов. В ранних версиях Unix таблица инодов занимала фиксированное пространство в начале устройства; в современных файловых системах эта таблица разбита на участки, распределенные по диску.

Например, в файловой системе BSD Unix FFS (Fast File System — быстрая файловая система), которая в Unix SVR4 называется просто UFS (Unix File System), диск разбит на группы цилиндров. Каждая группа цилиндров содержит копию суперблока, битовую карту свободных блоков для данного участ-

ка и таблицу инодов для файлов, расположенных в пределах этого участка (рис. 11.13). Такая распределенная структура имеет два преимущества:

- ускорение доступа к системным структурам данных. Когда системные данные расположены вблизи от блоков пользовательских данных, уменьшается расстояние, на которое перемещаются головки дисковода;
- повышенная устойчивость к сбоям носителя. При повреждении участка поверхности диска теряется только небольшая часть системных данных. Даже потеря суперблока не приводит к потере структуры файловой системы;

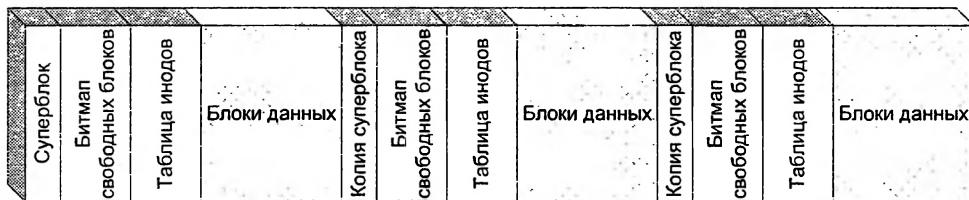


Рис. 11.13. Блоки цилиндров FFS

Инод хранит информацию о самом файле и его размещении на диске (рис. 11.14, пример 11.2). Информационная часть инода может быть получена системным вызовом:

```
int stat(const char * fname, struct stat * buf);
```

Формат структуры `stat` описан во многих руководствах по языку С, ОС UNIX и стандарту POSIX, например, в работе [Керниган-Ритчи 2000]. Эта структура содержит следующую информацию:

- тип файла. Исследуя это поле, можно понять, является данный объект файлом данных или специальным файлом. В этом же поле закодированы права доступа к файлу;
- идентификаторы владельца файла и группы. Вместе с правами доступа эти два идентификатора образуют список контроля доступа файла (*см. разд. 13.4.1*);
- время:
 - создания файла;
 - последней модификации файла;
 - последнего доступа к файлу;
- длина файла. Для специальных файлов это поле часто имеет другой смысл.

- идентификатор файловой системы, в которой расположен файл;
- количество связей файла. Это поле заслуживает отдельного обсуждения.

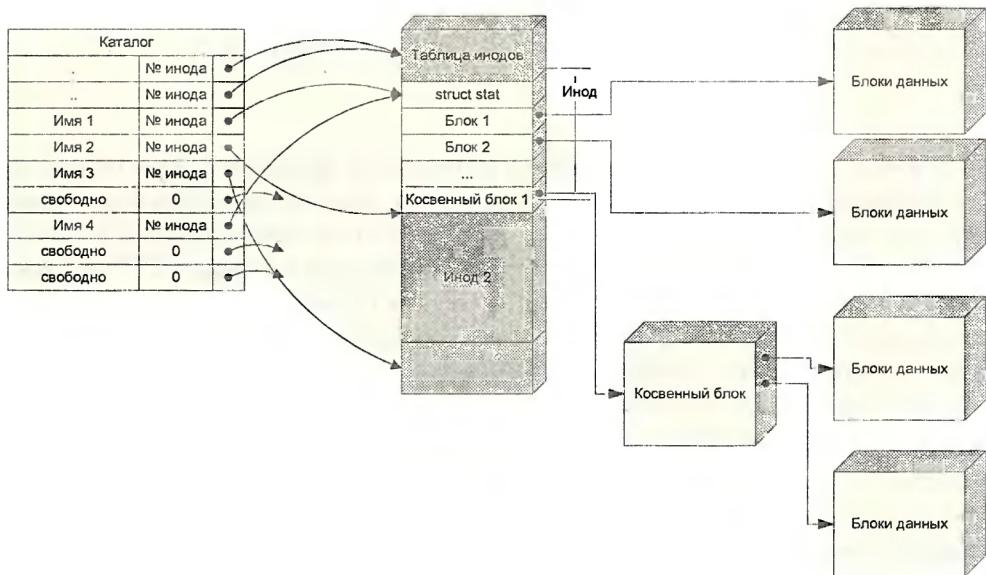


Рис. 11.14. Каталоги и иноды файловых систем семейства Unix

Пример 11.2. Структура инода файловой системы ext2fs

```
/*
 * Структура данных инода second extended file system, считанная в память
 */
struct ext2_inode_info {
    __u32 i_data[15];
    __u32 i_flags;
    __u32 i_faddr;
    __u8 i_frag_no;
    __u8 i_frag_size;
    __u16 i_osync;
    __u32 i_file_acl;
    __u32 i_dir_acl;
    __u32 i_dtime;
    __u32 not_used_1; /* FIX: не используется/зарезервировано для 2.2 */
    __u32 i_block_group;
    __u32 i_next_alloc_block;
```

```
__u32 i_next_alloc_goal;
__u32 i_prealloc_block;
__u32 i_prealloc_count;
__u32 i_high_size;
int i_new_inode:1; /* Этот инод только что выделен */
};
```

Структура, описывающая физическое размещение файла на диске, недоступна пользовательским программам. Собственно, формат этой структуры может быть не очень элегантен. Например, в файловой системе Veritas это список экстентов, похожий на HPFS; в файловых системах s5, Xenix и FFS это массив из 13 чисел, задающих номера физических блоков файла. Если файл в s5 содержит более десяти блоков (т. е. его длина больше 5 Кбайт), то предпоследние три указателя обозначают не блоки данных, а так называемые *косвенные блоки* (*indirection blocks*), в которых хранятся указатели на следующие блоки данных и, возможно, на следующие косвенные блоки.

Наиболее интересная особенность ФС семейства Unix состоит не в этом. Внимательный читатель, возможно, заметил, что инод не содержит имени файла. С другой стороны, он содержит счетчик *связей* (*link*) — ссылок на этот файл из каталогов. Таким образом, на один и тот же инод можно ссылаться из различных каталогов или из одного каталога под разными именами (рис. 11.15). Иными словами, один и тот же файл в этих ФС может иметь несколько различных имен. Именно это и имелось в виду, когда говорилось, что структура каталогов в ОС UNIX не обязана быть деревом.

Это свойство предоставляет неоценимые возможности для организации иерархии каталогов, но имеет и некоторые обратные стороны:

1. Создание нескольких связей для каталога потенциально опасно — оно может привести к возникновению кольца, в котором каталог является своим собственным подкаталогом. Отслеживать такую ситуацию сложно, поэтому разработчики ОС UNIX запретили создавать дополнительные имена для каталогов.
2. Удаление файла превращается в проблему: чтобы удалить файл, нужно проследить все его связи. Поэтому UNIX не имеет средств для удаления файла, а обладает только системным вызовом *unlink* — удалить связь. Когда у файла не остается связей, он действительно удаляется. Этот подход вполне разумен, но также имеет неожиданную обратную сторону: поскольку теперь стирание файла — это операция не над файлом, а над каталогом, то для удаления файла не нужно иметь никаких прав доступа к нему. Достаточно иметь право записи в каталог.

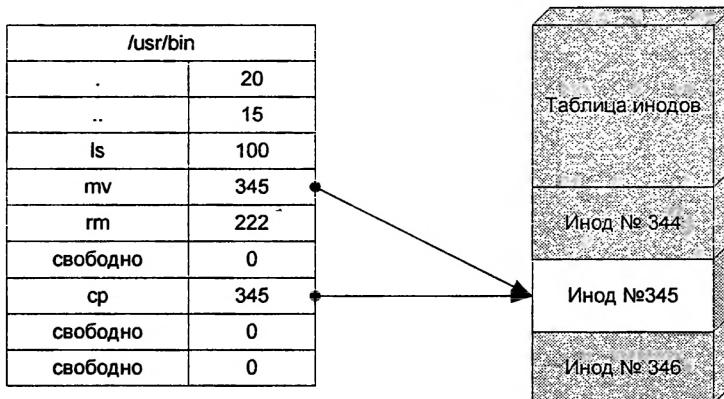


Рис. 11.15. Жесткие связи в Unix

На практике механизм защиты от стирания отдельных файлов предусмотрен — при установке в атрибутах каталога так называемого *sticky bit* ("липкий бит"), система запрещает стирать файлы, для которых вы не имеете права записи, но и этот механизм не лишен недостатков.

3. Такие связи (называемые еще *жесткими (hard)* связями), как легко понять, могут создаваться только в пределах одной файловой системы, поскольку каждая ФС имеет собственную таблицу инодов, и, соответственно, собственную их нумерацию.

Последнее обстоятельство резко уменьшает полезность жестких связей для организации иерархии каталогов. Эта проблема была осознана еще в 70-е годы, и программисты из группы BSD придумали интересное новое понятие — *символическую связь (symbolic link)* или *symlink*.

Символическая связь представляет собой специальный файл. Вместо блоков данных инод такого файла содержит текстовую строку — имя того файла, с которым создана связь (рис. 11.16). Это может быть файл из другой файловой системы, в том числе и из такой, которая сама по себе не поддерживает ни жестких, ни символьских связей, например, FAT, HPFS или файловый сервер Novell Netware. Такого файла может и вообще не существовать, например, потому, что его уже удалили, или потому, что файловая система, в которой он находится, не смонтирована, или просто потому, что имя было задано неправильно. Тогда попытки открыть символическую связь будут завершаться неудачей с кодом ошибки "файла не существует".

Единственным недостатком символьских связей является их относительно низкая "дурокоустойчивость" (fool-tolerance): глупый пользователь может не распознать ситуации, когда символическая связь указывает в никуда. Зато они обеспечивают полную свободу в размещении и именовании файлов.

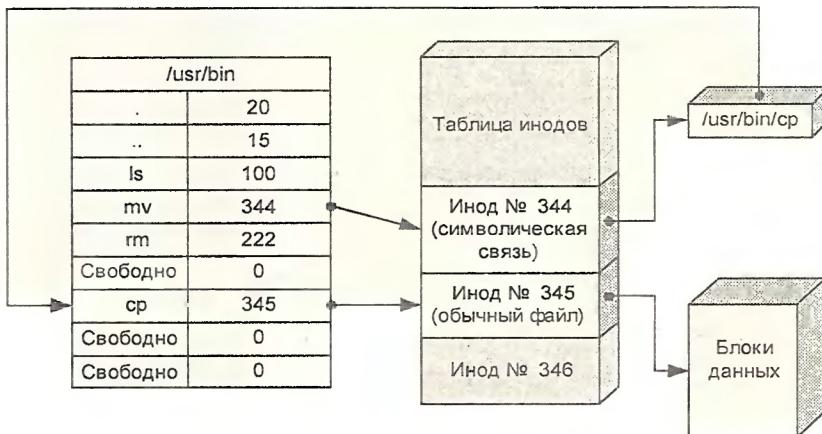


Рис. 11.16. Символическая связь

Пример из жизни

На двух Unix-системах с именами `orasrv` и `Indy` установлен один и тот же программный продукт: интегрированная среда разработки GNU Emacs. Бинарные загрузочные модули для этих систем различаются, но большая часть Emacs — это файлы на языке Emacs Lisp, которые с одинаковым успехом могут использоваться обеими системами. Поэтому у администратора системы возникает желание использовать одну копию lisp-файлов. При этом Emacs установлена таким образом, что она ищет все свои lisp-файлы в каталоге `/usr/local/lib/emacs/19.27/lisp`. Изменение этого каталога потребовало бы частичной переустановки продукта. Но его не надо менять! Мы просто подмонтируем на машине `orasrv` каталог `/fs/Indy` как `/fs/Indy` и исполняем следующие команды примера 11.3.

Пример 11.3. Перенос каталога без изменения его путевого имени

```
cd /usr/local/lib/emacs/19.27 # перейти в заданный каталог
rm -Rf lisp # стереть каталог lisp со всем содержимым
ln -s /fs/Indy/usr/local/lib/emacs/19.27/lisp lisp
# создать символьскую связь с именем lisp с соответствующим
# каталогом на машине Indy
```

Вся операция вместе с ее планированием занимает около двух минут. Освобождается около 9 Мбайт дискового пространства. Emacs ничего не замечает и работает без всяких проблем.

Через неделю у администратора возникает идея, что для удобства администрирования неплохо было бы монтировать с `orasrv` не всю файловую систему `Indy`, а только ее каталог `/home`. За чем дело стало: переносим на `Indy` каталог `lisp` в каталог `/home/emacs/19.27/lisp`; создаем в старом каталоге символьскую связь с новой; редактируем файл `/etc/mnttab` на машине `orasrv` так, чтобы с `Indy`

монтировался только /home; меняем символическую связь каталога lisp на машине orasrv. Заставляем orasrv подмонтировать диски в соответствии с новым /etc/mnttab. Готово.

Операция занимает чуть больше двух минут, потому что нужно переносить 9 Мбайт из одной файловой системы в другую. Emacs опять ничего не замечает и снова работает без всяких проблем. Администратор счастлив. Все довольны. Сравните это описание с впечатлениями пользователя, который пытается переместить с одного ДОС-драйва на другой типичную программу для Win32, которая при установке записывает во многие места своей конфигурации и системного реестра полное имя каталога, в который ее ставили...

Современные стандарты структур файловых систем для Unix специально оптимизированы для того, чтобы облегчить разделение кода и данных системы по сети, как в условиях загрузки по сети, так и просто если объем локального жесткого диска недостаточен. Так, в современных версиях Unix системные исполняемые файлы размещаются в каталогах /bin, /sbin и /usr/bin; файлы разделяемых библиотек — в каталогах /lib и /usr/lib, а файлы данных (например, документация, файлы описания терминалов и т. д.) — в каталоге /usr/share. Благодаря этому, версии ОС для разных аппаратных платформ (например, Solaris для SPARC и x86), вынужденные использовать разные каталоги программ и разделяемых библиотек, могут использовать общий каталог системных данных. Это может достигаться как с помощью соответствующей структуры точек монтирования сетевых ФС, так и с помощью символьических связей.

Символьические связи поддерживаются только системами семейства Unix. Напротив, эквиваленты жестких связей есть в других ОС. Фактически, жесткие связи можно создавать в любой ФС, где каталоги содержат ссылки на централизованную базу данных вместо самого дескриптора файла.

Жесткие связи в VMS и Windows NT/2000/XP

Например, в файловой системе VAX/VMS данные о размещении файлов на диске хранятся в специальном индексном (index) файле; каталоги же хранят только индексы записей в этом файле. Основное отличие этой структуры от принятой в Unix состоит в том, что вместо статической таблицы или набора таблиц используется динамическая таблица, пространство для которой выделяется тем же методом, что и для пользовательских файлов. Этот же подход реализован в файловой системе NTFS, используемой в Windows NT/2000/XP, но в ней индексный файл называется MFT (*Main File Table* — главная таблица файлов). Более подробное описание структуры NTFS приводится в статье [www.digit-life.com NTFS].

VAX/VMS и Windows NT позволяют создавать дополнительные имена для файлов, хотя в VMS утилиты восстановления ФС выдают предупреждение, обнаружив такое дополнительное имя. Все имена файла в этих ФС обязаны находиться в одной файловой системе. Кроме того, операция удаления файла в VMS ведет себя не так, как в Unix: применение операции удаления к любому из имен приводит к удалению самого файла, даже если существовали и другие имена.

11.4. Устойчивость ФС к сбоям

Свойство *устойчивости к сбоям питания* (*power-fault tolerance*) является одной из важных характеристик файловой системы. Строго говоря, имеется в виду устойчивость не только к сбоям питания, но и к любой ситуации, при которой работа с ФС прекращается без выполнения операции размонтирования. Поэтому правильнее было бы говорить об устойчивости к любым сбоям системы, а не только питания.

С другой стороны, если говорить просто об "устойчивости к сбоям", возникает неприятная двусмысленность. Под устойчивостью к сбоям можно понимать устойчивость к сбоям всей системы, например, к тем же сбоям питания, а можно понимать устойчивость к дефектам физического носителя, которые могут привести к потере части данных на диске. Вторая проблема обычно решается с помощью механизма горячей замены (логической переадресации) сбойных блоков, который обсуждается в разд. 11.4.4, а сейчас мы будем говорить только о первой проблеме, называя ее просто "устойчивостью" для краткости.

11.4.1. Устойчивость к сбоям питания

На самом деле, неожиданное прекращение работы с ФС может произойти не только при сбое питания, но и в следующих ситуациях:

- при извлечении носителя из дисковода (подробнее об этом см. далее);
- при нажатии кнопки RESET нетерпеливым пользователем;
- при фатальном аппаратном сбое;
- при аппаратном сбое, который сам по себе не был фатальным, но система не смогла правильно восстановиться, что привело к ее разрушению;
- при разрушении системы из-за чисто программных проблем.

На практике часто сложно провести границы между тремя последними типами сбоев. В качестве примера можно привести проблему, описанную в разд. 10.6.4. В системах класса ДОС последняя причина возникает очень часто, поэтому в таких системах можно использовать только устойчивые к сбоям ФС.

В системах класса ОС "спонтанные" зависания происходят намного реже. Система, зависающая по непонятным или неустранимым причинам раз в несколько дней, считается малопригодной для серьезного использования, а делающая это раз в месяц — подозрительно ненадежной. Поэтому в таких системах можно позволить себе роскошь использовать неустойчивые к сбоям ФС. Тем более что такие системы, как правило, обладают более высокой производительностью, чем устойчивые ФС.

К тому же перед выключением системы, интегрированной в сеть, необходимо уведомить всех клиентов и все серверы о разъединении. Только чисто клиентская машина может быть выключена из сети без проблем. Поэтому на сетевых серверах в любом случае необходима процедура *останова системы* (*shutdown*). Так почему бы не возложить на эту процедуру еще и функцию размонтирования ФС?

В системах же реального времени, которые по роду работы могут часто и неожиданно перезапускаться, например, при отладке драйверов или программ, осуществляющих доступ к аппаратуре, стараются использовать устойчивые к сбоям ФС.

Хотя первая из перечисленных ранее причин — извлечение носителя из дисковода — не является "сбоем" даже в самом широком смысле этого слова, с точки зрения ФС она мало чем отличается от сбоя. Поэтому на удаляемых носителях, таких как дискеты, можно использовать только устойчивые ФС.

Интересный альтернативный подход используется в компьютерах Macintosh и некоторых рабочих станциях. У этих машин дисковод не имеет кнопки для извлечения дискеты. Выталкивание дискеты осуществляется программно, подачей соответствующей команды дисководу. Перед подачей такой команды ОС может выполнить нормальное размонтирование ФС на удаляемом диске.

В узком смысле слова "устойчивость" означает лишь то, что ФС после аварийной перезагрузки не обязательно нуждается в восстановлении. Такие ФС обеспечивают целостность собственных структур данных в случае сбоя, но, вообще говоря, не гарантируют целостности пользовательских данных в файлах.

Нужно отметить, что даже если ФС считается в этом смысле устойчивой, некоторые сбои для нее могут быть опасны. Например, если запустить команду **DISKOPT** на "устойчивой" файловой системе FAT и в подходящий момент нажать кнопку **RESET**, то значительная часть данных на диске может быть навсегда потеряна.

С другой стороны, можно говорить об устойчивости в том смысле, что в ФС после сбоя гарантирована целостность пользовательских данных. Достаточно простого анализа для того, чтобы убедиться, что такую гарантию нельзя обеспечить на уровне ФС; обеспечение подобной целостности накладывает серьезные ограничения и на программы, работающие с данными, а иногда оказывается просто невозможным.

Характерный и очень простой пример: архиватор InfoZip работает над созданием архива. Программа сформировала заголовок файла, упаковала и записала на диск около 50% данных, и в этот момент произошел сбой. В zip-архивах каталог находится в конце архивного файла и записывается туда после за-

вершения упаковки всех данных. Обрезанный в случайном месте zip-файл не содержит каталога и поэтому, безусловно, является испорченным (хотя и не безнадежно: кроме каталога, описатель файла с указанием его атрибутов и имени размещается перед телом самого файла, как в архивах tar, поэтому оборванный архив в определенных пределах можно восстановить; как pkzip, так и InfoZip при использовании специальных ключей могут это сделать [[support.pkware.com appnote](http://support.pkware.com/appnote)]). Поэтому при серьезном обсуждении проблемы устойчивости к сбоям говорят не о гарантии целостности пользовательских данных, а об уменьшении вероятности их порчи. Следует учитывать также возможность полного и частичного восстановления данных при их порче — впрочем, такое восстановление является задачей прикладной программы и не может обеспечиваться системными средствами.

Поддержание целостности структур ФС обычно гораздо важнее, чем целостность недописанных в момент сбоя пользовательских данных. Дело в том, что если при сбое оказывается испорчен создававшийся файл, это достаточно неприятно; если же окажется испорчена файловая система, в худшем случае это может привести к потере всех данных на диске, т. е. к катастрофе. Поэтому обычно обеспечению целостности ФС при сбоях уделяется гораздо больше внимания.

Упоминавшаяся ранее "врожденная" устойчивость к сбоям файловой системы FAT объясняется тем, что в этой ФС удаление блока из списка свободных и выделение его файлу производится одним действием — модификацией элемента FAT (рис. 11.17). Поэтому если во время этой процедуры произойдет сбой или дискета будет вынута из дисковода, то ничего страшного не случится: просто получится файл, которому выделено на один блок больше, чем его длина, записанная в каталоге. При стирании этого файла все его блоки будут помечены как свободные, поэтому вреда практически нет.

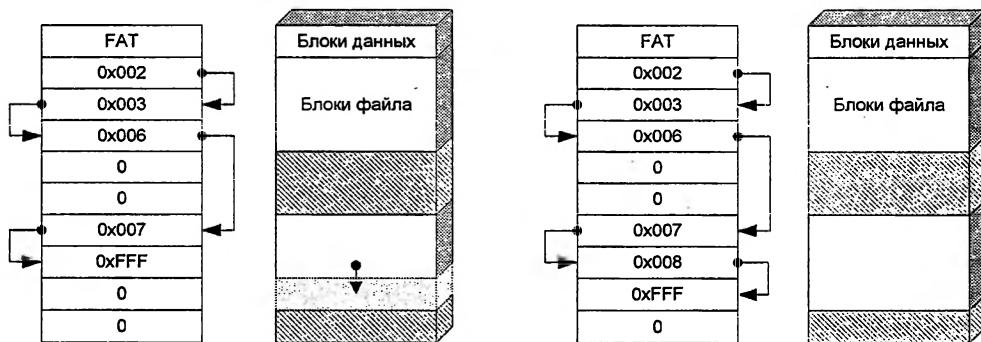


Рис. 11.17. Модификация FAT

Нужно отметить, что при активном использовании отложенной записи FAT и родственные ФС теряют это преимущество. Отложенная запись FAT является

единственным способом добиться хоть сколько-нибудь приемлемой производительности от ФС с 32-битной FAT. Поэтому, хотя Novell NetWare и использует ФС, основанную на 32-битной FAT, после аварийной перезагрузки эта система вынуждена запускать программу аварийного восстановления дисковых томов. Аналогичным образом ведет себя и FAT32.

Если же система хранит в одном месте список или карту свободных блоков, а в другом месте — списки блоков, выделенных каждому файлу (рис. 11.18), как это делают HPFS или ФС систем семейства Unix, то при прерывании операции выделения места в неподходящий момент могут либо теряться блоки (если мы сначала удаляем блок из списка свободных — рис. 11.19), либо получаться блоки, которые одновременно считаются и свободными, и занятыми (если мы сначала выделяем блок файлу).

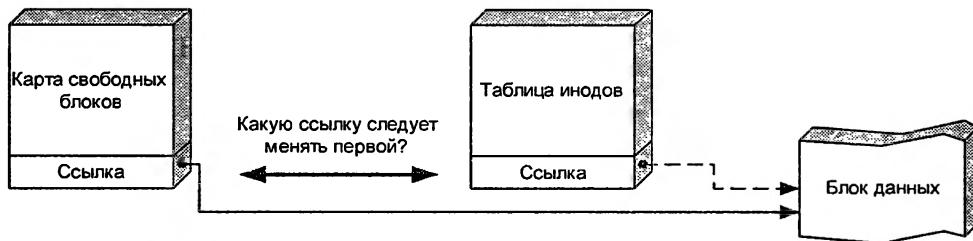


Рис. 11.18. Модификация структур данных сложной ФС

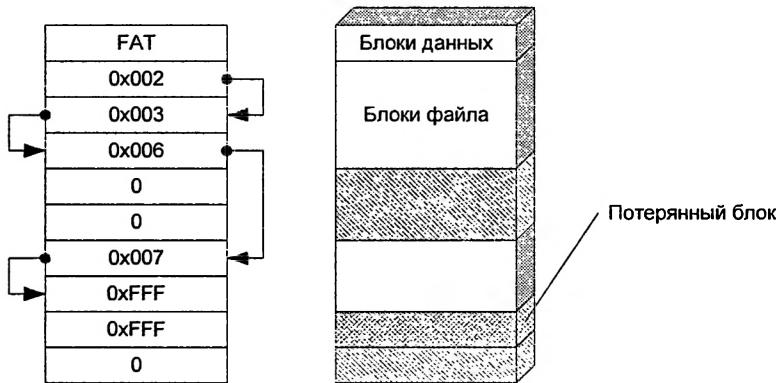


Рис. 11.19. Потерянный блок

Первая ситуация достаточно неприятна, вторая же просто недопустима: первый же файл, созданный после перевызова системы, будет "перекрещиваться" с испорченным (рис. 11.20). Поэтому все ОС, использующие файловые системы такого типа (системы семейства Unix, OS/2, Windows NT и т. д.), после

аварийной перезагрузки первым делом проверяют свои ФС соответствующей программой восстановления.

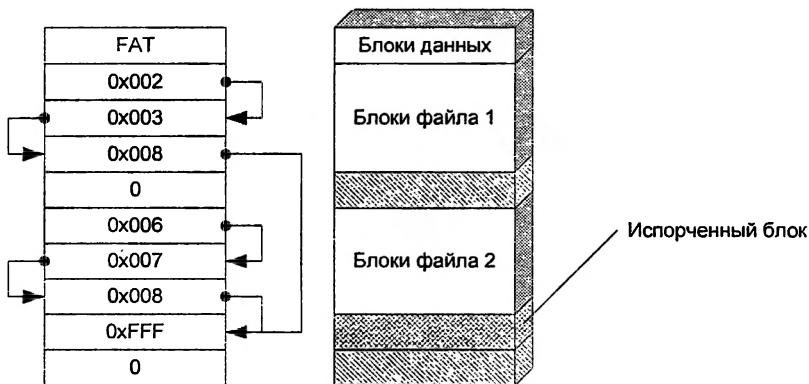


Рис. 11.20. Пересекающиеся файлы

Задача обеспечения целостности файловых систем при сбоях усложняется тем, что дисковые подсистемы практически всех современных ОС активно используют отложенную запись, в том числе и при работе с системными структурами данных. Отложенная запись, особенно в сочетании с сортировкой запросов по номеру блока на диске, может приводить к тому, что изменения инода или файловой записи все-таки пишутся на диск раньше, чем изменения списка свободных блоков, что может приводить к возникновению "скрещенных" файлов. Для того чтобы этого не происходило, сортировке, как правило, подвергают только запросы чтения, но не записи.

11.4.2. Восстановление ФС после сбоя

Но я, рукою твердою, беру иголку дюжину, и пришиваю морду и хвост туда, где нужно. Собака, оживая, встает из-под трамвая. Привет, говорит, живя я.

A. Барто

Чаще всего суперблок неустойчивых ФС содержит *флаг загрязнения* (dirty flag), сигнализирующий о том, что ФС, возможно, нуждается в восстановлении. Этот флаг сбрасывается при нормальном размонтировании ФС и устанавливается при ее монтировании или при первой модификации после монтирования. Таким образом, если ОС погибла, не успев размонтировать свои дисковые тома, после перезагрузки на этих томах dirty-флаг будет установлен, что станет сигналом необходимости починки.

Восстановление состоит в том, что система проверяет пространство, выделенное всем файлам. При этом должны выполняться определенные требования.

- Каждая запись в каталоге должна иметь правильный формат и содержать осмысленные данные. Например, если запись помечена как свободная, она не должна ссылаться на данные, помеченные как принадлежащие файлу, или на инод. Не во всех ФС можно обнаружить ошибки такого типа.
- Каждый блок или кластер диска должен принадлежать не более, чем одному файлу. Блоки, принадлежащие одновременно двум или более файлам, являются очень серьезной ошибкой. На практике это означает, что данные во всех этих файлах (в лучшем случае — во всех, кроме того, запись в который была последней) безнадежно испорчены. Чаще всего программа восстановления в этой ситуации требует вмешательства пользователя, с тем чтобы решить, какие из файлов следует удалить или обрезать по месту пересечения. Иногда для каждого из файлов создается копия "общего" блока, но и в этом случае пользователю все равно нужно определить, какие из файлов испорчены.

Практически все ФС при выделении блока сначала удаляют его из списка свободных и лишь потом отдают файлу, поэтому при "обычных" сбоях прекращение файлов возникнуть может только как следствие отложенной записи в сочетании с сортировкой запросов. Возникновение таких ошибок обычно сигнализирует либо об ошибке в самом файловом менеджере, либо об аппаратных сбоях на диске, либо о том, что структуры ФС были модифицированы в обход файлового менеджера. Например, в DOS это может быть признаком вирусной активности.

- Каждому файлу должно быть выделено пространство, соответствующее его длине. Если файлу выделено больше блоков, чем требуется, лишние блоки помечаются как свободные. Если меньше, файл укорачивается. Возможно, в укороченных файлах часть данных оказывается потеряна.
- Все блоки, не принадлежащие файлам, должны быть помечены как свободные. Соответствующий тип ошибок — *потерянные блоки (lost clusters)* — наиболее частый результат системных сбоев как в файловой системе FAT, так и в более сложных файловых системах. Сами по себе ошибки этого типа относительно безобидны.

Обычно программа восстановления не помечает потерянные блоки как свободные, а выделяет из них непрерывные цепочки и создает из этих цепочек файлы. Например, в OS/2 программа восстановления пытается найти в потерянных блоках файловые записи, а потом создает ссылки на найденные таким образом файлы в каталоге \FOUND.XXX. В DOS эти файлы помещаются в корневой каталог ФС под именами FILEXXXX.CHK (вместо XXXX подставляется номер). Предполагается, что пользователь просматривает все такие фай-

лы и определяет, не содержит ли какой-то из них ценной информации, например, конца насильственно укороченного файла.

В системах семейства Unix существует несколько специфических ошибок, связанных с инодами.

- Инод, внутренний счетчик ссылок которого не соответствует реальному количеству ссылок из каталогов. Эта проблема может возникать при системном сбое в момент удаления существовавшей связи или создания новой. Она решается коррекцией внутреннего счетчика инода. После этого можно обнаружить следующие две ошибки.
- Инод, не имеющий ни одной ссылки, но и не помеченный как свободный — *сирота (orphan)* (рис. 11.21). Ссылка на такой инод создается в каталоге *lost+found*.
- Инод с ненулевым количеством ссылок из каталогов, но помеченный как свободный. Чаще всего это свидетельствует о порче самого каталога. Обычно ссылки на такой инод удаляются.

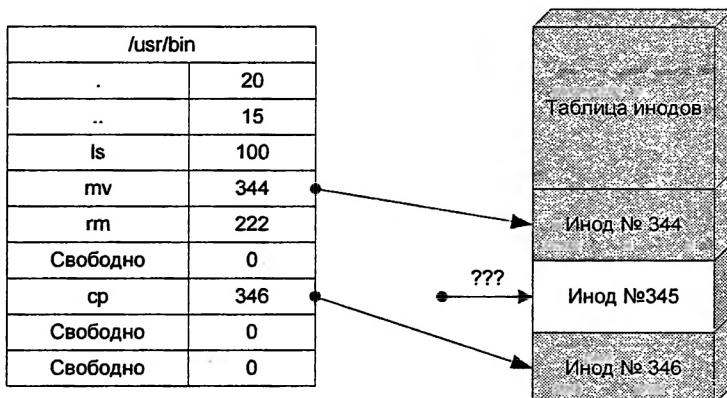


Рис. 11.21. Инод-сирота

Ручное восстановление файловой системы

В некоторых особенно тяжелых случаях программа восстановления оказывается не в состоянии справиться с произошедшей аварией и администратору системы приходится браться за дисковый редактор.

В процессе эксплуатации системы SCO Open Desktop 4.0 у меня неоднократно возникала необходимость выполнять холодную перезагрузку без нормального размонтирования файловых систем. Одна из таких перезагрузок привела к катастрофе.

Дисковая подсистема машины состояла из кэширующего контроллера дискового массива. Контроллер активно использовал отложенную запись, что, по-видимому, и послужило причиной катастрофы. Во время планового резервного

копирования драйвер лентопротяжки "впал в ступор" и заблокировал процесс копирования (механизм возникновения этой аварии подробнее обсуждался в разд. 10.6.4). Из-за наличия зависшего процесса оказалось невозможно размонтировать файловую систему, и машина была перезагружена нажатием кнопки <RESET> без выполнения нормального закрытия, в том числе и без выполнения операции закрытия драйвера дискового массива, которая должна была сбросить на диски содержимое буферов контроллера.

После перезагрузки система автоматически запустила программу восстановления файловой системы fsck (File System Check), которая выдала радостное сообщение: DUP in inode 2. Для незнакомых с системными утилитами Unix необходимо сказать, что DUP означает ошибку перекрещивания файлов, а inode 2 — это инод корневого каталога ФС. Таким образом, корневая директория дискового тома объемом около 2 Гбайт оказалась испорчена. При этом подавляющее большинство каталогов и файлов были не затронуты катаклизмом, но оказались недоступны. Программа восстановления не могла перенести ссылки на соответствующие иноды в каталог lost+found, т. к. ссылка на него также идет из корневого каталога.

Ситуация представлялась безвыходной. Катастрофа усугублялась тем, что произошла она во время резервного копирования, а последняя "хорошая" копия была сделана неделю назад. Большую часть пользовательских данных можно было бы восстановить, но среди потерянных файлов оказался журнальный файл сервера СУБД Oracle (сама база данных находилась в другом разделе диска). Пришлось заняться восстановлением ФС с использованием дискового редактора, мотивируя это тем, что "терять все равно уже нечего". Собственно, в восстановлении ФС участвовало два человека — я и штатный администратор системы. Я ни в коем случае не хочу создать у читателя впечатление, что план восстановления был разработан лично мной — это был плод совместных усилий, проб и ошибок.

Редактирование системных данных "сложных" ФС с использованием простого шестнадцатеричного дискового редактора является крайне неблагодарным занятием. Есть основания утверждать, что это вообще невозможно. Во всяком случае, мне не доводилось слышать об успехе такого предприятия. К счастью, системы семейства Unix предоставляют для редактирования ФС специальную программу fsdb (File System DeBugger — отладчик файловой системы). Пользовательским интерфейсом эта программа напоминает программу DEBUG.COM, поставляемую с MS DOS; главным ее преимуществом является то, что она "знакома" с основными понятиями файловой системы. Так, например, fsdb позволяет просмотреть содержимое 10-го логического блока файла с инодом 23 456, выделить физический блок 567 345 файлу с инодом 2 или пометить инод 1245 как свободный.

Первая попытка восстановления состояла в том, что мы удалили тот инод, с которым перекрецивался корневой каталог, смонтировали том командой "безусловного" монтирования (которая позволяла монтировать поврежденные тома), создали командой mkdir каталог lost+found и вновь запустили fsck. Попытка завершилась крахом. Беда была в том, что, как оказалось, корневой каталог пересекался также и со списком свободных блоков, т. е. создание каталога, а потом его расширение командой fsck снова приводило к порче корневого каталога и задача сводилась к предыдущей.

Таким образом, нам необходимо было либо исправить вручную список свободных блоков либо найти способ создать директорию lost+found без обращений к

этому списку. Дополнительная сложность состояла в том, что с каталогом `lost+found` не связано фиксированного инода, а определить инод старого `lost+found` не представлялось возможным.

Мы решили не связываться с восстановлением списка свободных блоков. Вместо этого мы просмотрели листинг последней "правильной" резервной копии, нашли там ненужный пустой каталог и присоединили его к корневому под названием `lost+found`. После этого нам оставалось лишь уповать на то, что вновь создаваемые `fsck`-ссылки на файлы не приведут к необходимости удлинить наш `lost+found`. К счастью, этого не произошло: все потомки корневого каталога благополучно получили имена в `lost+found`. По существу, ФС пришла в пригодное для чтения состояние, оставалось лишь правильно определить имена найденных каталогов. Это также оказалось относительно несложной задачей: большая часть каталогов на томе состояла из домашних каталогов пользователей, и их имена можно было восстановить на основании того, кому эти каталоги принадлежали. Для остальных каталогов имя достаточно легко определялось после сопоставления их содержимого с листингом резервной копии.

Проверка ФС занимает много времени; реальное время, впрочем, сильно зависит не только от объема диска, но и от скорости процессора, объема ОЗУ и производительности дискового канала. Так, мой компьютер, приобретенный в 1997 году (*Celeron 300, 196 Мбайт ОЗУ, диск с интерфейсом EIDE UDMA/66*), проверял том HPFS объемом 12 Гбайт около десяти минут. Сопоставимое время занимало восстановление ext2fs того же объема на машине сопоставимой конфигурации. Разумеется, я был убежден, что время традиционных ФС ушло и, как только появилась возможность, переразметил основной объем диска под журнальную файловую систему JFS. Когда в 2004 году я приобрел новый компьютер (*Pentium IV 2.4 ГГц, 512 Мбайт ОЗУ, диск EIDE UDMA/150*), я обнаружил, что том HPFS объемом 10 Гбайт на новой машине проверяется не более двадцати секунд, лишь ненамного дольше, чем происходит репликация журнала у JFS!

Так или иначе, для серверов с дисками большого объема (в современных условиях это многие сотни гигабайт или терабайты) и с высокими требованиями к времени восстановления после аварии, проверка ФС, представляет значительную проблему, даже если такие серверы "падают" относительно редко. Напротив, рабочие станции и домашние компьютеры, даже работающие под управлением защищенных ОС, "падают" относительно часто, потому что на них используется дешевое (и не всегда качественное) оборудование, а программное обеспечение не всегда правильно и аккуратно настроено; статистически, "падения" от этих причин сопоставимы по частоте со сбоями электрического питания в типичном жилом или офисном здании. Разумеется, длительное восстановление после таких событий приводит к потерям рабочего времени и раздражает пользователей.

Некоторые традиционные ФС используют различные приемы для сокращения времени проверки. Так, реализации UFS в BSD и Solaris поддерживают

собственный dirty-флаг для каждого блока цилиндров, а не для ФС в целом. После аварийного размонтирования проверяются только те блоки цилиндров, которые подвергались модификации, т. е., как правило, небольшая часть тома.

Во многих современных ОС реализованы устойчивые к сбоям файловые системы: jfs в AIX, OS/2 v4.5 и Linux, ext3fs в Linux, журнальная версия UFS в Solaris, Veritas в UnixWare и Solaris, NTFS в Windows NT/2000/XP. Практически все такие ФС основаны на механизме, который по-английски называется *intention logging* (*регистрация намерений*).

11.4.3. Файловые системы с регистрацией намерений

Термин, вынесенный в заголовок этого подраздела, является дословной калькой (возможно, не очень удачной) англоязычного термина *intention logging*. В русском языке, к сожалению, еще нет общепринятого термина для этого понятия. Далее мы будем называть регистрацию намерений *журнализированием* (*journaling*), а файловые системы, использующие этот механизм, — *журнальными*.

Идея журналов регистрации намерений пришла из систем управления базами данных. В СУБД часто возникает задача внесения согласованных изменений в несколько разных структур данных. Например, банковская система переводит миллион долларов с одного счета на другой. СУБД вычитает 1 000 000 из суммы на первом счету, затем пытается добавить ту же величину ко второму счету... и в этот момент происходит сбой.

Для СУБД этот пример выглядит очень тривиально, но мы выбрали его потому, что он похож на ситуацию в файловой системе: в каком бы порядке ни производились действия по переносу объекта из одной структуры в другую, сбой в неудачный момент приводит к крайне неприятной ситуации. В СУБД эта проблема была осознана как острая очень давно — ведь миллион долларов всегда был намного дороже одного сектора на диске...

Удовлетворительное решение проблемы заключается в следующем:

- во-первых, все согласованные изменения в СУБД организуются в блоки, называемые транзакциями (*transaction*). Каждая транзакция осуществляется как неделимая (атомарная) операция, во время которой никакие другие операции над изменяемыми данными не разрешены;
- во-вторых, каждая транзакция осуществляется в три этапа (рис. 11.22):
 - система записывает в специальный журнальный файл, что же она собирается делать;
 - если запись в журнал была успешной, система выполняет транзакцию;

- если транзакция завершилась нормально, система помечает в журнале, что намерение было успешно реализовано.

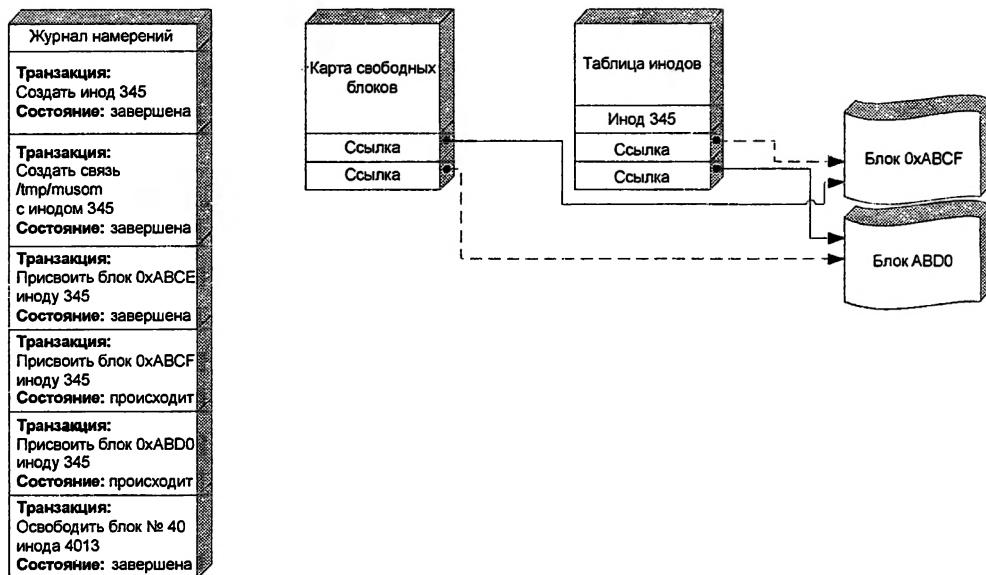


Рис. 11.22. Выполнение транзакции с регистрацией намерений

Журнал часто называют *журналом регистрации намерений* (*intention log*), что очень хорошо отражает суть дела, потому что в этот журнал записываются именно намерения (*intentions*).

При использовании отложенной записи транзакция считается полностью завершенной, только когда последний блок измененных данных будет физически записан на диск. При этом в системе обычно будет одновременно существовать несколько незавершенных транзакций (рис. 11.23). Легко понять, что при операциях с самим журналом отложенную запись вообще нельзя использовать.

Если произошел сбой системы, после перезагрузки запускается программа восстановления базы данных (рис. 11.24). Эта программа просматривает конец журнала:

- если она видит там испорченную запись, то игнорирует ее: сбой произошел во время записи в журнал;
- если все записи помечены как успешно выполненные транзакции, то сбой произошел между транзакциями: ничего особенно страшного не случилось; во всяком случае, ничего исправлять не надо.

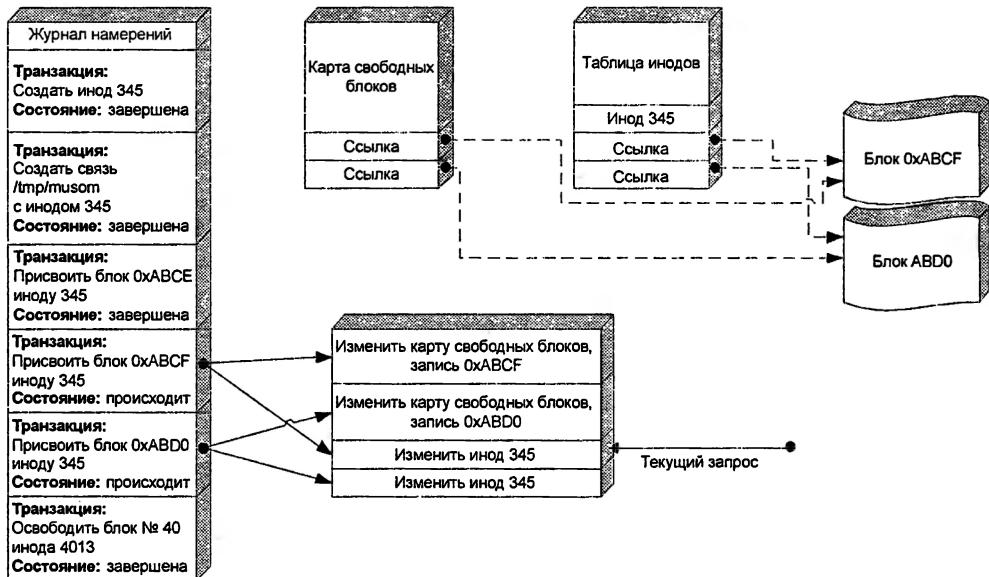


Рис. 11.23. Очередь исполняющихся транзакций

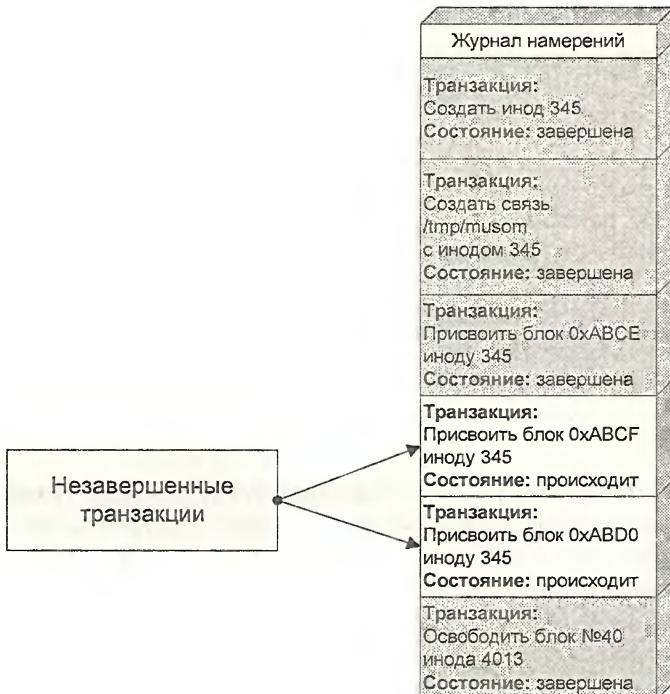


Рис. 11.24. Журнал транзакций после сбоя

- если найдена запись, которая отмечает начатую, но невыполненную транзакцию, то сбой произошел во время этой транзакции. Это наиболее неприятная ситуация, но журнал содержит достаточно информации для того, чтобы или восстановить состояние базы данных до начала транзакции (выполнить *откат* назад (*rollback*)), или же доделать предполагавшиеся изменения.

Нужно отметить, что данные, необходимые для выполнения отката, могут иметь большой объем. Фактически, это копия всех данных, подвергающихся изменению в ходе транзакции. Многие СУБД, такие как Oracle, хранят эти данные не в самом журнале, а в специальной области данных, называемой *сегментом отката* (*rollback segment*).

Более подробная информация о работе журналов намерений в базах данных может быть найдена в соответствующей литературе [Дейт 1999, Дейт 1988]. Необходимо только отметить, что книги и даже фирменная документация по простым СУБД типа dBase или FoxPro здесь не помогут, поскольку эти пакеты не содержат средств регистраций в журнале.

Идея журнала намерений достаточно естественно переносится в программу управления файловой системой. Но здесь возникает интересный вопрос — что же считать транзакцией: только операции по распределению пространства на диске или также все операции по изменению данных?

Первый вариант проще в реализации и оказывает меньшее влияние на производительность; зато он гарантирует только целостность самой ФС, но не может гарантировать целостности пользовательских данных, если сбой произойдет в момент записи в файл. Список ФС, поддерживающих транзакции при работе с системными данными, довольно велик. Среди наиболее распространенных ФС следует назвать NTFS в Windows NT и JFS, поддерживаемую в IBM AIX, IBM OS/2 и Linux. В последние годы получили распространение "совместимые" реализации журнальных ФС, которые совместимы по структуре данных на диске с более старыми нежурналируемыми системами. При этом для размещения журнала используются зарезервированные типы инодов. Именно так реализовано журналирование в ext3fs в современных версиях ядра Linux, и в версии UFS, используемой в Sun Solaris, начиная с версии 8.

"Совместимые" журнальные ФС очень привлекательны, потому что администратор системы может перейти к использованию журнала, не переформатируя диск. Более того, если окажется, что журнал оказывает негативное влияние на производительность или в реализации журнала будут обнаружены опасные ошибки, можно будет выключить журналирование — опять-таки без переразметки диска.

На вопросе об ошибках в реализации ФС следует остановиться отдельно. Я уже отмечал, что журнальная ФС, вообще говоря, не гарантирует целост-

ности системных данных и даже не так уж сильно снижает вероятность их порчи по сравнению с традиционными ФС, которые проверяются во время монтирования при установленном dirty-флаге. Единственное, что журналирование действительно обеспечивает, — это восстановление ФС после сбоя зафиксированное и, на практике, за относительно небольшое время. К сожалению, в журнальных ФС есть важный фактор, который повышает вероятность потери системных данных по сравнению с традиционными файловыми системами.

А именно, в традиционных ФС время от времени производится полная проверка всех системных данных. При этом устраняются не только ошибки, возникшие при аварийных размонтированиях томов, но и рассогласования данных, возникшие при штатной работе из-за ошибок программирования при реализации ФС. Напротив, в журнальной ФС программа восстановления полагается на то, что реализация ФС работает полностью корректно. Поэтому любая ошибка в реализации может приводить к катастрофическим последствиям. При этом необходимо помнить также, что код и дисковые структуры данных большинства журнальных ФС значительно сложнее, чем у традиционных ФС; ошибки могут возникать не только при модификации основной копии метаданных, но и при записях в журнал, поэтому вероятность таких рассогласований в журнальной ФС значительно, как минимум в несколько раз, выше, чем в традиционной нежурналируемой системе.

Практика подтверждает это рассуждение — внедрение журнальных ФС, в том числе NTFS и JFS, сопровождалось катастрофами, которые нередко приводили к полной потере данных, как пользовательских, так и системных. Форумы системных администраторов середины 90-х годов XX века просто наполнены такими историями. У Windows NT проблема сошла на нет приблизительно ко времени выхода Windows NT 4.0 SP2; у OS/2 — к 2000 или 2001 году; во всяком случае, приблизительно в это время в форумах исчезли крики о помощи, свидетельствующие о новых катастрофах, хотя воспоминания о них циркулируют до сих пор (ни один добросовестный администратор такого, конечно, забыть не может!).

Разумеется, в этих условиях "совместимые" журнальные ФС получают значительное преимущество — ведь администратор не только может отказаться от журналирования, но и может восстановить поврежденную ФС с помощью традиционных утилит полной проверки. Большинство дистрибутивов Linux настраивают ext3fs так, чтобы запускать полную проверку томов ext3fs при каждом десятом монтировании.

JFS (Journalled File System)

Файловая система JFS была разработана компанией IBM в первой половине 90-х годов для ОС AIX (версии Unix, основанной частично на кодах OSF Unix, частично на Unix System V Release 3), поставлявшейся с миникомпьютерами и

рабочими станциями на основе процессоров Power и, позднее, PowerPC. Во второй половине 90-х для OS/2 была разработана новая реализация JFS, так называемая JFS2, в которой в структуру данных на диске были внесены значительные изменения (в частности, старый JFS использовал линейные каталоги, а JFS2 использует отсортированные каталоги в виде B+-деревьев). Эта реализация вошла в состав OS/2 4.5 (Warp Server for eBusiness) и затем портирована обратно на AIX. В 2000 году IBM опубликовала исходные тексты JFS2 под лицензией GPL и портировала их для работы в ядре Linux. Успех JFS/Linux трудно назвать блестящим — этой ФС приходится выдерживать конкуренцию с "совместимой" ext3fs, переход на которую не требует переразметки томов.

Относительно серьезным недостатком JFS/Linux следует признать стратегию кэширования: в OS/2 и AIX управление дисковым кэшем осуществляет драйвер файловой системы. Linux использует централизованно управляемый дисковый кэш, общий для всех файловых систем. Таким образом, значительная часть работы по портированию JFS под Linux состояло в "откручивании" управления кэшем; при этом стратегия работы ФС с данными оказалась неоптимальной и большинство тестов, в которых JFS/Linux сравнивались с ext3fs или другими журнальными ФС (например, XFS), демонстрировали ее относительно низкую производительность. Напротив, пользователи JFS, работающие под OS/2 и AIX на производительность не жалуются. Для сравнения, портированные из Linux под OS/2 реализации ext2fs и FAT32 также страдают от неэффективного кэширования, что особенно заметно при записи больших объемов данных. Приблизительно в 2003 году, в JFS/Linux v 1.1.15 этот недостаток был устранен; во всяком случае, тесты, результаты которых опубликовала IBM, демонстрировали превосходство этой версии JFS над ext2fs в среднем на 10% на большинстве операций.

Тем не менее JFS следует рассмотреть в качестве примера хотя бы уже потому, что это, пожалуй, лучше всего документированная система среди ФС, разработанных в 90-е годы. Многие из реализованных в ней идей имеют аналоги в разрабатывавшихся приблизительно в это же время NTFS и Veritas.

JFS может размещаться на нескольких дисках и динамически добавлять и исключать диски без полной переразметки ФС. Эта возможность особенно удобна при использовании SAN, в которых дисковые массивы разделяются между несколькими серверами и динамически перераспределяются между ними.

JFS предполагает, что физические разделы дисков объединены в единый логический том, называемый агрегатом (aggregate). Это объединение возлагается на промежуточный между ФС и драйверами дисков слой, называемый LVM (Logical Volume Manager — менеджер логических томов). В агрегатах создаются блоки системных структур данных, наборы файлов (fileset). Теоретически, структура данных допускает несколько файловых наборов на агрегат (это могло бы соответствовать нескольким независимым ФС или несколькими деревьям каталогов для одной ФС, наподобие каталогов Joliet и Rockridge в файловых системах ISO 9660), но существующая реализация поддерживает только один набор в каждом агрегате.

Агрегат разбит на логические блоки определенного размера. Поддерживаются блоки размером 512 байт (соответствует блоку подавляющего большинства серийных жестких дисков), 1024, 2048 или 4096 байт. По умолчанию тома создаются с размером блока 4096 байт, потому что это уменьшает размеры системных структур данных; возникающая при этом внутренняя фрагментация считается приемлемой.

Для каждого агрегата создается заголовок (рис. 11.25), который содержит:

- 32к зарезервированного пространства;
- первичный суперблок агрегата (Primary Aggregate Superblock), в котором описывается размер агрегата, размер блока и другие параметры. В действительности эта структура данных занимает 4 Кбайт;

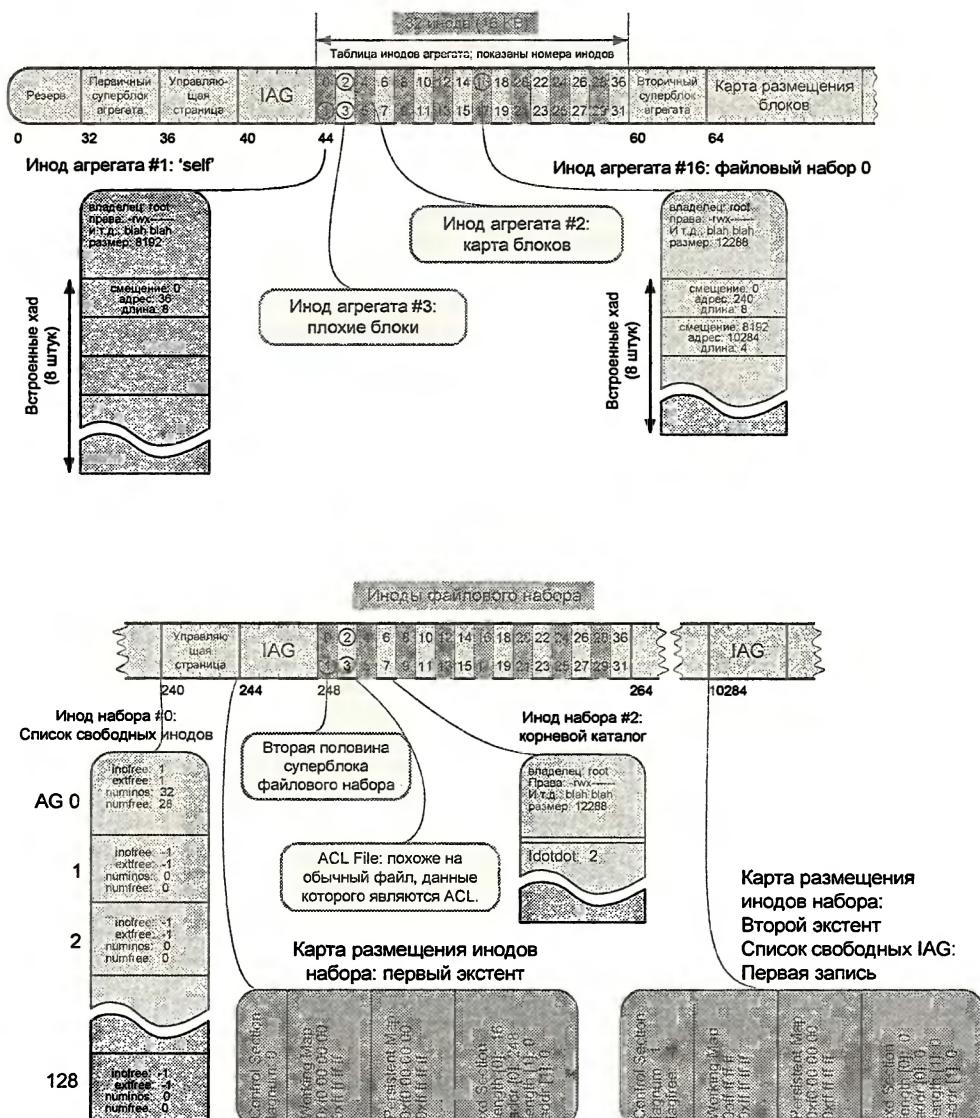


Рис. 11.25. Заголовок агрегата и начало таблицы инодов файлового набора JFS2

- вторичный суперблок агрегата (Secondary Aggregate Superblock), который представляет собой полную копию первичного суперблока;
- указатель на рабочее пространство fsck, которое может использоваться, если fsck не хватает оперативной памяти при восстановлении большого тома;
- таблицу инодов агрегата (Aggregate Inode Table) и ее реплику (Secondary Aggregate Inode Table, вторичная таблица инодов агрегата). В действительности, обе таблицы динамические. Первый экстент первичной таблицы имеет фиксированный размер и расположен в логических блоках с 44 по 69. Размещение соответствующего первого экстента вторичной таблицы указано в суперблоке. Этот экстент содержит тридцать два инода, двенадцать из которых зарезервированы;
- карту инодов агрегата (Aggregate Inode Map);
- инод 0 зарезервирован;
- инод 1 (self), который описывает размещение таблицы инодов по агрегату. Как уже отмечалось, первый экстент таблицы размещается в фиксированной области диска, поэтому драйвер ФС всегда может найти его;
- инод 2 (Block Allocation Map) описывает размещение битовой карты свободных и занятых блоков. Один бит этой карты соответствует одному логическому блоку агрегата. Эта карта создается и размещается при разметке агрегата; ее размеры могут изменяться при изменении размеров агрегата;
- инод 3 (In-Line Log) описывает размещение лога транзакций;
- инод 4 (Bad Blocks) описывает размещение дефектных блоков, обнаруженных при разметке агрегата и вновь обнаруживаемых при работе ФС;
- иноды с 5 по 15 зарезервированы;
- иноды с номерами 16 и более содержат описания файловых наборов, размещенных в агрегате.

Поскольку таблица инодов агрегата содержит только иноды файловых наборов и общих для агрегата структур данных, а современная JFS поддерживает только один набор на агрегат, иноды с номерами более 16 никогда не используются, так что таблица инодов агрегата всегда помещается в одном экстенте. Тем не менее возможность ее расширения предусмотрена. Ни один из инодов, перечисленных в таблице инодов агрегата, не имеет имени в файловой системе, но по структуре они, в основном, аналогичны инодам файловых объектов (эта структура вкратце рассматривается далее).

Остальное пространство агрегата содержит файловые наборы и разбито на логические группы размещения (Allocation Group), приблизительно соответствующие группам цилиндров UFS и ext2fs. Каждая AG занимает непрерывную область дискового пространства. Размер AG в логических блоках должен быть кратен степени двойки; допускается не более 128 AG на агрегат. Разумеется, размер агрегата в целом может быть не кратен степени двойки; при этом последняя AG будет неполной. Размер всех AG одинаков и указан в суперблоке ФС; неполная AG логически имеет такой же размер, но отсутствующие блоки описаны в ее битовой карте как занятые. При изменении размеров агрегата может произойти переразбиение ФС на AG:

Описание файлового набора, в действительности, представляет собой таблицу инодов этого набора (см. рис 11.25). Таблица инодов файлового набора дина-

мически размещается на диске и может изменять размер без переразметки тома. Она размещается как последовательность экстентов (непрерывных областей диска); количество экстентов теоретически не ограничено; их список хранится в виде сортированного B+-дерева в иноде из таблицы инодов агрегата (точнее, в самом иноде хранится только корень B+-дерева, а для узлов этой структуры используются блоки данных).

Нулевой инод файлового набора зарезервирован, первый инод хранит дополнительную информацию о файловом наборе, второй инод содержит корневой каталог, третий инод — ACL файлового набора, последующие иноды описывают обычные файлы, каталоги, символические связи и другие обычные объекты ФС. Размещение инодов в динамической таблице позволяет "отвязать" нумерацию инодов от их размещения по диску; это особенно удобно при изменении размеров тома и/или при переразмещении таблицы инодов, которое неизбежно при исключении физических дисков из логического тома.

Ряд других ФС, использующих динамическую таблицу инодов, в том числе NTFS, допускают расширение этой таблицы (у NTFS она называется MFT — Main File Table), но никогда не сжимают ее. Действительно, сжатие такой таблицы может быть сложной задачей из-за ее фрагментации. Это приводит к определенной проблеме — создав много коротких файлов и затем удалив их, злоумышленник может уничтожить часть объема ФС. JFS допускает сжатие таблицы инодов, как с конца (это приводит к тому, что часть инодов с большими номерами становится недоступна), так и из середины. При этом таблица инодов становится разреженной: некоторые ее экстенты логически "существуют", но места под их описатели на диске не выделено. Дефрагментация пространства номеров инодов никогда не производится, поэтому ссылки на иноды из каталогов никогда не приходится обновлять, даже во время глобальных реорганизаций агрегата, необходимых при уменьшении его физического размера.

JFS пытается размещать иноды файлов как можно ближе к блокам данных этого файла. На практике, это обеспечивается за счет того, что JFS ищет свободное пространство и свободные иноды в пределах одной AG, а если их там нет, то выделяет новые экстенты таблицы инодов в той же AG. Для ускорения такого поиска используется карта размещения инодов (Inode Allocation Map), которая имеет довольно сложную структуру, но по сути своей представляет список номеров инодов, размещенных в определенной AG; карта размещения инодов хранит также битовую маску, описывающую, какие из инодов свободны.

Структура инода в основном аналогична структуре инодов в других ФС семейства Unix, но можно усмотреть и некоторые аналогии с HPFS. Дисковый образ инода занимает 512 байт и содержит:

- атрибуты файлового объекта, требуемые стандартом POSIX (идентификаторы хозяина и группы, тип объекта, 12-битную маску прав и атрибутов доступа, длину файла в байтах, даты создания, последней модификации и последнего доступа к файлу, счетчик жестких связей и т. д.);
- атрибуты объекта, специфические для ОС, например, расширенные атрибуты OS/2 и/или список управления доступом;
- корень B+-дерева экстентов или данные объекта (короткие файлы, каталоги или символические связи размещаются прямо в иноде без выделения дополнительных экстентов).

Каждый экстент описывается структурой `xad`, ее вид показан в примере 11.4.

Пример 11.4. Структура xad, цит. по [www6.software.ibm.com/jfs-layout]

```
struct xad {
    unsigned     flag:8;
    unsigned     rsvrd:16;
    unsigned     off1:8;
    uint32       off2;
    unsigned     len:24;
    unsigned     addr1:8;
    uint32       addr2;
} xad_t;
```

Поле флагов содержит различные атрибуты экстента; например, биты этого поля могут указывать, что экстент принадлежит к разреженному файлу и для него не выделено дискового пространства. Пространство будет выделено только при записи в этот экстент; при чтении же из него будут читаться нули.

Поля off1 и off2 образуют единое 40-битное поле и описывают логическое смещение экстента от начала файла. Смещение отсчитывается в логических блоках; таким образом, при размере блока 4096 байт, максимальная длина файла может достигать $2^{52}-1$ байт.

Поля addr1 и addr2 также образуют единое 40-битное поле, представляющее собой первый логический блок экстента. Таким образом, размер агрегата также ограничен $2^{52}-1$ байтами.

24-битное поле len описывает длину экстента в логических блоках. Очевидно, что длина экстента на диске и его логического образа в файле должна быть одинакова. При блоке в 4096 байт максимальная длина экстента составляет 64 Гбайт. Экстент всегда расположен внутри одного агрегата, но может размещаться в нескольких AG одного агрегата.

Описатели экстентов образуют B+-дерево, отсортированное в соответствии с логическим смещением экстента в файле. Корневой узел B+-дерева должен размещаться внутри инода и имеет нестандартный размер (в нем размещается всего 8 описателей экстентов или ссылок на дочерние узлы). Если файл имеет менее восьми экстентов, то этот узел оказывается листьевым.

Таким образом, при произвольном доступе даже к сильно фрагментированному файлу (то есть состоящему из многих экстентов), необходимый описатель экстента находится довольно быстро. Впрочем, JFS использует размещение по принципу Worst Fit (вновь создаваемые файлы создаются в начале самого большого блока свободного пространства) и некоторые другие эвристики, так что большинство файлов на практике занимает всего несколько экстентов и не требуют создания B+-дерева как такого.

В поставку JFS входит дефрагментационная утилита, которая пытается переразместить файлы так, чтобы уменьшить количество занимаемых ими экстентов. Я должен признать, что ни разу не наблюдал измеримого увеличения скорости доступа к файлам после завершения работы этой утилиты.

Кроме экстентов данных, файл может иметь два специальных экстента, предназначенные для размещения списка управления доступом (ACL) и расширенных атрибутов OS/2. Если размер каждой из этих структур достаточно мал, они могут быть размещены в самом иноде.

Каталог JFS существенно отличается от обычного файла. Под него не выделяется экстентов данных; вместо этого B+-дерево каталога содержит сами записи каталога и отсортировано по именам этих записей. Имена хранятся в кодировке Unicode и могут иметь довольно большую длину, однако логика работы B+-дерева требует, чтобы записи имели фиксированный размер. Для этого в индексных блоках хранятся только короткие префиксы имен, достаточные, чтобы отличить имя от предыдущего и последующего имен в этом каталоге (рис. 11.26). В результате получается довольно сложная структура: каждый 4-килобайтный узел B+-дерева содержит отсортированную таблицу записей (Directory Slot Array), в которой хранятся префиксы имен и указатели на остаток записи, и фрагментированное пространство, в котором хранятся суффиксы имен. Д каталогах с большим количеством длинных имен значительная часть узлов B+-дерева будет иметь не полностью заполненную таблицу записей, но это считается умеренной платой за быстрый поиск имени в каталоге.

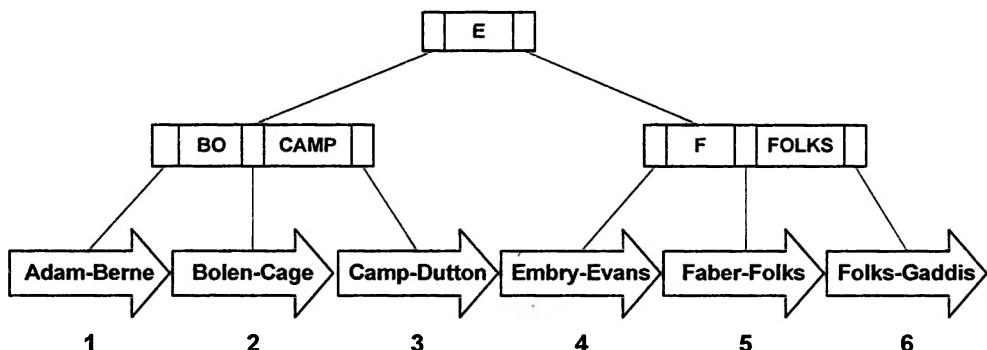


Рис. 11.26. B+-дерево каталога JFS

Создание или удаление большого количества файлов в одном каталоге приводит к многократной перебалансировке дерева, поэтому наблюдаемая производительность JFS в такой ситуации значительно ниже, чем у традиционных ФС семейства Unix с несортированными каталогами.

Лог транзакций формируется при создании файловой системы и его размер определяется автоматически пропорционально размеру тома. Реализации для Linux и AIX допускают создание лога на другом устройстве и разделение лога между несколькими томами; версия для OS/2 этого не допускает. Все версии JFS допускают выделение пространства для лога на самом томе. К записи в лог приводят следующие операции:

- создание файла;
- создание жесткого линка на файл;
- создание каталога;

- создание инода другого типа (например, именованной трубы или символьической связи);
- удаление файла;
- переименование файла;
- удаление каталога;
- изменение ACL и расширенных атрибутов, при условии, что это потребовало изменения размера соответствующих экстентов;
- запись в файл (при условии, что она привела к выделению новых экстентов и особенно к перебалансировке B+-дерева);
- отбрасывание части данных файла (*truncate*). Системы семейства Unix допускают отбрасывание данных не только из конца файла (что приводит просто к уменьшению длины файла), но и из его середины; при этом длина файла не изменится, но файл станет разреженным.

Все эти операции требуют модификации нескольких структур данных на диске. Так, создание файла требует:

- создания записи в каталоге (что может привести к перебалансировке B+-дерева каталога);
- модификации карты свободных блоков, если при этом пришлось увеличить размер каталога;
- поиска свободного инода и пометки его как занятого;
- модификации карты инодов;
- возможно, расширения таблицы инодов и, соответственно, модификации инода файлового набора и, также, модификации карты свободных блоков.

Все эти операции объединены в транзакцию. Начиная эту группу операций, JFS выбирает идентификатор транзакции (*tid*) и создает группу записей в логе. Такие записи имеют тип `LOG_REDOPAGE` и содержат информацию, необходимую для повтора операции — изменения в карте инодов, в карте свободных блоков, в B+-деревьях и т. д. Записи, принадлежащие к разным транзакциям, могут чередоваться, поэтому каждая запись содержит идентификатор транзакции. Последняя запись группы имеет тип `LOG_COMMIT` и означает, что описание транзакции завершено. В момент, когда запись `COMMIT` заносится в лог, модификации в рабочую копию структур данных на диске еще не вносились, но копии соответствующих структур данных в памяти (в дисковом кэше) уже модифицированы. Эти изменения позднее записываются на диск, но с более низким приоритетом, чем у других дисковых операций. Приоритет этих операций может повышаться по мере заполнения лога или оперативной памяти.

По мере записи модифицированных метаданных на диск, менеджер лога создает записи типа `LOG_SYNCPT` (*Synchronization Point* — точка синхронизации), отмечая, что записи сделанные до этой точки, не требуют переигрывания при восстановлении ФС. Аналогичный эффект имеют записи типа `LOG_MOUNT`: эта запись создается при монтировании тома, т. е. все расположенные до нее записи относятся к предыдущей сессии работы с ФС и либо были отыграны при штатном размонтировании тома, либо переиграны при восстановлении после нештатного размонтирования.

Суперблок JFS, как и суперблоки традиционных ФС, имеет *dirty*-флаг, который устанавливается при монтировании тома и сбрасывается при его штатном раз-

монтировании, после того как все модификации метаданных будут записаны на диск. При попытке монтирования тома с установленным dirty-флагом запускается программа fsck (в Linux и AIX) или CHKDSK (в OS/2), которая начинает восстанавливать том.

По умолчанию, fsck пытается переиграть записи в логе. Она анализирует записи, начиная от конца лога, т. е. в порядке, обратном тому, в котором они записывались. Найдя запись LOG_COMMIT, CHKDSK запоминает идентификатор транзакции и исполняет все операции, требуемые записями типа LOG_REDOPAGE с таким идентификатором транзакции. Записи LOG_REDOPAGE с идентификаторами, для которых не было найдено записи LOG_COMMIT, игнорируются. Когда обнаруживается запись LOG_SYNCPT или LOG_MOUNT, просмотр лога прекращается и том считается восстановленным.

Таким образом, при "отыгрывании" лога, утилита восстановления доделывает все транзакции, которые полностью описаны в логе, и игнорирует неполные записи о транзакциях.

В действительности, закончив "отыгрывание" лога, CHKDSK выполняет еще несколько проверок — например, на совпадение счетчиков свободных инодов в суперблоке и в карте инодов и т. д. Если какие-то из этих проверок завершаются неудачей, инициируется полная проверка тома, аналогичная процедуре восстановления традиционных ФС, и занимающая сопоставимое (а на практике даже несколько большее) время. Такая же проверка в обязательном порядке запускается, если сбой произошел во время изменения размеров агрегата — ведь при этом происходит полная перестройка битовой карты блоков и ряда других структур, т. е. операции, для описания которых не предусмотрено соответствующих типов записей в логе.

Кроме того, администратор системы может запросить полное восстановление тома, задавая соответствующие ключи программе CHKDSK; при этом он может потребовать либо отыгрывания лога, либо, наоборот, его игнорирования (это может быть полезно, если администратор имеет основания подозревать, что лог поврежден).

Включение в транзакции пользовательских данных требует выделения сегмента отката и сильно замедляет работу. Действительно, ведь теперь все данные, а не только метаданные ФС, пишутся на диск два раза: сначала в сегмент отката, а потом в рабочую копию данных. При этом, чтобы вероятность порчи данных действительно снизилась, прикладная программа при работе с ФС должна отмечать начала и концы транзакций. В некоторых случаях можно считать открытие файла началом транзакции, а его закрытие — концом. Однако при работе с базами данных и в ряде других ситуаций это привело бы к слишком большому объему изменяемых данных и к недопустимо большому сегменту отката.

Ряд современных ФС с регистрацией намерений поддерживают оба режима работы и предоставляют выбор между этими вариантами администратору системы. Например, у файловой системы vxfs или Veritas, входящей в пакет UnixWare (версия Unix SVR4, поставляемая фирмой Novell), существует две

версии. Одна версия, поставляемая вместе с системой по умолчанию, включает в транзакцию только системные данные. Другая, "advanced", версия, которая поставляется за отдельные деньги, осуществляет регистрацию намерений как для системных, так и для пользовательских данных.

В некоторых других системах, например, в Novell Netware, транзакционность включается для отдельных файлов (что характерно, это поддерживается на ФС, которые не поддерживают транзакции для системных данных ФС). С такими файлами необходимо работать с помощью специального транзакционного API.

Ряд современных ОС (Solaris, Linux, Windows 2003) поддерживают *контрольные точки* (чекпоинты, *checkpoint*) или, что в данном случае то же самое, *"моментальные снимки"* (*snapshot*) — механизм, который можно описать как упрощенную реализацию поддержки транзакций над пользовательскими данными.

Контрольные точки используются главным образом для резервного копирования данных. Во многих случаях остановка серверного приложения даже на короткое время бывает крайне нежелательна. Но для создания резервной копии весьма желательно, чтобы копируемые данные находились в согласованном состоянии. Поэтому поставщики приложений, особенно серверов баз данных, настоятельно не рекомендуют осуществлять резервное копирование при запущенной СУБД.

Создавая контрольную точку, администратор получает виртуальное устройство, соответствующее состоянию ФС на момент создания точки. Это устройство доступно только для чтения; его можно смонтировать как обычную ФС, но, разумеется, также в режиме только для чтения. При этом основная ФС может подвергаться модификации; старое содержимое модифицируемых блоков сохраняется в специальной области диска, которая создается в момент создания контрольной точки. Такому сохранению подвергаются как пользовательские, так и системные структуры данных. Контрольные точки часто реализуются не на уровне ФС, а на уровне LVM.

Если ОС предоставляет механизм контрольных точек, администратор может остановить сервер СУБД буквально на несколько секунд, создать контрольную точку и перезапустить сервер. Затем он может запустить резервное копирование виртуального устройства; при этом скопируется согласованное состояние файлов БД, существовавшее в момент остановки сервера. Разумеется, изменения, внесенные после перезапуска сервера, в резервную копию не попадут, но любые механизмы резервного копирования так или иначе предполагают отставание резервной копии от рабочих данных.

Все современные реализации контрольных точек страдают важным недостатком. А именно, если объем журнала контрольной точки оказывается недостат-

точным, система прекращает сохранение старых данных. Работа с основной ФС при этом не прекращается, но сохраненная копия состояния ФС неожиданно начинает изменяться. При этом, поскольку эта копия "потеряла" часть изменений, данные в ней могут оказаться сильно повреждены. Поэтому большинство реализаций в этой ситуации просто отключают доступ к сохранившему образу ФС. Таким образом, создавая контрольную точку, администратор должен как можно точнее оценить объем данных, которые могут быть изменены за время действия контрольной точки.

11.4.4. Устойчивость ФС к сбоям диска

Кроме общесистемных сбоев, ФС должна обеспечивать средства восстановления при физических сбоях диска. Наиболее распространенным видом таких сбоев являются нечитаемые — "плохие" (bad) — блоки, появление которых обычно связано с физическими дефектами магнитного носителя.

Быстрее всего "плохие" блоки возникают на гибких магнитных дисках, которые соприкасаются с головкой чтения/записи и из-за этого подвержены физическому износу и повреждениям. Кроме того, гибкие диски подвергаются опасным воздействиям и вне дисковода. Например, при внесе дискеты с улицы в теплое помещение на поверхности диска будет конденсироваться влага, а соприкосновение головки дисковода с влажным диском практически наверняка повредит магнитный слой.

Жесткие магнитные диски помещены в герметичный корпус и — в норме — не соприкасаются с головками дисковода, поэтому срок службы таких дисков намного больше. Появление одиночных плохих блоков на жестком диске скорее всего свидетельствует о заводском дефекте поверхности или же о том, что магнитный слой от старости начал деградировать.

Весьма опасной причиной порчи жестких дисков является соприкосновение головок чтения/записи с поверхностью вращающегося диска (head crash), например, из-за чрезмерно сильных сотрясений диска во время работы. В частности, из-за этого не следует переставлять работающие компьютеры, особенно во время активных операций с диском. Обычно такое соприкосновение приводит к повреждению целой дорожки или нескольких дорожек диска, а зачастую и самой головки. Для несъемных жестких дисков это нередко означает потерю целой рабочей поверхности: считывание данных с нее требует замены блока головок, что весьма дорого.

Обычно ошибки данных обнаруживаются при чтении. Дисковые контроллеры используют при записи кодировку с исправлением ошибок, чаще всего коды Хэмминга (см. разд. 1.7), которые позволяют обнаруживать и исправлять ошибки. Тем не менее, если при чтении была выявлена ошибка, боль-

шинство ОС отмечают такой блок как плохой, даже если данные удалось восстановить на основании избыточного кода.

В файловой системе FAT плохой блок или кластер, содержащий такой блок, отмечается кодом 0xFFB или 0xFFFFB для дисков с 16-разрядной FAT. Эта файловая система не способна компенсировать плохие блоки в самой FAT или в корневом каталоге диска. Такие диски просто считаются непригодными для использования.

В "сложных" файловых системах часто используется более сложный, но зато и более удобный способ обхода плохих блоков, называемый *горячей заменой* (*hotfixing*). При создании файловой системы отводится небольшой пул блоков, предназначенных для горячей замены. В файловой системе хранится список всех обнаруженных плохих блоков, и каждому такому блоку поставлен в соответствие блок из пула горячей замены (рис. 11.27). При этом плохие блоки, на которые оказались отображены системные структуры данных, например участок таблицы инодов, также подвергаются горячей замене. Таблица горячей замены может быть как статической, так и динамической.

Современные контроллеры жестких дисков часто сами реализуют горячую замену блоков, осуществляя ее незаметно для центрального процессора.

На первый взгляд, динамическая таблица горячей замены предпочтительна, однако не нужно забывать о двух немаловажных факторах:

- файловая система вынуждена использовать для справки таблицу горячей замены при всех обращениях к диску, поэтому увеличение таблицы приводит к замедлению работы;

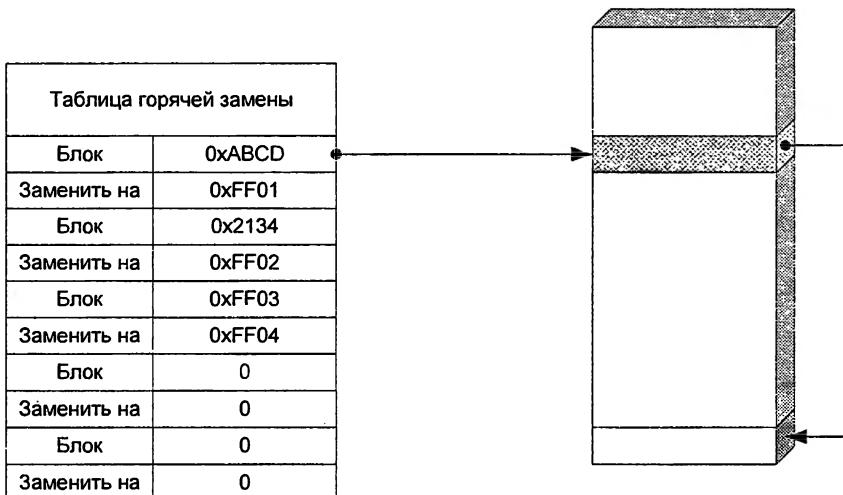


Рис. 11.27. Горячая замена (динамическое переназначение) блоков диска

- множественные плохие блоки на жестком диске свидетельствуют либо о том, что диск дефектный, либо о том, что магнитный слой начал разрушаться от старости (иными словами, что этот диск пора выбрасывать), либо о каких-то других не менее серьезных проблемах, вроде разгерметизации корпуса и проникновения в него пыли.

С учетом обоих факторов кажется целесообразным установить предел количества плохих блоков, после достижения которого диск нуждается в замене. Следует отметить также, что этот предел не может превышать нескольких процентов общей емкости диска. В свете этого небольшая статическая таблица блоков горячей замены представляется вовсе не такой уж плохой идеей.

11.5. Файловые системы с копированием при записи

В последние годы появился новый подход к организации файловых систем, позволяющий более естественным образом, нежели это возможно в традиционных и журнальных ФС, реализовать как транзакционность при работе с метаданными, так и контрольные точки при работе с пользовательскими данными. Подход этот состоит в использовании механизма, который мы несколько раз упоминали в предыдущих главах, а именно — в копировании данных при записи (copy-on-write).

В традиционной ФС (журнальные ФС в этом смысле тоже являются традиционными), если пользовательская программа модифицирует блок данных файла, ФС записывает новый блок поверх старого. Если ранее был создан чекпойнт, ФС или LVM копируют старый блок в журнал чекпойнта. Это позволяет избежать изменения метаданных при модификации файлов: поскольку новый блок находится в том же месте, что и старый, не надо обновлять структуры данных инода, ссылающиеся на этот блок.

Подход write-anywhere (дословно, запись повсюду) состоит в том, чтобы записывать каждый модифицированный блок данных в новое место. При этом, разумеется, придется обновлять и инод файла. Последовательно применяя принцип копирования при записи, мы должны не трогать старую копию инода, а вместо этого создать новый инод. При этом мы также должны обновить структуру метаданных, в которой описывается размещение таблицы инодов по диску (в JFS это инод файлового набора). Она обновляется тем же способом, т. е. старое значение этой структуры сохраняется на старом месте, а новое значение записывается в новый, ранее свободный, блок. Таким образом, изменение распространяется вверх по дереву метаданных, и при этом мы все время сохраняем копию старых данных. При системном сбое мы можем про-

сто откатиться к ней. Либо, если ФС поддерживает журнал вносимых изменений, мы можем повторно отработать все эти изменения.

Окончательное обновление ФС происходит, когда мы обновляем объект метаданных, находящийся на вершине их иерархии, который нельзя перемещать, т. е., скорее всего, суперблок. Сохранив копию старого суперблока, мы можем получить снимок файловой системы.

Определенную сложность при таком подходе к модификации данных представляет управление свободным пространством. Скорее всего, необходимо поддерживать несколько копий карты свободных блоков, тем или иным способом переключаясь между ними по мере модификаций суперблока.

Первой относительно успешной попыткой реализовать подход *write-anywhere* в реализации файловой системы следует, по-видимому, считать экспериментальную ФС TransArc Episode [Anderson et al 1992], которая не нашла практического применения. Первая коммерчески успешная реализация данного подхода — это реализованная также в первой половине 90-х годов файловая система WAFL (Write-Anywhere Filesystem Layout — структура файловой системы с записью повсюду) компании NetApp.

NetApp WAFL

Файловая система WAFL применяется в своеобразных устройствах, производимых компанией, так называемых "аппаратных файловых серверах". Эти устройства представляют собой файловые серверы с закрытой ОС, предоставляющие доступ к своим дискам по протоколу NFS.

Структура WAFL детально не документирована в открытых источниках, однако есть несколько публикаций, в которых в общих чертах описана архитектура ФС и применение *write-anywhere* для создания контрольных точек и транзакционного хранения метаданных.

Файловые серверы NetApp имеют высокоскоростную энергонезависимую оперативную память, которая используется для хранения лога транзакций. Вместо записи в лог операций по перераспределению пространства на диске, устройства NetApp записывают в этот лог запросы протокола NFS. Утверждается, что это приводит к значительной экономии объема лога, и для многих операций — например, для переименования файла — это, безусловно, справедливо.

WAFL использует многоуровневую древовидную структуру, в которой все крупные объекты метаданных, такие как таблица инодов и карта свободных блоков, сами описываются инодами и могут динамически расширяться и занимать произвольные области дискового пространства (рис. 11.28). Роль суперблока играет запись, называемая корневым инодом (*root inode*), — не следует путать ее с корневым каталогом ФС.

Структура инодов похожа на используемую в традиционных ФС семейства Unix, таких как UFS, с тем лишь отличием, что уровень косвенности у всех блоков одинаков. Инод WAFL хранит 16 указателей на блоки. Для файлов объемом менее 64 Кбайт, эти указатели непосредственно указывают на блоки данных файла (размер логического блока в WAFL фиксирован и всегда составляет

4 Кбайт). Файлы размером от 64 Кбайт до 64 Мбайт используют блоки одинарной косвенности, файлы большего размера — блоки двойной косвенности и т. д.

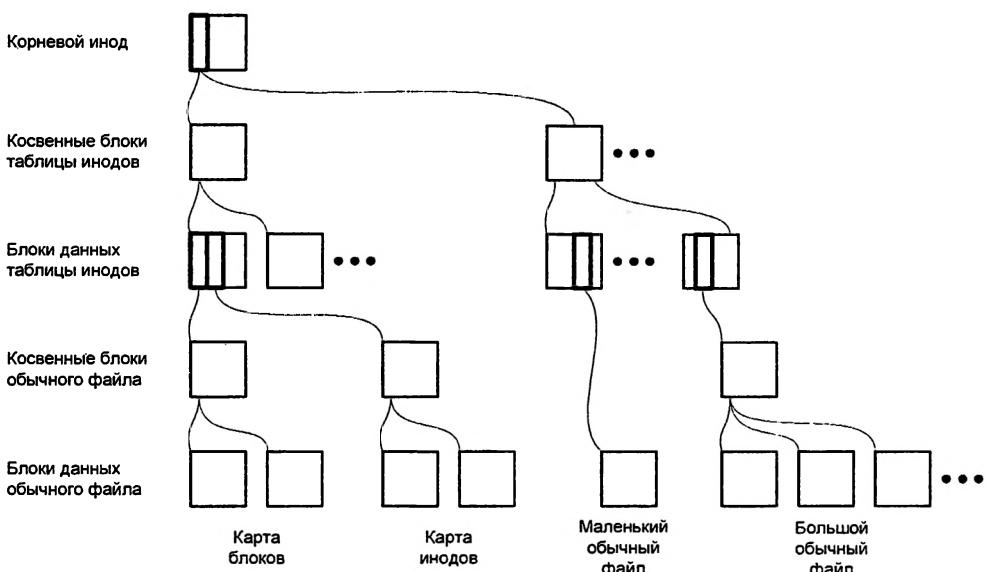


Рис. 11.28. Структура метаданных NetApp WAFL

Таким образом, все метаданные WAFL можно представить как древовидную структуру инодов и их косвенных блоков. Каждая модификация файла или метаданных приводит к созданию новой копии блока данных и распространению изменения вверх по этой древовидной структуре. Рано или поздно это изменение доходит до корневого инода. Сохранение корневого инода приводит к созданию нового снимка (рис. 11.29).

WAFL периодически создает копии корневого инода, соответствующие моментальным снимкам состояния ФС. Каждые несколько секунд создается специальный снимок, называемый точкой согласования (consistency point). При создании точки согласования все запросы NFS, исполненные к этому моменту, помечаются в логе транзакций как исполненные. Таким образом, при аварийной перезагрузке сервера оказывается достаточно взять за основу последнюю точку согласования и заново отработать все сохраненные в логе и непомеченные запросы.

Кроме точек согласования, WAFL создает до четырех "часовых" (hourly — словно, ежечасные, но это не совсем корректно, т. к. они создаются вовсе не каждый час) снимков в течении дня, по одному — каждый день (в полночь) и по одному — каждую неделю (в полночь с субботы на воскресенье). Срок жизни "часовых" снимков составляет два дня, ежедневных — неделю, а еженедельных — две недели. Администратор системы может изменить расписание моментов создания снимков и сроки их жизни, а также создавать снимки самостоятельно в произвольные моменты. WAFL может содержать до 255 снимков

одновременно, хотя, разумеется, реальное количество снимков ограничено также объемом дисков и объемами данных, изменявшихся в промежутках между снимками.

Кроме легкости создания снимков ФС, политика write-anywhere имеет еще одно своеобразное преимущество, особенно ощутимое при использовании дисковых массивов: поскольку ФС действительно может записывать данные куда угодно, она может объединять группы записей таким образом, чтобы они попадали в один столбец дискового массива. При использовании RAID уровней 4 и 5 это может обеспечить значительную экономию на модификациях контрольной суммы.

Стратегия управления картой свободных блоков в открытой документации детально не описана. Известно, что эта карта использует не по одному биту на блок, как традиционные ФС, а несколько бит: различные биты соответствуют состояниям блока в различных снимках. При наивной реализации это потребовало бы хранения 256 бит на каждый логический блок (так что карта блоков должна была бы занимать 1/16 объема диска) и перестройки всей карты блоков при создании и удалении каждого снимка. Поведение реального WAFL, по-видимому, свидетельствует, что в нем используется какой-то более разумный механизм [www.netapp.com 3002].

(a) Перед снимком (b) После снимка (c) После обновления блоков

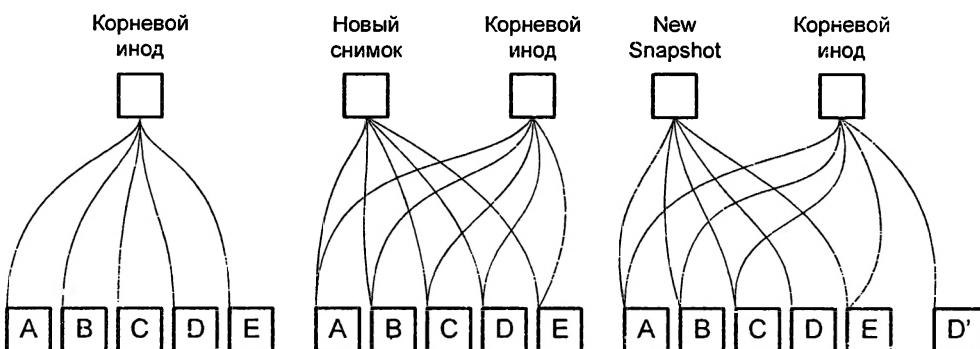


Рис. 11.29. Процесс создания снимка NetApp WAFL

Sun ZFS

Файловая система ZFS компании Sun была включена в вышедшую в начале 2005 года версию Solaris 10 [www.sun.com zfs]. Аббревиатура ZFS официально расшифровывается как Zettabyte File System; термин Zettabyte не является общепринятым, но пытается отразить тот факт, что архитектура ZFS использует 128-разрядные счетчики и указатели, так что объем файлов может достигать 2^{127} байт, а объем дисковых томов — 2^{128} логических секторов. Если емкость запоминающих устройств будет возрастать в соответствии с законом Мура (то есть удваиваться каждые полтора года), такие объемы могут быть достигнуты лишь более чем через столетие. Существуют даже оценки, из которых следует,

что запоминающие устройства такого объема вообще невозможны — в статье [www.sun.com zfs] цитируется фраза главного архитектора ZFS Джеффа Бонвика: "Нельзя заполнить 128-разрядное запоминающее устройство, не вскипятив океаны" (я затрудняюсь сказать, каким именно способом получена такая оценка и, в частности, о каких ограничениях на время заполнения устройства идет речь).

Некоторые менее официальные публикации интерпретируют название ZFS как "последнее слово в развитии файловых систем" — не в смысле "самое новое сказанное", а, скорее, в смысле "последнее логически возможное".

На время подготовки книги к печати открытой информации по ZFS было доступно не так уж много. Известно, что:

- файловая система основана на механизме копирования при записи или, что в данном случае то же самое, записи повсюду;
- поддерживает значительное или даже практически неограниченное (в некоторых публикациях фигурирует значение 2^{63}) количество одновременно хранимых снимков;
- транзакционную модификацию как мета-, так и пользовательских данных и динамическое включение и исключение новых физических дисков не только без переразметки, но даже без перемонтирования файловой системы.

Заявлены также более экзотические для современных ФС свойства:

- дополнительная защита данных с помощью 64-разрядных контрольных сумм;
- динамически изменяемый, в зависимости от типа нагрузки, размер логического блока;
- балансировка загрузки дисков. При работе с дисковыми массивами ZFS балансирует частоту и порядок записи на диски с учетом их наблюдаемой относительной производительности и текущих состояний очередей запросов к этим дискам;
- совместное использование дисков несколькими файловыми системами — как относящимися к одной копии Solaris, так и разным, например, в кластерах Sun Fire или в условиях, когда дисковый массив подключен одновременно к нескольким компьютерам с помощью SAN.

Утверждается, что несколько (или даже несколько тысяч) логически независимых ZFS могут использовать общий пул дисков или дисковых массивов как единый ресурс. Доля дискового пространства, приходящаяся на каждую из таких файловых систем, регулируется механизмом квот и может изменяться "на ходу". Механизмы управления списками свободных блоков в таких условиях в открытых публикациях, доступных ко времени выхода второго издания этой книги, не обсуждались. По неофициальным сообщениям сотрудников Sun, используются механизмы, обходящие патенты NetAPP; компания Sun подала заявки на четыре патента, защищающие отдельные аспекты работы ZFS. Исходные тексты ZFS опубликованы в рамках проекта OpenSolaris на условиях CDDL. На время выхода второго издания книги доступна версия ZFS для FreeBSD. Несовместимость CDDL и GPL не допускает включения ZFS в ядро Linux, но в настоящее время продемонстрирована работа ZFS под FUSE.

Вопросы для самопроверки

1. Что такое файл?
2. Чем файл отличается от записи базы данных, а файловая система — от СУБД? (Ответ на этот вопрос не следует из текста главы и требует самостоятельных размышлений.)
3. Почему сложно или даже невозможно добавить поддержку длинных имен файлов в ОС, которая поддерживала только короткие имена файлов? Не объясняет ли ответ на этот вопрос также и то, почему сложно изменять другие параметры имени файла, например набор допустимых символов?
4. Какие преимущества дают вложенные каталоги? Почему вложенные каталоги обычно реализуют в виде файлов специального типа, а не хранят в едином на всю ФС каталоге иерархические имена файлов? Какие из известных вам файловых систем используют второй подход?
5. Каковы преимущества и недостатки каталогов, отсортированных по имени файла, по сравнению с несортированными? Почему отсортированные каталоги применяются не везде?
6. Почему в этой главе утверждается, что для "нормальной" работы с файловой системой желательно позволить файлам занимать несмежные участки диска? Имеете ли вы опыт работы с ФС, на которых файлы обязаны занимать непрерывное пространство?
7. Для чего используются множественные корневые каталоги в CDFS?
8. Каковы недостатки файлов, реализованных в виде связных списков (каждый блок данных файла хранит указатель на следующий)? Почему структуру файловой системы FAT можно описать как вариант реализации файлов в виде связных списков?
9. Какие преимущества дает FAT по сравнению с хранением указателя на следующий блок в самом блоке?
10. Каковы преимущества и недостатки у логических блоков малого размера? У логических блоков большого размера? Почему нельзя неограниченно увеличивать размер логического блока?
11. Что такое субаллокация? Какие преимущества она дает? Как можно было бы реализовать субаллокацию в файловой системе типа FAT?
12. Почему ФС с субаллокацией должны поддерживать резерв свободного пространства? Из каких соображений определяется размер этого резерва? (Ответ на второй вопрос непосредственно не обсуждается в этой главе и требует самостоятельных исследований.)

13. Почему "сложные" файловые системы не хранят атрибуты файла и данные о его размещении по диску в каталоге, а практически всегда выносят их в отдельную запись? Почему так важно минимизировать размер каталога? Может ли это каким-то образом быть связано с тем, использует ФС отсортированные каталоги или несортированные?
14. Сравните различные стратегии размещения записей, описывающих файлы (инодов, ф-нодов и записей метафайла). Каковы преимущества каждого из этих методов и его недостатки?
15. Что такое жесткие связи? Почему они могут создаваться только в пределах одной файловой системы?
16. Почему ни одна из ФС для ОС семейства Unix не допускает произвольного создания жестких связей для каталогов?
17. Что такое символические связи? Каковы их преимущества и недостатки по сравнению с жесткими?
18. Почему NTFS никогда не уменьшает объем метафайла? Нельзя ли использовать это поведение для атаки отказа сервиса? Как эта проблема решается в других ФС с динамической таблицей инодов, например в JFS?
19. Почему большинство ФС для ОС семейства Unix никогда не уменьшают размер каталога? Нельзя ли использовать это поведение для атаки отказа сервиса? Всегда ли такая атака может быть предотвращена посредством дисковых квот?
20. К чему может привести нарушение целостности метаданных файловой системы? Как обеспечивается целостность метаданных в традиционных ФС?
21. Какие преимущества дают журнальные файловые системы по сравнению с традиционными? Какое из этих преимуществ является наиболее практически важным?
22. Почему большинство журнальных ФС включают журналирование только для метаданных, но не для пользовательских данных? Почему в этой главе утверждается, что автоматически выделять транзакции в операциях над пользовательскими данными невозможно?
23. Полностью ли соответствует истине утверждение, что журнальные ФС гарантируют целостность метаданных? Гарантируют ли они целостность пользовательских данных, и если да, то каким образом?
24. Опишите принцип работы журнала намерений. Как вы стали бы реализовывать журнал, допускающий одновременное выполнение нескольких транзакций? Является ли схема, применяемая в JFS, единственной возможной?

25. Почему журнал транзакций не приводит к снижению производительности ФС вдвое? Ведь при его использовании все метаданные пишутся на диск дважды. Почему журналирование может приводить даже к повышению наблюдаемой производительности, причем значительному, иногда на десятки процентов? При каких обстоятельствах это возможно?
26. Что такое моментальный снимок ФС? Как он реализуется? Для чего моментальные снимки применяются на практике? Как еще их можно использовать? Что ограничивает их применение?
27. Что такое "запись повсюду"? Какие требования она накладывает на структуру метаданных? Почему при "записи повсюду" каждая модификация данных не обязательно приводит к модификации всех метаданных?



ГЛАВА 12

Обработка ошибок и исключений

Веет с поля ударной волною,
Тихо падает свет голубой.
Ты признайся, что это такос,
Ты скажи, аспирант молодой.
Может, снова в расчетах ошибка,
Вместо суммы ты разность берешь...
Что ж ты опыты ставишь так близко.
Что ты девушкам спать не даешь.

Студенческий фольклор

В предыдущих главах мы неоднократно сталкивались с различными примерами исключительных ситуаций, которые могут возникнуть при работе программы. Часто, но далеко не всегда, эти ситуации порождаются ошибками в самой программе.

Вопросу обработки ошибок посвящена обширная теоретическая литература; внимательный читатель должен был также отметить, что весьма сложные и ресурсоемкие подсистемы современных вычислительных систем, такие как виртуальная память, были придуманы исключительно или главным образом для нужд обработки ошибок. При всем этом вполне удовлетворительных решений на сегодня не известно и разговоры о кризисе программирования, симптомами которого является неадекватная обработка ошибок, не прекращающиеся с 60-х годов XX века и имеют под собой серьезные и реальные основания.

Наиболее общее определение ошибки в программе звучит примерно так: это несоответствие поведения программы решаемой задаче и/или ожиданиям пользователя. Очевидно, что ни аппаратура компьютера, ни операционная система ничего не знают о решаемой задаче, поэтому в общем случае задача автоматического обнаружения ошибок неразрешима. Корень проблемы состоит, по-видимому, в том, что на средства обработки ошибок возложена

своеобразная задача. Именно эти средства пытаются состыковать два разных мира с разными и, вообще говоря, взаимоисключающими законами.

С одной стороны находится мир математики, идеальных задач с четкими и принимаемыми как данность непротиворечивыми формулировками. Для этих задач существуют точные решения, корректность которых доказана в соответствии с требованиями, которые предъявляются к доказательству теорем, либо столь же точно доказана их неразрешимость (впрочем, даже в этом мире есть обширные классы задач, неразрешимость которых является недоказанной гипотезой, например, знаменитая гипотеза о том, что NP-полные задачи не разрешимы за полиномиальное время).

С другой стороны находится реальный мир, в котором алгоритмы реализуются криво, оборудование работает не всегда, пользователь не понимает, чего хочет, а постановка задачи определяется в ходе многостороннего опосредованного политического торга между:

- наемным менеджером (представителем заказчика);
- профсоюзом пользователей;
- наиболее скандальными представителями самих пользователей;
- научным руководителем разработчика, которому нужно защищать диссертацию;
- многочисленными категориями других "заинтересованных лиц" (stakeholder), например, акционерами компании-заказчика; влияние каждой из этих категорий по отдельности слишком мало, чтобы упоминать ее отдельным пунктом списка, но самих категорий достаточно много, чтобы сумма ряда не сходилась, и общее влияние на результат оказывалось заметным.

"Обработку ошибок" можно описать как комплекс мероприятий, направленный на стыковку этих двух миров и устранение, обход или хотя бы скрытие возникающих на стыке противоречий.

Из такого описания очевидно, что без обработки ошибок никакая вычислительная система не сможет найти практического применения. Очевидно также, что задача обработки ошибок не может решаться "идеально" в математическом смысле; все предназначенные для этого механизмы в том или ином отношении формально неполны или некорректны и вводятся и оцениваются из pragматических соображений.

Как уже отмечалось, задача автоматического обнаружения ошибок в программах в общем случае неразрешима. Для некоторых важных ошибок — например, для "зацикливаний" программы, которые приводят к бесконечному времени исполнения, — даже доказана невозможность их автоматического обнаружения (знаменитая проблема финитности алгоритма, [Минский 1971]).

Примечание

Строго говоря, неразрешимость проблемы автоматического определения финитности (завершения за конечное время) алгоритма доказана лишь для бесконечных автоматов — машины Тьюринга и эквивалентных ей устройств с бесконечной памятью. Однако одно из доказательств справедливо и для конечных автоматов — при условии, что проверяемый автомат может исполнить любую программу проверяющего.

Однако существуют категории ошибок, которые, с одной стороны, приводят к невозможности нормального исполнения программы и, с другой стороны, могут обнаруживаться аппаратурой процессора или программным обеспечением, причем не только ОС, но и, например, стандартными библиотеками языка программирования или кодом, который вставляется компилятором.

Список этих ситуаций весьма обширен и включает в себя следующее:

- ошибки в самой программе:
 - деление на ноль;
 - другие математические ошибки, например попытка вычислить действительный квадратный корень из отрицательного числа;
 - выход индекса за границы массива (переполнение буфера). Некоторые компиляторы языков высокого уровня (например, Pascal и Visual C .Net) могут по запросу программиста вставлять такие проверки при каждом обращении к элементу массива. Разумеется, это приводит к потере производительности;
 - ошибки работы с указателями, например обращение к защищенной странице памяти в системах с виртуальной памятью;
 - обращение к невыровненному слову на процессорах, которые этого не допускают;
- физическая нехватка ресурсов (чаще всего, оперативной памяти) и превышение квот;
- отсутствие требуемых файлов в файловой системе или каких-либо других логических ресурсов, например, требуемого символа в разделяемой библиотеке или семафора;
- неисправности и особые состояния (например, отсутствие удаляемого носителя в накопителе или бумаги в принтере) периферийных устройств;
- отсутствие полномочий, необходимых для выполнения операции;
- изменение намерений пользователя. Многие интерактивные программы сами предоставляют средства обнаружения таких ситуаций (например, кнопка **Отменить** в диалоговом окне, в котором изображается индикатор состояния процесса), но многие ОС предоставляют также средства, устроенные аналогично рассматриваемым далее в этой главе средствам обра-

ботки исключительных ситуаций, которые позволяют пользователю "прибить" нежелательную задачу, а задаче — более или менее корректно обеспечить свое завершение или изменить поведение при исполнении пользователем такого запроса.

12.1. Типы ошибок, которые следует обрабатывать

Сколько нужно программистов, чтобы сменить перегоревшую лампочку?

Ни одного, это аппаратная проблема.

Народный юмор

В начале главы мы видели общий список проблем, которые — с той или иной степенью натяжки — можно считать ошибками. Сейчас мы попробуем ввести несколько более формальную их классификацию.

Ошибки аппаратуры. Эти ошибки могут быть обусловлены как дефектами изготовления или сборки оборудования (например, дефектный бит в ОЗУ или в одном из регистров контроллера устройства, или плохой контакт в разъёме), так и преходящими факторами — перегревом, электромагнитными помехами и т. д. В соответствии с эпиграфом данного раздела, в этой книге мы будем обсуждать эти проблемы лишь вкратце — в действительности, контроль качества изготовления оборудования, обеспечение надлежащих условий его эксплуатации, диагностика неисправностей и их устранение — огромная тема, которой посвящено множество различной литературы, от фундаментальных учебников до разделов в руководствах пользователя.

Ошибки программирования. Исчерпывающая их классификация, по-видимому, невозможна; в этой главе мы дадим короткий обзор наиболее важных ошибок с указанием, почему они могут быть опасны. В целом, борьба с этими ошибками — также огромная тема, которой посвящен ряд отдельных книг (например, [Уэллин 2004]). Значительная часть учебников по методологии проектирования программ и собственно программирования, такие как [Дейкстра 1978, Хоар 1989], явно или неявно посвящены теме борьбы с ошибками; в действительности, сама потребность в сложных и формализованных методологиях обусловлена именно стремлением избежать ошибок. Из-за высокой стоимости поиска и исправления многих типов ошибок, а также из-за того, что стоимость эксплуатации программного комплекса с неисправленными ошибками может оказаться совершено астрономической (например, если из-за ошибки в программе будет уничтожена база данных крупной компании), понятно желание вложить средства в их предотвращение. К сожалению, до сегодняшнего дня все эти вложения оказывались относительно малоэффективными, и тому есть ряд причин.

Некоторые технологии программирования — например, использование сборки мусора вместо явного освобождения объектов — действительно позволяют устраниить целые классы ошибок (так, сборка мусора позволяет устраниить утечки памяти и висячие ссылки), обычно ценой повышения стоимости исполнения программ. К сожалению, даже это обычно не приводит к повышению качества кода: внедрение таких технологий приводит к удешевлению разработки, а удешевление приводит к тому, что пользователи начинают заказывать больше программ с более сложной и/или более точно привязанной к их нуждам функциональностью. При этом общее количество кода увеличивается, а значит, увеличивается и количество ошибок.

При этом возможны и эффекты второго порядка. Повышение спроса на программистов приводит к тому, что в программисты идут менее квалифицированные и менее мотивированные люди, которые делают больше ошибок; уменьшение бюджета разработки приводит к сокращению тестирования и полному отказу от дорогостоящих приемов разработки, таких как независимый аудит кода и доказательства корректности. Поэтому, как ни парадоксально это может прозвучать, появление более совершенных технологий программирования и исключение некоторых классов ошибок в долгосрочной перспективе приводит не только к увеличению объемов кода, но и к снижению среднего качества этого кода. Это явление наблюдалось как в 60-е—70-е годы при распространении компилируемых ЯВУ, так и в 80-е—90-е годы при распространении интерпретируемых ЯВУ со сборкой мусора (Visual Basic, PHP, Perl, Java, C#).

Ошибки определения требований к программе и ошибки проектирования. Эти ошибки, пожалуй, наиболее опасны для программистов и организаций-распространителей программ, потому что программа, не соответствующая требованиям, которые к ней предъявляют пользователи, может привести к большим финансовым потерям и даже к выходу из бизнеса. Для пользователей они значительно менее опасны, потому что пользователи их относительно легко могут обнаруживать и отказываться от работы с такими программами — хотя это далеко не всегда просто и часто может пересекаться с еще одним кругом проблем, а именно — организационными проблемами в компании, которая заказывает программу. Эта задача частично пересекается с проблемой ошибок программирования и задачами методологий разработки ПО, но, так или иначе, обсуждение этого круга проблем слишком далеко ушло бы нас от темы книги. Поэтому в данной книге мы не будем обсуждать эту тему, несмотря на всю ее сложность и важность.

Ошибки пользователя. Строго говоря, ошибки этого класса связаны "серыми зонами", с одной стороны, с ошибками программирования, с другой — с ошибками определения требований и проектирования и с третьей — с организационными проблемами. Впрочем, достаточно важны и "чистые" поль-

вательские ошибки, когда пользователь нажимает не ту кнопку, пытается выполнить логически невозможную функцию (например, прочитать файл, который он сам же только что стер) или операцию, которая запрещена бизнес-логикой приложения (например, работник пытается сам себе изменить зарплату). Компьютерные профессионалы любят ехидничать по поводу пользователя, который "сам себе злобный буратино", но очевидно, что такие ошибки по разным причинам неизбежны.

В значительной мере предотвращение пользовательских ошибок — это задача эргономики пользовательского интерфейса. При этом разработчик интерфейса оказывается связан противоречивыми требованиями: с одной стороны, многие эргономические доктрины требуют, чтобы заведомо ошибочные действия были физически невозможны — например, это может достигаться блокировкой или скрытием тех или иных элементов интерфейса. С другой стороны, необходимо сделать программу как можно более гибкой и функциональной, т. е. предоставить пользователю как можно больше управляющих элементов. Проблема в том, что компромиссное решение в данном случае гораздо чаще сочетает недостатки крайних вариантов, чем их преимущества, так что многие "дружественные пользователю" интерфейсы, в действительности, скорее провоцируют пользователя на совершение ошибок, чем предотвращают их. Очевидно также, что определение того, какое действие является "заведомо ошибочным", тесно связано с определением требований к программе, т. е. мы попадаем в одну из упоминавшихся ранее "серых зон". Эргономика — также большая и важная тема, которой посвящено много самостоятельной литературы. Детальное ее обсуждение уело бы нас далеко от основной темы книги.

Так или иначе, значительная часть логики большинства интерактивных программ оказывается посвящена проверке корректности пользовательского ввода. Напротив, многие неинтерактивные программы могут взаимодействовать как непосредственно с пользователем (который набирает данные в текстовом редакторе или каком-либо нештатном интерфейсе, например присоединившись программой telnet к порту TCP, на котором "сидит" неинтерактивный сетевой сервис), так и с другой программой. В этих условиях обработка ошибок пользователя ничем не отличается от обработки ошибок программы. Как мы увидим далее в этой главе, многие из реальных стратегий обработки ошибок пригодны как для восстановления после программных ошибок, так и для коррекции ошибок пользователя.

В дополнение к приведенной ранее классификации, все исключительные ситуации можно разбить на два больших класса: *синхронные*, т. е. возникающие при исполнении (или, точнее, при попытке исполнения) определенной операции и, соответственно, связанные с определенным участком кода, и *асинхронные*, т. е. возникающие в результате какого-то внешнего по отношению к исполнению программы события.

В зависимости от семантики операций, одни и те же, на первый взгляд, ошибки, могут оказываться как синхронными, так и асинхронными. Так, если при выделении памяти происходит физическое резервирование ОЗУ и/или своп-пространства, то ошибка нехватки памяти будет синхронной, ведь она будет обнаруживаться в момент попытки выделения. Если же, как в современных системах семейства Unix, при выделении памяти только формируется виртуальное адресное пространство, а сама память выделяется при первом обращении, то нехватка памяти (или, точнее, своп-пространства) может обнаруживаться в самые неожиданные моменты и должна быть признана асинхронной исключительной ситуацией. Аналогично, при синхронной работе подсистемы ввода/вывода все ошибки периферийных устройств будут синхронными, а при использовании отложенной записи или опережающего чтения — асинхронными.

12.2. Стратегии обработки ошибок

Как-то ночью, проснувшись с сильного похмелья, Федор очень захотел пить. Не зажигая света, он вышел на кухню, нашупал на полке бутыль и начал пить. Сделав первый глоток, он понял, что ошибся, и в бутыли не вода, как он предполагал, а керосин.

Однако Федор с такой силой овладел дзен-буддизмом, что нашел в себе мужество не исправлять ошибки и спокойно допил бутыль до конца.

В. Шинкарев

В общем случае реакция на все перечисленные ситуации должна сводиться к минимизации вредных последствий. В идеальном случае программа должна обнаружить ошибку и каким-то образом обойти ее или принять меры к ее исправлению. Так, при нехватке памяти программа могла бы провести сборку мусора или изменить стратегию работы с памятью, например, уменьшив объемы каких-то внутренних кэшей или буферов. Если исправить ситуацию невозможно, программа должна корректно завершиться, освободив все занятые ресурсы и приведя в согласованное состояние все разделяемые данные и, особенно, все данные, хранящиеся в постоянной памяти (файлы, базы данных и т. д.). О многих ошибках — например, об отсутствии бумаги в принтере или о нехватке памяти — следует оповестить пользователя и/или оператора системы.

Если программа не в состоянии обработать ошибку самостоятельно, система "прибывает" задачу и неявно освобождает все занятые ею ресурсы, пытаясь минимизировать ущерб для системы в целом.

Разумеется, это сильно идеализированная схема, полностью реализовать которую на практике не получается. В частности, многие ошибки (это наиболее очевидно для арифметических ошибок, таких как деление на ноль) проще

предотвратить, чем восстановиться после них. Сам факт возникновения такой ошибки часто свидетельствует о том, что программист не предусмотрел соответствующую ситуацию и не знает, как адекватно реагировать на нее — или, во всяком случае, не заботился передачей этого знания программе.

После некоторых ошибок оказывается невозможно — или, точнее, нежелательно — восстанавливать состояние хранимых данных. Действительно, такие ошибки программирования, как ошибки работы с указателями или "ручками" оверлейных данных, или банальный срыв буфера (выход индекса за границы массива), приводят к записи значений в области памяти, занятые другими структурами данных, т. е., как правило, к разрушению этих структур данных. При динамическом управлении памятью обычно даже нельзя сказать, какие именно структуры данных были нарушены. В этой ситуации попытка перенести несогласованные данные из памяти в постоянное хранилище приведет к рассогласованию или, в худшем случае, к полному разрушению этого постоянного хранилища. Поэтому при ошибках работы с памятью обычно оказывается целесообразно полностью прекратить исполнение задачи; возможные при этом рассогласования в хранимых данных оказываются меньшим злом, чем то, что мы могли бы записать в эти структуры на основе разрушенных оперативных данных.

В частности, этот подход применяется и при ошибках в ядре ОС: обнаружив ошибку при работе с памятью в самом ядре, ОС немедленно останавливает работу и рисует на консоли регистры центрального процессора ("синий экран смерти" в Windows, kernel panic в системах семейства Unix), не пытаясь сбросить на диск пользовательские данные из дискового кэша и размонтировать файловые системы и, тем более, не пытаясь оповещать пользовательские программы об аварийном завершении. Восстановление файловых систем и пользовательских данных при этом производится теми же средствами, которые использовались бы при внезапном прекращении работы по другим причинам, например при выключении питания (механизмы восстановления файловых систем в таких ситуациях рассматриваются в разд. 11.4).

12.2.1. Автоматический перезапуск

— Парашюты у нас нигде не хранятся, потому что никаких парашютов не нужно.
— Это почему же? — озабоченно спросил Постренький.
— Потому что, если вы прыгнете с парашютом, он сейчас же запутается в лопастях пропеллера, и вас изрубит вместе с парашютом в куски. В случае аварии лучше прыгать вовсе без парашюта.

Н. Носов

В системах класса ДОС аварийный останов задачи — рискованная операция, которая часто приводит к разрушению данных; поскольку ДОС обладают

лишь весьмаrudиментарными средствами самоконтроля, обычно это заканчивается даже не "синим экраном смерти", а неконтролируемым распадом, который выглядит для пользователя как "зависание" системы: машина перестает реагировать на внешние раздражители и выводится из этого состояния только холодным перезапуском. В большинстве РС-совместимых компьютеров кнопка холодного перезапуска выведена на переднюю панель, да и в портативных компьютерах она является важной частью пользовательского интерфейса.

Обязательным элементом современных микроконтроллеров является *watchdog timer* (*сторожевой таймер*, *watchdog* дословно — "сторожевая собака"), часто работающий от собственного осциллятора, а не от общего тактового генератора машины. Программа микроконтроллера должна периодически сбрасывать сторожевой таймер, иначе, досчитав до конца, он делает вывод, что программа "зависла", и инициирует системный сброс.

Практика использования таких таймеров немного старше однокристальных микроконтроллеров — эти устройства появились в то же время, когда началось использование управляющих миникомпьютеров в достаточно ответственных промышленных и встраиваемых приложениях, т. е. в 60-е годы XX века.

Именно сторожевой таймер несколько раз перезагружал бортовой компьютер посадочного модуля "Аполлона-11" и, по-видимому, спас этим жизни астронавтов и лунную программу США [NASA 182505].

Эти устройства часто применяются во встраиваемых приложениях, особенно ориентированных на длительную автономную работу. Действительно, управляющий компьютер может оказаться в весьма неприятном для человека месте, например, вблизи от активной зоны реактора или ускорителя (понятно, что для таких применений необходимо специальное исполнение микросхем, например на сапфировой подложке). В этих случаях иногда выводят кнопку системного сброса в "чистые" области многометровым кабелем. Но, скажем, для управляющего компьютера космического аппарата такое решение просто нереализуемо. Бывают и ситуации, когда выводить кнопку системного сброса наружу нежелательно по более банальным, эргonomическим и тому подобным соображениям, например из-за опасности случайного нажатия.

О чём-то аналогичном такому сервису часто мечтают администраторы серверов. В новосибирском FIDO однажды вполне серьезно обсуждалась такая схема автоматизированного перезапуска: сервер каждые пять минут перепрограммирует источник бесперебойного питания на то, чтобы он через десять минут от текущего момента выключился и снова включился.

В FIDO встречаются описания экстравагантных решений, например "деглюкатор" (устройство, включаемое в шину ISA и по запросу программы выпол-

няющее сброс внешнего модема) или рычажный механизм, посредством которого компьютер, выдвинув поднос CD-ROM, может сам себе нажать на кнопку сброса (рис. 12.1). Этот механизм полезен в ситуациях, когда система еще условно работоспособна, но с высокой вероятностью может "зависнуть" при попытке нормальной перезагрузки. Такое устройство может быть полезно не только для ДОС, но и для коррекции ошибок в ядре защищенных ОС.

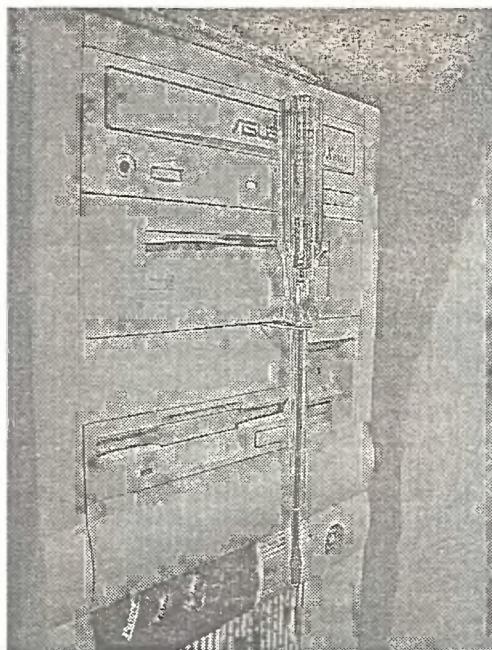


Рис. 12.1. Механизм для программной инициации аппаратного сброса.
При выдвижении подноса CD-привода отвертка нажимает на кнопку системного сброса.
Сборка и фото автора

Напротив, ОС специально проектируют так, чтобы задачи были в достаточной мере изолированы друг от друга, так что снятие одной задачи не должно повреждать остальные задачи и, тем более, ядро системы. На практике, однако, эту задачу не всегда удается полностью решить, даже используя мощные и дорогостоящие средства защиты, такие как виртуальная память.

12.2.2 Неявное освобождение ресурсов

Проблема состоит в том, что задачи, работающие под управлением ОС, все-таки не полностью изолированы друг от друга. Эти задачи нередко взаимодействуют между собой, разделяя те или иные ресурсы или пользуясь средствами межпроцессного взаимодействия. При аварийном снятии задачи необ-

ходимо корректно освободить все эти ресурсы, а средства межпроцессного взаимодействия привести в такое состояние, чтобы партнеры "убитой" задачи поняли, что дальнейшее взаимодействие невозможно.

Освобождение разделяемых ресурсов обычно достигается простыми средствами, напоминающими сборку мусора подсчетом ссылок. А именно, для каждого ресурса — например, для описателя открытого файла или сетевого соединения или для загруженной DLL — ведется счетчик количества задач, которые его используют. Когда задача, использовавшая этот ресурс, освобождает его — неважно, явным или неявным образом, или произошло ли неявное освобождение в ходе нормального или аварийного завершения задачи — счетчик задач уменьшается на единицу. Если счетчик при этом оказывается равен нулю, ресурс удаляется.

Если говорить конкретно о DLL, то многие реальные ОС пытаются кэшировать их загрузку, т. е. вместо выгрузки DLL сразу после освобождения они некоторое времядерживают ее в памяти, в расчете на то, что она может понадобиться вновь загружаемым программам. Кроме того, если допускаются циклические зависимости между DLL, то может возникнуть потребность в полноценной сборке мусора. Однако большинство других ресурсов не могут ссылаться друг на друга и не нуждаются в кэшировании, поэтому они действительно уничтожаются в момент освобождения их всеми процессами.

Однако для средств межпроцессного взаимодействия такой простой прием не годится. Рассмотрим простую и жизненную ситуацию: две задачи взаимодействуют через сегмент разделяемой памяти, защищенный мутексом (аналогичные проблемы возникают также при взаимодействии через файл, защищенный файловыми блокировками).

Если одна из задач аварийно завершается, удерживая мутекс, вторая задача может навечно зависнуть на мутексе. Соблазнительное решение этой проблемы — освобождать все мутексы, захваченные аварийно завершающейся задачей, но легко понять, что оно только усугубляет проблему.

Прежде чем мы перейдем к объяснению, почему это так, мы должны отметить менее значительный, но также неприятный момент. Для семафоров-счетчиков этот прием не годится, потому что семафор-счетчик не имеет прямой связи с "удерживающим" его процессом; строго говоря, для него вообще не определено "удерживающего" процесса (*см. разд. 7.3.1*). Тем не менее некоторые реализации семафоров предоставляют механизмы, позволяющие отменять операции над семафорами при завершении задачи.

Отмена операций над семафорами в System V IPC

При операциях над семафорами-счетчиками в System V можно указать флаг `SEM_UNDO`. При завершении процесса, совершившего такую операцию, произойдет ее отмена — т. е. если операция состояла в вычитании единицы из сема-

фора, то при завершении задачи к флаговой переменной семафора будет добавлена единица, и наоборот.

Поскольку операции добавления и вычитания значений из семафора-счетчика не обязаны делаться парами, система не может просто отменять последнюю операцию над семафором. Вместо этого, система вынуждена хранить всю историю операций процесса над семафором (или, точнее, только операций с флагом SEM_UNDO). История эта хранится в достаточно компактном виде — в форме целочисленного счетчика отмены (undo count), в котором накапливается сумма всех операций, осуществлявшихся процессом над семафором. При завершении процесса этот счетчик отмены вычитается из значения флаговой переменной. Документация по большинству систем, реализующих System V IPC, не очень внятно отвечает на вопрос, что будет, если в результате флаговая переменная окажется отрицательной. С определенной точки зрения было бы логично останавливать в этой ситуации завершение процесса, но понятно, что это совершенно недопустимо, а при аварийном завершении и просто нереализуемо. Так или иначе, понятно, что использовать этот флаг нужно с большой осторожностью; непродуманное его применение может привести к гораздо более серьезным проблемам, чем зависание на семафоре. Понятно также, что выполнение части операций над семафором с использованием этого флага, а части — без него, просто недопустимо.

Главная проблема отмены операций над мутексами и семафорами, впрочем, состоит в том, что если процесс удерживает мутекс, то это означает, что он работает с ресурсом, защищаемым этим мутексом. То есть если возник вопрос об отмене операции над мутексом, это, скорее всего, означает, что ресурс находится в несогласованном состоянии. Поэтому просто освобождать мутекс, занятый аварийно завершающейся задачей, было бы некорректно — необходимо оповестить процесс, который попытается его захватить, что предыдущая операция над этим мутексом (или, точнее, над связанным с ним ресурсом) была завершена нештатно.

Восстановление разделяемых ресурсов в Solaris Threads

В Solaris Thread Library и в реализации POSIX Thread Library для Solaris мутексы имеют атрибут, называемый *робастностью* (*robustness* — прочность). Это флаг, который может принимать два значения: PTHREAD_MUTEX_ROBUST_NP или PTHREAD_MUTEX_STALLED_NP (суффикс NP расшифровывается как Non Portable и указывает, что эти функции и значения не входят в стандарт POSIX).

В режиме PTHREAD_MUTEX_ROBUST_NP, если процесс, удерживавший мутекс, аварийно завершился, то мутекс остается заблокированным навсегда. В более мягком режиме PTHREAD_MUTEX_STALLED_NP такой мутекс остается в специальном несогласованном состоянии. Первая нить, которая попытается захватить такой мутекс, получит код ошибки EOWNERDEAD (владелец мертв), однако мутекс все-таки будет захвачен, так что нить сможет получить доступ к ресурсу. Эта нить должна попытаться привести ресурс в согласованное состояние.

Каким образом это может быть сделано, решающим образом зависит от структур данных, из которых состоит ресурс, и от операций, которые над этими данными производятся.

Если восстановление ресурса завершится удачно, нить должна вызвать функцию `pthread_mutex_consistent_np()`, которая приведет мутекс в нормальное состояние. Если же восстановление не удастся, нить просто освобождает мутекс. При этом все последующие попытки захвата мутекса будут приводить к ошибке `ENOTRECOVERABLE` ([ресурс] невосстановим).

Большинство реальных приложений, однако, решают эту проблему несколько проще. А именно, если несколько процессов работали с сегментом разделяемой памяти и один из них аварийно завершился, оставив захваченные мутексы, то остальные процессы тоже завершаются — не обязательно средствами аварийного завершения ОС, но, так или иначе, нештатно. То есть если приложение реализовано в виде нескольких задач ОС, взаимодействующих через разделяемую память, это, вообще говоря, не приводит к повышению надежности по сравнению с единственным процессом — аварийное завершение любого из процессов приводит к падению или, во всяком случае, к остановке приложения в целом. Это является дополнительным (к изложенным в разд. 7.4) аргументом против использования разделяемой памяти для и в пользу средств гармонического межпроцессного взаимодействия (труб, очередей сообщений и т. д.).

12.2.3. Оповещение пользователя

Кто мне писал на службу жалобы?
(Не помнишь, нет?)
Не ты?! Да я же их читал!

В. Высоцкий

С оповещением пользователя также не все просто. С одной стороны, слишком частые сообщения об ошибках раздражают пользователя, поэтому те ситуации, после которых программа способна восстановиться самостоятельно, следует обрабатывать именно так (впрочем, во многих случаях оказывается целесообразно все-таки записать сообщение об ошибке в лог-файл). С другой стороны, очень часто неудачные или частично удачные попытки восстановления приводят к цепочкам ошибок, возникающих в различных местах программы, так что когда программа все-таки "осознает" необходимость привлечь внимание пользователя, информация о том событии, с которого началась эта цепочка, в значительной мере уже утрачена. В результате оказывается невозможно сформулировать сообщение об ошибке так, чтобы пользователь понял, что же, собственно, произошло и какие действия ему следует предпринять.

Вообще говоря, такое развитие событий свидетельствует о неудачной или, во всяком случае, непродуманной стратегии обработки ошибок; проблема, впрочем, именно в том, что универсальной удачной стратегии такой обработки не существует, и выбор оптимальной стратегии для данного конкретного приложения является творческой задачей.

Знаменитые модальные диалоги Windows и Microsoft Office, дающие неисчерпаемые источники поводов для развлечения уже третьему поколению пользователей, возникли именно в результате неудачного подхода или, точнее, последовательности неудачных подходов к обработке ошибок.

Windows и MS Office первого поколения (так называемые "16-разрядные версии") отличались крайне низкой устойчивостью при работе. Среднее время работы любого из компонентов Office от запуска до аварийного завершения измерялось десятками минут, а при работе с большими документами — минутами и даже секундами. Приблизительно при каждом втором аварийном завершении Office уносил с собой всю систему, вынуждая пользователя делать холодный рестарт. Большая часть этих аварий была обусловлена ошибками в коде приложений, а большая часть ошибок — весьма своеобразной стратегией управления памятью, применявшейся в системах семейства Win16 (*см. разд. 4.4.1*). Без преувеличения можно сказать, что эти программы были совершенно непригодны для практического использования.

Основной причиной аварийного останова были ошибки доступа к памяти, обнаруживавшиеся подсистемой виртуальной памяти процессоров 80286 и 80386 как общая ошибка защиты (general protection fault). При обнаружении такой ошибки Windows выводила на экран модальный диалог с заголовком **General Protection Fault**, именем программного модуля, при исполнении которого произошла ошибка, и значениями регистров на момент возникновения ошибки. Если ошибка обнаруживалась в ядре системы, приблизительно та же информация выводилась на экран в текстовом режиме (так называемый BSOD, blue screen of death — "синий экран смерти").

Пользователей, разумеется, не устраивало такое положение дел, однако руководство Microsoft почему-то решило, что пользователей раздражают не сами ошибки, а сообщения о них; можно даже высказать некоторые гипотезы о том, почему так произошло, но все эти гипотезы описывают организационные, а не технические причины.

Кроме того, стандартная жалоба на "непонятные сообщения об ошибках" была проинтерпретирована как желание избавиться от технических терминов и информации в тексте сообщения, а не как желание получать такие сообщения, которые могли бы быть полезны для устранения причины ошибок. Впрочем, поскольку основная часть ошибок была порождена неудачной архитектурой системы, то совет по устранению ошибки должен был бы звучать приблизительно так: "Установите другую ОС и выберите замену для приложений".

В переходной ОС Windows 95 схема управления памятью была полностью изменена (*также см. разд. 4.4.1*); это привело к многократному увеличению наработки на отказ и, пожалуй, даже к переходу границы, которая отделяет

совершенно неприемлемые для эксплуатации системы от более или менее терпимых. К сожалению, разработчики Microsoft в своих попытках сделать систему "дружественной для пользователя" не ограничились этим, а попытались учесть жалобы пользователей или, точнее, свое понимание этих жалоб.

Все сообщения ОС и приложений были переработаны; так, сообщение про "general protection fault" с адресами и регистрами было заменено на знаменитое "приложение совершило недопустимую операцию и будет закрыто" без какой-либо дополнительной информации. Таким образом, ошибок стало меньше, но сообщения о них стали гораздо более непонятными; во многих случаях из текста сообщения оказалось удалена вся сколько-нибудь полезная информация. Некоторые ситуации — например, конфликты линий запроса прерывания между устройствами — часто вообще проходили без какой-либо видимой диагностики, просто целые подсистемы по-тихому отказывались работать.

На этот раз жалобы пользователей на "непонятные сообщения об ошибках" уже несомненно относились именно к непонятному тексту сообщений. При разработке систем следующих версий, Windows 2000 и Windows 98, разработчики решили, что текст можно сделать более понятным, удлиннив его. Полезной информации в сообщениях при этом не появилось (см. рис. 12.2), частично из-за возникшего на предыдущих этапах страха перед техническими терминами, частично из-за описанного в начале раздела сценария, когда в многокомпонентном приложении каждый из компонентов пытается, как может, восстановиться после ошибки, но восстановиться не может, и передает следующему компоненту частичную или даже искаженную информацию о том, что же, собственно, случилось.

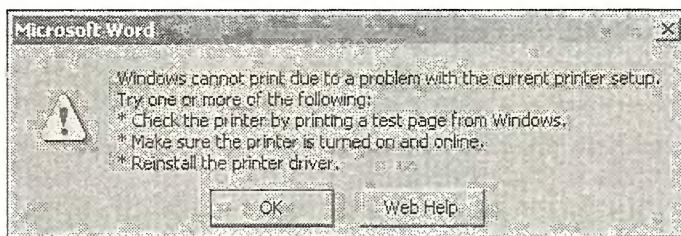


Рис. 12.2. Сообщение MS Word 2000, выдаваемое при попытке печати на недоступный сетевой принтер

Сообщения, таким образом, нисколько не стали понятнее. При этом был достигнут еще один побочный эффект; в ряде форумов я встречал конспирологические теории, что этот эффект в действительности и был основным ожидавшимся. А именно, выдавая длинные и преимущественно не относящиеся к

делу сообщения, разработчики Microsoft вообще отучили людей читать какие бы то ни было выдаваемые компьютером сообщения.

Особенно печальны последствия, к которым это привело в области интернет-безопасности. Действительно, Microsoft Internet Explorer честно предупреждает пользователя, что запуск управляющих элементов ActiveX, полученных из неизвестного источника, потенциально опасен. Но при этом тот же Explorer с такой же настойчивостью (и почти такими же диалогами) предупреждает, что опасно заполнять веб-формы, запускать Java-апплеты и делать множество других операций. Фактически, невозможно пользоваться Explorer, не натыкаясь постоянно на предупреждения такого рода. Все эти предупреждения выводятся модальными диалогами, занимают полэкрана, увешаны всеми предусмотренными CUA предупредительными значками, их тексты отличаются всего двумя или тремя словами (при том, что длина текста иногда измеряется десятками предложений), и единственный способ работать нормально — это в каждом диалоге такого типа отметить галочку **Больше не выводить это сообщение**. Таким образом, столкнувшись с предупреждением об опасности управляющих элементов ActiveX, средний пользователь на автомате нажимает кнопку "согласиться" и запускает управляющий элемент без каких-либо предосторожностей — и хорошо, если этот управляющий элемент окажется относительно безобидным adware, ведь таким образом распространяются и гораздо более опасные программы, например сетевые черви или спам-боты (см. гл. 13).

12.2.3. Логи

— Позвольте, товарищ, у меня все ходы записаны.
И. Ильф, Е. Петров

Гораздо больший успех имела другая стратегия оповещения об ошибках, направленная не столько на пользователя, сколько на оператора или администратора системы, а косвенно — и на разработчика приложения и/или системы.

При такой стратегии программа (прикладная или системная) просто пишет сообщения обо всем, заслуживающем внимания (не обязательно только об ошибках), в некоторое хранилище, называемое логом или *лог-файлом* (log, возможно от англ. "бревно"; по одной из легенд, это название пошло от практики лесорубов делать заметки и писать объявления на обтесанном бревне). Лог-файл может быть просто текстовым; некоторые системы, например Windows NT, пишут логи в специализированную базу данных.

Лог — это очень ценное, в ряде отношений незаменимое средство диагностики ошибок, потому что он позволяет восстановить последовательность событий, которая привела к ошибке. Особенно ценные логи серверных приложе-

ний, длительное время работающих без непосредственного участия оператора, или многокомпонентных приложений, имеющих серверные компоненты.

В ряде отношений очень привлекателен распределенный сбор логов, когда лог хранится на удаленной системе, а записи в него передаются через сеть. Это позволяет сохранить лог даже в случае полной потери данных на исследуемой системе; кроме того, удаленный лог защищен от подчистки и, таким образом, крайне ценен при анализе прорывов системы безопасности.

Большинство программных комплексов предоставляют возможность настраивать уровень лога, т. е. что именно следует писать в лог-файл. Уровни такой настройки обычно довольно грубы; часто их не более трех: "только ошибки", "ошибки и предупреждения", "полный лог". Некоторые сложные программные пакеты позволяют индивидуально настраивать уровень лога для разных компонентов.

Настройка уровня лога и выбор, что именно следует записывать в полный лог, что считать предупреждениями и т. д. — довольно сложная и творческая задача. Действительно, при низком уровне лога мы можем потерять информацию о событии, которое привело к ошибке — как из-за потери самого события, так и из-за того, что событие в логе было отмечено, но о нем не было сохранено достаточной информации. При высоком уровне лога мы можем получить огромный объем информации, в котором релевантные данные просто утонут.

Некоторую помощь в поиске релевантной информации при высоком уровне лога могут предоставить стандартные средства работы с текстом, такие как простой поиск и поиск по шаблонам, и специализированные пакеты, так называемые анализаторы логов (log analyzer). Такие пакеты обычно производят поиск в логе строк или записей, соответствующих определенным шаблонам — т. е. их работа, в конечном итоге, сводится к тому же поиску по шаблонам. Чтобы поиск по шаблонам был эффективен, записи в логе должны обладать определенной регулярностью; впрочем, поскольку логи пишет автомат, это не так уж сложно обеспечить.

Накопление данных в логе представляет и еще одну практическую проблему: поскольку данные только накапливаются, за конечное время они заполнят любое доступное запоминающее устройство. Поэтому все реальные системы в той или иной форме предполагают средства ротации или ретенции логов (retention), т. е., попросту говоря, стирания устаревших данных.. При выборе политики ротации нужно, разумеется, учитывать регламент обслуживания системы — необходимо хранить логи как минимум в течение интервала времени, за который администратор обязан хотя бы раз посмотреть, что происходит с системой. На практике обычно хранят логи за несколько таких интервалов.

Для некоторых приложений, например для системы управления бизнесом крупных финансовых предприятий, оказывается целесообразным полностью отказаться от ретенции логов при довольно высоком уровне детализации этих логов, обеспечивающих возможность проследить каждую проводку (т. е. перевод денег с одного счета на другой). Хранение таких логов требует специализированных программно-аппаратных решений; в современных условиях это механизированные библиотеки удалаемых носителей (магнитных лент, магнитооптических или записываемых оптических дисков).

С технической точки зрения запись в лог не представляет значительных сложностей. Системы семейства Unix предоставляют для работы с логами специальный режим работы с файлом, который может задаваться при открытии (флаг `O_APPEND` в параметре системного вызова `open`) или при работе с файлом (для этого необходимо использовать системный вызов `fcntl`). Append означает добавку, дополнение. При работе с файлом в этом режиме система позволяет программе передвигать указатель чтения/записи, но перед реальной записью этот указатель всегда устанавливается на конец файла. Благодаря этому гарантируется, что даже если несколько процессов пишут в один лог-файл, их записи не будут перекрывать друг друга, а порядок записей будет соответствовать хронологическому порядку исполнения соответствующих системных вызовов.

Если логи используются для анализа возможных атак на систему безопасности, большую опасность представляет их подчистка, т. е. удаление записей, свидетельствующих о несанкционированной деятельности взломщика. Некоторые системы предоставляют для предотвращения подчистки специальный уровень доступа, который можно описать как принудительное включение режима добавления. В системах семейства Unix такого режима нет — программа может указать флаг `APPEND` при работе с файлом, но нет способа принудить ее этот флаг указывать.

Чтобы решить проблему подчистки логов в таких условиях, в системах семейства Unix был реализован специальный сервис `syslog`. Этот сервис реализуется с помощью специализированного сервисного процесса, `syslog`-демона, который единственный имеет право записи в файлы логов. Все остальные задачи обращаются к демону через именованную трубу или сетевой сокет, и передают записи, которые считаются необходимым поместить в лог. Для каждой записи указывается, к какому сервису она относится, ее тип (ошибка, предупреждение, сообщение) и произвольная информация, которую счел нужным записать программист. Интерфейса для удаления ранее записанной информации не предусмотрено. Если общение с `syslog` происходит по сети, логи хранятся на удаленной машине и подчистить их физически невозможно, даже если взломщик получит максимальный уровень привилегий на взломанной системе.

Важным недостатком логов является то, что они хранят информацию о событии, но сами по себе не оповещают оператора системы, администратора и разработчика об ошибке. Чтобы узнать об ошибке, оператор должен заглянуть в лог и посмотреть, что произошло. В тех ситуациях, когда нужна оперативная реакция (обслуживание периферийного устройства, перезапуск серверного процесса и т. д.), это нежелательно. Довольно большую популярность получило автоматическое оповещение о таких событиях — по почте, с помощью пейджеров (как через шлюзы из электронной почты в пейджерную сеть, так и посредством специализированных протоколов), а в последние годы — с помощью SMS.

При использовании таких сервисов очень большую роль играет настройка фильтрации событий, о которых происходит оповещение: если оповещение может разбудить системного администратора посреди ночи, это должно быть оповещение о действительно важном событии.

В остальном реализация таких оповещений не представляет большой технической сложности. Нередко удается построить приемлемые системы такого рода, работающие посредством анализа штатных логов системы: периодически сканируя лог, сервисный процесс выбирает заслуживающие внимания события и генерирует оповещение. Такое сканирование может быть совмещено с периодической высылкой администратору дайджестов лога. Впрочем, этот подход страдает проблемами, характерными для всех схем обработки событий путем опроса: время оповещения о событии сопоставимо с периодом опроса, а слишком частый опрос порождает слишком высокую нагрузку на систему.

Ряд современных ОС и серверных прикладных пакетов реализует специальные механизмы "тревог" (алертов, от англ. alert). Обычно это системный сервис, который принимает сообщения о нештатных событиях, фильтрует их по тем или иным правилам и оповещает указанных в конфигурации абонентов выбранными способами.

Интересным вариантом этого подхода является автоматическая отправка оповещений об ошибке разработчику программы. Разумеется, это имеет смысл только для ошибок, несомненно являющихся ошибками программирования (деление на ноль, ошибки работы с указателями, внутренние проверки и т. д.). При поддержке малотиражных программ, разрабатываемых для собственных нужд или на заказ, такое оповещение является мощным и довольно эффективным средством повышения качества поддержки. При работе над системой поддержки бизнеса в компании Новософт мы встроили в систему обработчик ошибок, отправлявший письмо при каждой возникшей невосстановимой ошибке при работе серверных агентов и элементов пользователь-

ского интерфейса. В результате удалось достичь очень быстрой реакции на ошибки программирования и согласования данных в рабочей версии системы; некоторые ошибки исправляли быстрее, чем пользователь дозванивался до ответственных за поддержку.

При поддержке крупнотиражных программ, особенно "коробочных", условно бесплатных и свободно распространяемых — т. е. поставляемых без контракта на поддержку — такое оповещение менее эффективно и наталкивается на ряд специфических проблем.

Первой из проблем является то, что для диагностики и исправления многих ошибок программирования разработчик должен довольно много знать о контексте, в котором исполняется программа (версия программы, ОС, оборудование), а также о данных, над которыми программа работала в момент аварийного останова. Большая часть этой информации содержится в посмертном дампе памяти, но такие дампы велики по объему, и пересылать их после каждого аварийного останова было бы слишком расточительно — как для пользователей, так и для разработчиков. По-видимому, именно из этих соображений системы семейства Windows вплоть до Windows XP SP 2 штатными средствами не создавали посмертных дампов памяти для аварийно завершившихся программ.

Другая, возможно даже более важная проблема, состоит в том, что в момент ошибки пользователь мог работать над какой-либо конфиденциальной информацией. Данные о контексте исполнения программы в момент завершения, особенно полный дамп памяти, могут содержать часть этой информации. Если программа приобреталась вместе с контрактом на поддержку, в контракте так или иначе оговаривается ответственность разработчика за разглашение или нецелевое использование полученных таким образом сведений, а в сложных случаях — то, на получение какой именно информации может претендовать разработчик (ранее мы специально оговорили, что ведем речь о программах, поставляемых без контракта).

Поэтому попытки внедрения средств автоматического оповещения разработчика в крупнотиражные программы наталкиваются на более или менее организованное сопротивление пользователей, вплоть до судебных процессов и парламентских слушаний. Тем не менее такие средства появляются все чаще и чаще, особенно в тестовых ("бета") версиях программ, например в тестовых билдах браузеров Mozilla. Иногда средствами такого оповещения удается собрать действительно ценную информацию — например, встроив в Windows XP систему автоматического оповещения, разработчики Microsoft обнаружили, что 90% всех аварийных остановов Microsoft Office по общей ошибке защиты, происходящих у пользователей, обусловлены всего двумя ошибками в коде.

12.3. Обнаружение ошибок

Вот сстап опять повис на девяностом
Из процентов. Тут попробуй, не опухни
Эти баги раздражают меня, Постум,
Даёкс больше тех, что ползают по кухне.

Ю. Нестеренко

Все мероприятия, связанные с обработкой ошибок, невозможны, если программа не знает, что ошибка вообще произошла. При этом, разумеется, чем раньше мы обнаруживаем ошибку, тем лучше — как с точки зрения восстановления, так и с точки зрения диагностики и устранения причин ошибки перед перезапуском программы.

Способы обнаружения ошибок не отличаются разнообразием. Часть ошибок обнаруживается аппаратурой. Это, главным образом, два типа ошибок:

- неверные операнды команд, например ноль в качестве делителя в команде деления или отрицательное число в качестве параметра команды взятия логарифма (многие современные 32-разрядные процессоры имеют такие команды);
- недопустимые указатели.

В системах без виртуальной памяти "недопустимым указателем" является указатель за пределы физической памяти. Контроллер системной шины РС-совместимых компьютеров таких ошибок не генерирует, просто при обращениях к несуществующему ОЗУ или периферийным устройствам считаются слова, заполненные единицами, а попытки записи игнорируются.

Некоторые процессоры, такие как SPARC и PDP-11, считают недопустимыми нечетные адреса при обращении к словам (SPARC также считает недопустимым указатель на слово, не делящийся на 4). При попытке обращения к такому адресу контроллер системной шины генерирует прерывание, которое так и называется — *ошибка шины (bus error)*. У современных процессоров x86 такое поведение можно включить установкой флага в одном из управляющих регистров процессора.

В системах с виртуальной памятью диспетчер памяти может генерировать исключения по довольно обширному набору событий — при обращениях к несуществующим адресам (т. е. с селектором, превосходящим размер таблицы трансляции), при обращениях на запись к страницам, защищенным от записи и т. д. Как мы видели в главе 5, далеко не всегда такие обращения являются ошибками в строгом смысле этого слова; например, бит защиты от записи может использоваться для реализации Copy-on-Write механизмов, а бит защиты от чтения — для эмуляции clock-бита. Тем не менее эти механизмы позволяют также обнаруживать ошибки программирования.

Так, по стандарту языка ANSI C, нулевой указатель — т. е. указатель, полученный приведением целочисленного значения 0 к типу указатель, как, например, в конструкции `char *a=(char *)0;` — не должен соответствовать никакому допустимому объекту. В системах семейства Unix это достигается за счет того, что первые восемь мегабайт адресного пространства каждой задачи образуют защитную область. Все дескрипторы соответствующих страниц защищены как от чтения, так и от записи.

Константные значения, особенно строковые литералы, можно размещать в защищенной от записи памяти. Стандарт ANSI C требует, чтобы строковые литералы были, если это возможно, защищены от записи, но некоторые старые программы все-таки их модифицируют, поэтому многие компиляторы C/C++ имеют опцию, позволяющую регулировать размещение строковых литералов (например, у GNU C это регулируется параметром командной строки `-f writable-strings`).

В качестве ограничителя стеков практически все известные мне ОС используют "сторожевую область" — защищенную от чтения и записи страницу, расположенную в конце области памяти, выделенной под стек. Когда стек дорастает до этой страницы, происходит ошибка защиты.

Не следует путать переполнение стека, которое обычно является результатом бесконечной рекурсии, и срыв стека (переполнение размещенного в стеке буфера с разрушением записей активации процедур). Сторожевая область защищает от переполнения стека, но не от его срыва.

Физические ошибки запоминающих устройств и линий передачи данных могут обнаруживаться с помощью контрольных сумм; некоторые устройства — особенно запоминающие устройства со сменными носителями или адаптеры беспроводной связи — используют алгоритмы подсчета контрольных сумм, которые позволяют не только обнаруживать такие ошибки, но и восстанавливать данные после них. Так, на компакт-дисках данные защищены избыточным кодом Рида-Соломона, который позволяет обнаруживать и исправлять множественные ошибки.

Напротив, для снижения стоимости некоторые запоминающие устройства вообще не используют никаких контрольных сумм. Так, в старых PC-совместимых компьютерах оперативная память была защищена битом четности (1 бит четности на каждые 8 бит данных). В середине 90-х годов на рабочих станциях и даже на маломощных серверах от этого бита постепенно отказались — в разъемах модулей памяти соответствующие контакты сохранились, но сами модули его не используют.

Напротив, в серверных материнских платах и серверных модулях памяти вместо простых битов четности реализован код Хэмминга (так называемый ECC DRAM).

Поскольку современное динамическое ОЗУ довольно надежно, незащищенная память на практике оказывается относительно приемлемым решением. Впрочем, необходимо отметить, что отсутствие контроля данных затрудняет диагностику систем с дефектной памятью: при использовании четности контроллер системной шины генерировал бы ошибку, а при незащищенной памяти система просто тихо "глючит" в непредсказуемые моменты и с труднопредсказуемой симптоматикой.

Некоторые периферийные устройства могут генерировать специальные прерывания при ошибках, но в большинстве случаев ошибки периферийных устройств обнаруживаются по флагам в слове состояния устройства или по задержке прерывания, сигнализирующего о выполнении операции.

Один из важных типов ошибок, которые также иногда удается обнаруживать аппаратно (с помощью сторожевых таймеров), — это "зацикливания" программы. ОС с вытесняющей многозадачностью также могут обнаруживать такие ошибки, используя квоты процессорного времени.

Таким образом, видно, что возможности аппаратуры по обнаружению ошибок не так уж велики. Наиболее дорогостоящие (по стоимости устранения из реальных программ) ошибки, такие как ошибки соревнования в многопоточных программах, утечка памяти и висячие указатели, выход индекса за границу массива и др., аппаратура не может обнаружить или обнаруживает лишь по отдаленным последствиям. Например, если в результате разрушения разделяемой структуры данных в поле, которое никогда не должно быть равно 0, этот самый ноль все-таки был помещен, то аппаратура сама по себе этого не обнаружит. Процессор обнаружит ошибку деления на ноль при попытке использовать это поле в качестве делителя или ошибку защиты — при использовании в качестве указателя, но это произойдет только при обращении к разрушенной структуре, т. е. в совсем другом месте программы, зачастую в модуле, который не имел никакого отношения к причинам разрушения.

Более универсальным средством обнаружения ошибок является проверка утверждений. А именно, большинство алгоритмов так или иначе сохраняют некоторые соотношения между обрабатываемыми данными (так называемые инварианты) или исходят из некоторых предположений об этих данных — диапазонах, которые могут принимать скалярные значения, размерах массивов и т. д. Рекомендуемый во многих книгах "защитный" стиль программирования состоит в том, чтобы вставлять проверки этих инвариантов и допущений везде, где только можно.

Так, большинство компилируемых языков программирования не проверяют при обращениях к массивам нахождение индекса в допустимых границах; диспетчер памяти сам по себе не может обнаруживать такие ошибки, или, точнее, может, но только когда в результате выхода индекса за границы мы

выходим также и за границу сегмента (обычно это требует превышения границы на несколько тысяч или даже миллионов элементов). Однако некоторые компиляторы вставляют проверки границ индекса в генерируемый код, так что ошибка обнаруживается сразу в момент индексации. Разумеется, это снижает производительность генерируемого кода, поэтому практически всегда такое поведение контролируется ключами компилятора или директивами в исходном коде.

Ряд разработчиков, работающих на C++, рекомендуют при отладке программы использовать STL `vector` вместо массивов (`vector` — это класс-шаблон, который реализует массивы переменного размера с элементами практически произвольного типа; при обращениях к таким массивам проверка границ индекса все-таки происходит) и лишь при компиляции окончательной версии заменять их на массивы.

Разумеется, проверки требуют дополнительного кода и дополнительного времени на исполнение. Некоторые из проверок могут работать дольше основного алгоритма; так, дихотомический поиск в отсортированном массиве опирается на предположение, что массив отсортирован; поиск в массиве размера N работает за время $O(\log N)$, а проверка, отсортирован ли массив, — за время $O(N)$. Поэтому при практическом программировании количество таких проверок приходится ограничивать.

Как уже говорилось, очень распространенный прием — это собирать две разные версии программы, одну — для отладки и тестирования, с проверками, и потому медленную, и другую — рабочую, без проверок. При этом код проверок обходится директивами условной компиляции, а проверки, вставляемые компилятором, выключаются с помощью ключей или операторов `#pragma`.

В языке С в файле `assert.h` стандартной библиотеки определен макрос `assert`, который проверяет условие; в качестве условия может использоваться любое выражение языка С. При компиляции с ключом `-DDEBUG` компилятор генерирует проверку условия и, при его невыполнении, вызов некоторой функции, которая печатает в стандартный вывод диагностики строку `assertion failed`, номер строки и название модуля исходного текста, где была поставлена проверка. При компиляции без ключа `-DDEBUG` вызовы `assert` не генерируют никакого кода.

При интерактивной отладке почти того же эффекта можно достичь, используя условные точки останова (см. разд. 2.10.2), но некоторые отладчики накладывают ограничения на конструкции языка С, которые можно использовать в условиях этих точек.

Если в рабочей версии программы происходит ошибка, можно прогнать отладочную версию на тех же данных и получить детальную информацию о том, что и когда пошло неправильно. Организация такого прогона может

обеспечиваться по-разному — либо за счет того, что пользователь пересыпает разработчику свои данные, либо разработчик высыпает пользователю отладочную версию загрузочного модуля.

Однако, как уже говорилось, из-за высокой стоимости проверок во время исполнения, из коммерчески используемых программ их часто исключают или, во всяком случае, ограничивают их использование. При разработке на языках С/C++ при помощи директив условной компиляции можно собирать с одних исходных текстов две версии кода — одну отладочную, с включенными проверками, и другую — рабочую, в которой эти проверки выключены. Обычно более или менее полной проверке подвергается пользовательский ввод и, реже, хранимые в долгосрочной памяти данные, которые могли быть модифицированы другими программами или повреждены в ходе аварийного останова. В сетевых приложениях настоятельно рекомендуется подвергать полной проверке все, получаемое из сети, — ведь даже если штатно мы общаемся только с программами, генерирующими "правильные" данные, взломщик или просто ошибающийся пользователь может присоединиться к нам посредством нештатной программы.

В крупных программах с модульной структурой часто происходит разделение (а иногда и неконтролируемый распад) программного комплекса на "модули" или, скорее, "домены доверия", при передаче данных внутри которых проверок не происходит; проверки выполняются только при передаче параметров между такими доменами. При этом границы доменов могут не совпадать с другими схемами декомпозиции программы, например структуре классов или модулей исходного текста или даже структуре процессов в работающем программном комплексе. Так, драйвер файловой системы в ядре ОС обычно полностью доверяет внешнему модулю, который осуществляет починку ФС, и почти не осуществляет проверок данных файловой системы на внутреннее согласование (как мы видели в разд. 11.4, такая проверка требует анализа всей ФС или значительной ее части, и поэтому обычно стоит слишком дорого, чтобы ее можно было проводить при каждом обращении к файлу).

Нельзя сказать, что само по себе появление таких доменов — явление нежелательное, более того, часто оно целесообразно и даже неизбежно. Так, саму по себе операционную систему можно представить как домен, который не доверяет пользовательским программам, и благодаря этому может защищать программы друг от друга и, в определенных пределах, от самих себя. Но неконтролируемый распад программы на такие домены представляет большую опасность. Действительно, если архитектор системы или сами разработчики не представляют себе границ доменов доверия, вполне возможна ситуация, когда границы между доменами не сплошные. При некоторых междоменных взаимодействиях проверка параметров происходит, а при других — не про-

исходит. Поэтому с одной точки зрения домены существуют (во всяком случае, тратятся ресурсы на их поддержание), а с другой точки зрения их не существует (ошибочные данные могут проникнуть через междоменную границу), поэтому при реальном поиске ошибки все равно придется тестировать и отлаживать весь программный комплекс.

Еще одной важной проблемой, ограничивающей применимость проверок во время исполнения, является тот факт, что они должны вставляться программистом. Типичная ошибка программирования состоит в том, что программист неявно делает некоторое допущение, которое в действительности не соблюдается. Слово "неявно" здесь ключевое — при написании ошибочной программы программист обычно не осознает этого допущения, и поэтому не только пишет код, полагающийся на это допущение, но и не заботится о том, чтобы вставить проверку, — неважно, будет ли этой проверкой отладочный `assert` или проверка, осуществляемая какими-то другими средствами.

Автоматическая генерация проверяющего кода малоэффективна для решения этой проблемы. Действительно, типичные автоматически генерируемые проверки, такие как проверка на попадание индекса в границы массива, страдают тем же недостатком, что и аппаратно обнаруживаемые ошибки — они обычно обнаруживают лишь симптом ошибки (вычислен неверный индекс), в то время как сама ошибка была сделана совсем в другом месте кода. Поэтому автоматически порождаемый проверочный код обычно не пытается восстановить, что же имел в виду программист, а просто прекращает исполнение программы с более или менее детальной диагностикой.

С этой точки зрения, идеальной границей между упоминавшимися ранее "доменами доверия" являются границы между зонами ответственности разных разработчиков. Разумеется, такое проведение границ доверия часто оказывается неоптимально с точки зрения производительности, а на практике может приводить и к чисто личностным конфликтам (проверка данных, передаваемых модулем, может быть воспринята как недоверие к квалификации программиста, который писал этот модуль).

12.4. Передача информации об ошибках

Абрадокс: Соседство с галактикой Кин-дза-дза — наша беда. Они сидят на нас счастьем...

Деконт: Порочными!

Абрадокс: А продолжение жизни в виде растений для них благо...

Деконт: И для всех!

Г. Данелия. Р. Габриадзе, к. ф. "Кин-дза-дза"

Обычно модуль программы, обнаруживший ошибку, не имеет достаточной информации для того, чтобы восстановиться после нее, поэтому возникает

потребность в передаче информации об ошибках каким-то другим модулям, обычно, но не всегда, вверх по цепочке вызовов. Несмотря на то, что эта потребность очень насущна, а известные методы ее удовлетворения, мягко говоря, не вполне адекватны, способов решения этой задачи не так уж много.

Все применяемые на практике методы передачи информации об ошибке попадают в одну из следующих категорий (с тем уточнением, что есть методы, которые можно описать как гибридные):

- коды ошибок;
- простые обработчики (хуки, callback, сигналы в системах семейства Unix);
- стековые и структурные обработчики ошибок;
- оператор On Error Goto (используются в PL/I, REXX, диалектах языка Basic);
- исключения языков высокого уровня (наиболее известны исключения в Java/C# и C++).

Как уже отмечалось, все эти средства более или менее неадекватны задаче. Одна из важных проблем, общая для всех этих методов, может быть описана следующим образом. Система обработки ошибок может предоставлять "закрытый" список возможных ошибок или "открытый". Оба варианта по своему плохи, а компромисс между ними логически невозможен. Единственное решение, которое приближается к такому компромиссу, которое мы рассмотрим далее в этом разделе, обладает собственными недостатками.

Если список ошибок "закрытый", то в интерфейсе модуля описано, какие именно ошибки он может генерировать. Добавление новых ошибок приводит к несовместимому интерфейсу. Логика, стоящая за закрытым списком ошибок, вполне очевидна: если модуль, пользующийся нашим интерфейсом, получает информацию о неизвестной ошибке, то он, разумеется, не в состоянии обработать ее и восстановиться после нее.

На практике, однако, решение сделать список ошибок закрытым приводит к неожиданным и неприятным последствиям. Рассмотрим простую и практическую ситуацию: мы разработали файловый API для простой однопроцессной дисковой операционной системы, который не предусматривает никаких средств защиты данных и разделения доступа (в ДОС эффективные средства разделения доступа все равно невозможны). Затем мы расширяем нашу ДОС сетевой файловой системой. Даже если ДОС остается однозадачной и незащищенной, сетевой файловый сервер должен иметь средства синхронизации доступа к файлам и разделения доступа между пользователями. Соответственно, в файловом API должны появиться коды ошибок, сообщающие о блокировке доступа или недостаточных привилегиях. Но, по условию, список ошибок у нас закрытый!

Аналогичные проблемы возникнут при попытке обеспечить работу программ для нашей ДОС в многозадачной и многопользовательской среде.

При разработке MS DOS эта проблема была решена довольно искусственным образом: ДОС, оснащенная многозадачной надстройкой или сетевым редиректором, кроме стандартного кода ошибки возвращает еще один, расширенный код ошибки. Старые приложения анализируют только стандартный код ошибки, а новые приложения, "знающие" о новой функциональности, могут запросить расширенный код ошибки и проанализировать также и его.

Впрочем, последовательное применение этого подхода потребовало бы вводить новый расширенный код ошибки и собственный интерфейс для его получения при каждой ревизии интерфейса модуля. При этом количество различных интерфейсов для получения информации об ошибках должно быть равно "старшему" номеру версии системы, т. е. для систем с достаточно длинной историей это количество быстро станет неприемлемо большим.

Таким образом, мы получаем сильный аргумент в пользу введения открытого списка ошибок, т. е. решения, что новая версия модуля может возвращать новые типы ошибок. Впрочем, мы уже видели сильный аргумент против этого решения: программа, которая ничего не знает о новом типе ошибки, не сможет не только обработать такую ошибку, но даже не сможет сообщить о ней пользователю!

Одно из решений последней части проблемы состоит в том, чтобы вместе с сообщением об ошибке передавать не только машинно-интерпретируемую информацию (код ошибки и ее параметры), но и текстовое сообщение, которое можно показать пользователю. Этот подход в ряде отношений очень привлекателен; в частности, он не только позволяет реализовать открытый список ошибок, но и позволяет избежать текстовых сообщений, подобных приведенному на рис. 12.2.

Действительно, наиболее вероятный сценарий возникновения приведенного на этом рисунке сообщения таков: подсистема печати "знает", что печатать невозможно из-за отсутствия сетевого соединения. Однако, из-за закрытого списка возможных ошибок, она не может сообщить об этом приложению, она может сказать только, что с принтером все плохо. Приложение видит, что с принтером все плохо, но не знает, что именно, поэтому оно выдает список наиболее вероятных причин, по которым все может быть настолько плохо. То, что настоящая причина в этот список не попадает, — неизбежная издержка избранного метода оповещения об ошибках.

Напротив, если бы подсистема печати генерировала текстовое сообщение об ошибке вместе с машинно-читаемым, она вполне могла бы передать текстовое сообщение приблизительно такого вида: "принтер не работает потому, что сервер печати не доступен" или даже, объединив свое сообщение с тек-

стовым сообщением сетевой подсистемы, она могла бы сообщить, что "принтер не работает потому, что нет маршрута к узлу 192.168.1.1". Посмотрев на настройки принтера, пользователь мог бы быстро выяснить, что именно на этом узле и расположен принтер, на котором он пытался печатать. Если нас не пугают длинные сообщения об ошибках, можно было бы даже сформировать сообщение "Word не может печатать, потому что система печати говорит, что принтер расположен на узле 192.168.1.1, и нет маршрута до узла 192.168.1.1". Впрочем, генерация синтаксически правильных предложений такого рода — сложная задача, и реальное сообщение выглядело бы примерно так:

"Word: ошибка печати; принтер \\192.168.1.1\hplj недоступен; нет маршрута к узлу 192.168.1.1".

Не очень хорошо с точки зрения канонов литературного русского языка, но вполне понятно и содержит всю необходимую информацию для диагностики и, возможно, даже для устранения проблемы.

Необходимо также отметить, что централизованная генерация текстовых сообщений об ошибках значительно повышает согласованность интерфейса: все приложения, пользующиеся подсистемой печати ОС, будут выдавать однотипные сообщения при одинаковых ошибках. Также это значительно упрощает локализацию приложений: не надо переводить сообщения об ошибках или, во всяком случае, достаточно перевести только префикс, который приложение подставляет перед системным сообщением.

Впрочем, надо отметить, что запуск приложений, локализованных на "неродном языке" — например, запуск англоязычного приложения на русскоязычной ОС — может привести к странным последствиям в виде многоязычных сообщений (часть по-русски, часть по-английски). Но можно предположить, что пользователь, который ставит такую конфигурацию, в каких-то пределах знает оба языка и вообще знает, что делает, и многоязыковые сообщения не должны оказаться для него неожиданностью.

При всей привлекательности такого подхода, необходимо отметить, что он все-таки ограниченно применим. В частности, для того чтобы использовать его в приложении с многоуровневой архитектурой, необходимо, чтобы все межуровневые интерфейсы были устроены в определенном смысле однотипно; во всяком случае, необходимо, чтобы все эти интерфейсы использовали открытые списки ошибок и могли передавать текстовое сообщение (причем сообщение потенциально неограниченной длины!) вместе с ошибкой — иначе вся конструкция теряет смысл.

Как мы увидим из рассмотрения реальных механизмов оповещения об ошибках, это весьма нетривиальное условие, которое сложно реализовать, особенно если мы пытаемся сделать межуровневые интерфейсы независимыми от платформы и/или языка.

12.4.1. Коды ошибок

Это хуже, чем преступление. Это ошибка.
Иш. М. Талейран

Коды ошибок — это простейший механизм оповещения об ошибках. Он состоит в том, что, кроме обычного значения, вызванная процедура или функция (или даже системный вызов) возвращают также и код ошибки.

Код ошибки обычно представляет собой числовое значение; разные ошибки соответствуют разным кодам. Сам по себе данный подход допускает как открытые, так и закрытые списки типов ошибок.

Способы разделения основного значения и кода ошибки отличаются большим разнообразием.

Коды ошибок в системах семейства Unix

Так, в системах семейства Unix, системные вызовы синтаксически оформлены как функции, возвращающие значение. Часть этих функций возвращает какое-то содержательное значение, обычно скалярное или, реже, указатель. При этом недопустимое значение скаляра (для целых чисел обычно `-1`) или нулевой указатель сигнализируют о том, что вызов завершился неудачей.

Системные вызовы, не возвращающие скалярного значения, обычно все равно возвращают целое число: `0` при успехе, `-1` при неудаче.

Видно, что сам по себе код возврата системного вызова не содержит кода ошибки. Этот код передается в глобальной целочисленной переменной `errno`; все документированные значения описаны в виде символьных констант языка C (в действительности, в виде препроцессорных макроопределений) в заголовочном файле `errno.h`.

При этом используется любопытное соглашение, что переменная `errno` имеет смысл, только если предыдущий системный вызов завершился неудачей (вернул код неуспешного завершения). Таким образом разработчики ОС резервируют за собой право устанавливать `errno` даже при успешном завершении системного вызова, хотя я должен признать, что не понимаю, для чего это право могло бы быть полезно.

Список кодов ошибок открытый; хотя для большинства системных вызовов описано, какие коды ошибок он возвращает, но не гарантируется, что в следующих версиях ОС этот список не будет расширен, а на практике некоторые вызовы могут возвращать недокументированные коды. С каждым кодом ошибки связано текстовое сообщение, которое может быть получено из глобального массива `char * sys_errlist[]`, при этом значение переменной `errno` просто используется как индекс в этом массиве. Современные версии систем подставляют разные массивы в зависимости от значения переменных среды `LANG` и `LC_MESSAGES`, которые указывают, на каком языке следует выдавать сообщения.

Вместо прямых обращений к этому массиву, стандарт POSIX рекомендует использовать функцию `char *strerror(int errnum);`. Сами тексты сообщений, разумеется, не стандартизованы, но обычно довольно информативны.

На первый взгляд, данный механизм обработки ошибок представляется идеальной реализацией схемы открытого списка ошибок, описанной в разд. 12.3: приложение может не знать, какие коды ошибок ему могут вернуть, но всегда может получить текстовое сообщение для каждого из возможных кодов ошибок. Если мы согласны на перекомпиляцию нашего приложения, то так оно и есть. Однако если мы по какой-либо причине хотим избежать перекомпиляции, вовсе не все так радужно: массив `sys_errlist` — это переменная, определенная в библиотеке `libc`, а не часть ядра системы. Поэтому расширение списка кодов ошибок, выдаваемых ядром, требует коррекции `libc` и пересборки статически собранных программ (в динамически собираемых программах можно обойтись заменой `libc.so`). Поэтому же в стандарте POSIX список ошибок открытый, но в описаниях бинарных интерфейсов, таких как iBCS (intel Binary Compatibility Standard — стандарт двоичной совместимости на процессорах Intel), список кодов ошибок закрытый.

Еще одно неприятное свойство Unix-подобной схемы оповещения об ошибках — это заложенная в ней нереентерабельность. Действительно, если два потока в рамках одного процесса вызывают систему и затем проверяют `errno`, есть риск, что один из потоков увидит код ошибки, предназначенный для другого потока. Чтобы избежать этого, многопоточные библиотеки, такие как POSIX threads library, вводят макроопределение, подменяющее во время компиляции обращения к переменной `errno` на вызов функции, которая, в свою очередь, извлекает код ошибки для текущего потока.

В ряде работ систему кодов ошибок Unix также критикуют за отсутствие классификации ошибок. Действительно, при использовании открытого списка ошибок часто хотелось бы получать некоторую дополнительную информацию о положении дел: например, уровень серьезности ошибки (что бы это ни означало), следует ли писать в лог сообщение о ней и нужно ли оповещать оператора системы (например, вряд ли следует оповещать оператора об отсутствии файла на диске, но почти наверняка его нужно оповестить об отсутствии бумаги в принтере), можно ли попытаться обойти эту ошибку простым повторением операции и т. д. Если эта информация каким-то образом закодирована в коде ошибки (возможно, но не обязательно, в определенных битах этого кода — например так, чтобы все коды фатальных ошибок были нечетными, а допускающих восстановление при второй попытке — четными), программа может в определенных пределах выбирать стратегию реагирования, ничего не зная о том, что же, собственно, произошло.

Вряд ли этот недостаток сам по себе можно счесть серьезным, ведь на практике адекватная реакция на нештатную ситуацию без понимания этой ситуации в лучшем случае маловероятна. Впрочем, отдельныеrudimentарные формы такой классификации в Unix присутствуют — так, существует код ошибки `EAGAIN`, который возвращается многими системными вызовами в целом ряде ситуаций, на первый взгляд совершенно разнородных. Все эти ситуации объединяются лишь одним свойством: повтор системного вызова с теми же самыми параметрами не обязательно приведет к повторению ошибки.

Главным достоинством кодов ошибок является концептуальная простота этого метода. Впрочем, его недостатки многочисленны и современные теоретики

программирования, да и многие практикующие программисты, относятся к этому методу без уважения.

Перечислим недостатки кодов ошибок.

- Неудобство использования или, точнее, громоздкий код, получающийся при работе с кодами ошибок. Действительно, каждый вызов процедуры или функции превращается как минимум в три строки ЯВУ: сам вызов, условный оператор проверки кода, распечатка сообщения об ошибке и/или генерация ошибки для вызывающей процедуры. Если код пытается обработать некоторые из ошибок, объем кода еще увеличивается. Это затрудняет понимание и отладку программ, при просмотре кода становится сложно понять его структуру.
- Далеко не во всех ситуациях использование кодов ошибок удобно и желательно. Сомневающимся в этом предлагается представить себе программу на ЯВУ, в которой после каждой операции деления, разыменования указателя или обращения к массиву необходимо было бы проверить код ошибки.
- В некоторых ситуациях коды ошибок вообще невозможно использовать. Так, в объектно-ориентированных языках конструкторы (в том числе и копи-конструкторы в C++) и деструкторы (или финализаторы) объектов вообще не имеют ничего похожего на возвращаемое значение. Единственный аналог кода ошибки, который можно было бы использовать для оповещения об ошибке при исполнении конструктора — это какой-либо флаговый атрибут у объекта. Это решение обладает столь многочисленными недостатками, что проще признать его абсолютно неприемлемым. Деструкторы же объектов вызываются неявно, а в языках со сборкой мусора (таких как Java и C#) — и в непредсказуемые моменты времени, поэтому для них этот прием не только неприемлем, но и физически не реализуем.
- Коды ошибок пригодны только для оповещения о синхронно возникающих ошибках, т. е. об ошибках, возникающих и обнаруживаемых в момент выполнения определенной операции. Асинхронно возникающие ошибки могут возникать в моменты, слабо связанные с порядком исполнения кода, и ассоциировать с этими событиями проверку кода ошибки невозможно — разумеется, если мы не хотим, чтобы вся работа программы состояла в проверке этого кода в холостом цикле.

В силу перечисленных, а также некоторых менее существенных причин, использовать коды ошибок в наше время стало не модно, этот механизм считается архаичным и во вновь разрабатываемых программных интерфейсах встречается все реже и реже.

12.4.2. Простые обработчики ошибок

Я слышу крик в темноте,
Возможно, это сигнал.
В. Бутусов

К этой группе относится ряд механизмов, не имеющих общепринятого родового названия. Названия самих этих механизмов отличаются большим разнообразием — обработчики (*handler*), хуки (*hook*, от англ. "крюк"), *callback*, сигналы (в системах семейства Unix) и т. д.

Сам механизм довольно прост и состоит в том, что при возникновении ошибки вызывается определенная процедура. Разумеется, предварительно программа должна зарегистрировать эту процедуру как обработчик для ошибок этого типа. Обычно, если никакой процедуры явным образом зарегистрировано не было, то вызывается системная процедура по умолчанию, которая нередко состоит в принудительном завершении процесса.

Таким способом обрабатываются практически все ошибки, обнаруживаемые аппаратурой — центральным процессором и контроллером системной шины: деления на ноль, недопустимые коды команд, обращения к невыровненным словам, ошибки четности ОЗУ. Такие ошибки приводят к возникновению аппаратных исключений, которые аналогичны аппаратным прерываниям (см. разд. 6.4). Обработчики аппаратных исключений регистрирует система; во многих ОС системный обработчик может затем вызвать обработчик, зарегистрированный пользовательской программой, но, как правило, пользовательским программам запрещено напрямую работать с таблицей векторов прерываний и самостоятельно обрабатывать аппаратные исключения.

Сигналы в системах семейства Unix

В системах семейства Unix этот механизм называется сигналами (*signal*). Сигналы делятся на два класса:

- синхронные (т. е. возникающие в момент исполнения программой определенной операции, например деления на ноль);
- асинхронные (т. е. возникающие из-за внешних по отношению к программе событий).

Примерами синхронных сигналов являются:

- уже упоминавшееся деление на ноль и другие арифметические ошибки — SIGFPE (SIGnal Floating Point Exception — исключение при работе с плавающей точкой), хотя он генерируется и при целочисленном делении на ноль;
- ошибка доступа к памяти — SIGSEGV (SIGnal Segmentation Violation — нарушение сегментации);
- ошибка шины — SIGBUS, обращение к невыровненным словам на процессорах, которые этого не допускают;

- разрыв трубы — SIGPIPE, попытка записи в трубу или сокет, противоположный конец которых закрыт;
- чтение с терминала — SIGTTIN, генерируется, если фоновый процесс пытается читать данные с терминала.

Примерами асинхронных сигналов являются:

- прерывание пользователем — SIGINT (SIGnal INTerrupt), генерируется при нажатии пользователем комбинации клавиш <Ctrl>+<C> (ASCII ETX, символ прерывания передачи); в старых системах этот сигнал генерировался при получении символа ASCII DEL;
- завершение терминальной сессии — SIGHUP (SIGnal HangUP), при подключении терминала по модему генерируется при потере модемом несущей и/или разрыве телефонной линии, т. е. когда "повесили трубку";
- будильник — SIGALRM, генерируется в определенный момент времени по таймеру, устанавливаемому системным вызовом `alarm`;
- потеря питания — SIGPWR, в компьютерах общего назначения может рассыпаться демоном, наблюдающим за состоянием источника бесперебойного питания; на ноутбуках — при приближении батареи к разряду;
- исчерпание квоты времени центрального процессора — SIGXCPU.

Сигналы могут также генерироваться искусственно; для этого служит системный вызов `kill` (в действительности, это семейство системных вызовов с различными механизмами выбора процессов, которым будет послан сигнал) и однократная команда `shell`.

Процесс может зарегистрировать для каждого из сигналов функцию-обработчик. Эта функция вызывается при возникновении события, соответствующего сигналу. Точнее, при возникновении события ядро системы просто устанавливает флаг в дескрипторе процесса. При каждой передаче управления процессу — при возврате из системных вызовов, при возврате из обработчиков исключений и прерываний и при возврате управления вытесненному процессу — планировщик проверяет, есть ли у процесса необработанные сигналы. Если они есть, то сначала вызываются обработчики и только потом — если эти обработчики не завершили процесс — управление передается коду процесса.

В старых версиях Unix флаги были реализованы в виде битовой маски, каждый бит которой соответствовал номеру допустимого сигнала. Таким образом, если до обработки сигнала произошло второе аналогичное событие, то информация о нем терялась. Современные системы поддерживают "надежные" сигналы, которые доставляются столько раз, сколько раз произошло событие.

Если в момент возникновения сигнала процесс исполнял блокирующийся системный вызов, этот системный вызов прерывается. Таким образом, если во время возникновения сигнала исполнялся системный вызов, то управление будет передано в его точку возврата; при этом код возврата будет сигнализировать об ошибке, а значение `errno` будет равно `EINTR`. Если разработчик программы все-таки желает исполнить системный вызов, он должен проверить код ошибки и, если он равен `EINTR`, повторить вызов с теми же параметрами.

Современные версии системы, такие как BSD 4.3 и старше, System V Release 4, при установке обработчика сигнала позволяют задать флаг, требующий автоматического перезапуска прерванных этим сигналом системных вызовов.

Вместо обработчика можно также зарегистрировать "функции" SIG_DFL и SIG_IGN. Эти "функции" представляют собой численные значения, приведенные к типу указателя на функцию. SIG_IGN (от IGNore) обозначает игнорирование сигнала; при возникновении соответствующего события ничего не происходит, никакие обработчики не вызываются, системные вызовы не прерываются, информации о пришедшем сигнале не сохраняется. SIG_DFL (DeFaUlt, по умолчанию) соответствует системному обработчику по умолчанию; для разных сигналов эти обработчики, вообще говоря, различны, но их общая номенклатура не так уж широка.

Для большинства сигналов обработка по умолчанию состоит в завершении процесса. Для некоторых сигналов, таких как SIGFPE и SIGSEGV, процесс завершается с созданием посмертного дампа памяти. Дамп создается в текущем каталоге в файле с именем core (современные системы, в зависимости от настроек, могут добавлять к core идентификатор процесса или другую информацию).

Есть группа сигналов, реакция на которые состоит в остановке процесса. Это такие сигналы, как:

- SIGTSTP — Terminal StoP, остановка с терминала, генерируется при нажатии <Ctrl>+<Z> (ASCII SUB);
- SIGSTOP — безусловная остановка, генерируется только программно;
- SIGTTIN, SIGTOUT — ввод и вывод с терминала, генерируются для фонового процесса, который пытается вести диалог с пользователем;
- SIGTRAP — точка останова в отладчике.

Процесс, остановленный такими сигналами, не завершается, но приостанавливается. Чтобы возобновить его исполнение, необходимо послать ему сигнал SIGCONT. Сигнал SIGTRAP используется отладчиками; остальные сигналы этой группы применяются командными процессорами, поддерживающими управление заданиями, такими как csh, ksh и bash.

Обработчик сигнала получает единственный параметр — номер обрабатываемого сигнала. Таким образом, один обработчик может использоваться для нескольких сигналов, но, с другой стороны, представления обработчика о том, в каком контексте произошла ошибка, следует описать как достаточно смутные. Современные Unix-системы реализуют несколько типов параметризованных сигналов, которые, кроме номера сигнала, могут получить или запросить дополнительную информацию о контексте ошибки [Стивенс 2002].

Обработчик имеет тип void, т. е., в соответствии с синтаксисом языка C, не возвращает значения. Таким образом, набор средств, с помощью которых обработчик сигнала может сообщить основной нити программы об ошибке, достаточно узок; далее в этом разделе мы рассмотрим некоторые методы, с помощью которых можно достичь более тесной координации действий между обработчиками и основной нитью программы.

Одной из важных проблем при работе с сигналами является то, что сигнал проходит, вообще говоря, в произвольные моменты по отношению к исполнению основной нити программы, поэтому при взаимодействии обработчика с основной нитью мы сталкиваемся со всем тем букетом проблем, который описывался

в главе 7, — критическими секциями, нереентерабельными функциями и т. д. Основным средством решения этих проблем является маскирование сигналов.

Процесс имеет атрибут, называемый *маской сигналов* (signal mask). Этот атрибут представляет собой битовую маску, в которой каждому типу сигнала соответствует один бит. Общее количество допустимых сигналов невелико, в старых системах оно не превосходило 32, в новых — ненамного превосходит это значение, поэтому 64-разрядной маски вполне достаточно. Мaska проверяется при каждой попытке доставить сигнал; если соответствующий бит установлен, то сигнал не доставляется — ни установленный программой, ни системный обработчики не вызываются, системные вызовы не прерываются, но, в отличие от игнорируемых сигналов, информация о сигнале сохраняется: когда бит в маске будет сброшен, сигнал будет доставлен. Современные системы предоставляют вызовы, которые реализуют атомарные блоки операций, например установку обработчика и размаскирование сигнала или, напротив, размаскирование сигнала и вход в системный вызов `pause()` (такой блок может быть полезен для избежания ошибок потерянного пробуждения, которые мы рассматривали в разд. 7.3.).

Critical Error Handler в MS/DR DOS

В MS.DOS достаточно современных версий (я должен признать, что недостаточно знаком с историей этой системы, чтобы сказать, когда именно этот механизм появился, но точно знаю, что в DOS 3.30 он уже был) можно зарегистрировать обработчик для довольно обширной группы ошибок, прежде всего связанных с операциями ввода/вывода. Типичными примерами таких ошибок являются обращение к дисководу, в котором отсутствует дискета, попытка записи на защищенную дискету, несхождение контрольной суммы у прочитанного сектора диска и т. д.

Стандартный обработчик этих ошибок встроен в ядро системы; он выдает сообщение об ошибке и хорошо знакомый пользователям DOS вопрос: "Abort, Retry, Fail?" (или, для некоторых ошибок: "Abort, Retry, Ignore, Fail?"); пользователь должен ответить, нажав клавишу, соответствующую первой букве каждого из возможных ответов. Abort приведет к принудительному завершению текущей программы, Retry — к повторной попытке выполнения операции (например, если ошибка состоит в отсутствии дискеты в дисководе, пользователь может вставить дискету и нажать <R>), Fail — к возврату кода ошибки программе. Действие ответа Ignore зависит от операции.

Текст сообщения и вопрос всегда выдаются серыми буквами на черном фоне, независимо от настроек видеoadаптера. Буквы рисуются средствами BIOS, который "знает", какой видеорежим сейчас установлен, поэтому они появятся на экране даже в графическом режиме — но, разумеется, испортят картинку, поэтому большинство графических приложений для DOS и системы Windows 1.x—3.x (которые, в определенном смысле, были не более, чем графическими надстройками над DOS) были вынуждены зарегистрировать собственный обработчик и рисовать из этого обработчика модальный диалог, обычно с переключателями, обозначенными теми же самыми словами Abort, Retry, Fail.

Обработчик получает параметры в регистрах процессора. В регистре AH передается классификация ошибки в виде набора битовых флагов. Различаются дисковые и "недисковые" ошибки, произошла ли ошибка в результате чтения

или записи и какие реакции на нее допустимы. Всегда допустима реакция Abort, в то время как допустимость реакций Ignore, Fail и Retry задается независимыми битами. В регистре AL передается номер логического диска ("буква", т. е. то, что в DOS называется drive letter), в регистре DI — код ошибки, в регистрах BP:SI — указатель на блок состояния устройства, из которого можно попытаться получить дополнительную информацию. Список кодов ошибок закрытый, их всего 12 [Финогенов 2001].

При возврате обработчик передает в регистре AL код, соответствующий выбранной реакции — 0 соответствует Ignore, 1 — Retry, 2 — Abort, 3 — Fail. Если во входных параметрах Fail был запрещен, а обработчик все-таки вернет Fail, последствия в документации описываются как непредсказуемые; на практике, впрочем, с высокой достоверностью можно предсказать, что система по-просту "зависнет".

Главным источником сложностей при разработке Critical Error Handler является тот факт, что он вызывается из ядра DOS, а поскольку ядро не является реентерабельным, то код обработчика должен соблюдать жесткие ограничения на то, какие системные вызовы он может исполнять. Нарушение этих ограничений обычно приводит к разрушению системы, т. е. опять-таки "зависанию". В системах семейства Unix эта проблема обойдена за счет того, что обработчик сигнала всегда прерывает системный вызов, безотносительно к тому, порожден сигнал попыткой исполнения этого вызова или каким-то посторонним событием.

Обработчики ошибок лишены одного из основных недостатков кодов ошибок — они пригодны для обработки как синхронно, так и асинхронно возникающих ошибок. Однако недостатки этой методики также весьма значительны.

Один из недостатков уже упоминался в обсуждении сигналов Unix-систем: обработчик вызывается, вообще говоря, асинхронно, поэтому он имеет лишь ограниченное представление о том, в каком состоянии находится основная нить программы, поэтому неясно, как, используя обработчик, можно изменить поведение программы для обхода или компенсации ошибки. Наиболее естественным применением обработчиков представляется "аккуратное" — т. е. с освобождением всех занятых ресурсов и сохранением данных — завершение задачи. Однако даже при фатальной ошибке обработчик не может сам организовывать сохранение данных на диск, потому что копия данных в оперативной памяти может находиться в несогласованном состоянии.

Ряд других недостатков, которые мы также наблюдали и применительно к сигналам Unix-систем и применительно к Critical Error Handler в MS/DR DOS, также порождается асинхронным исполнением сигналов — это проблемы доступа к критическим секциям и нереентерабельным процедурам как в установленной обработчик программе, так и в ядре системы.

Своебразную проблему представляет собой опасность рекурсивного вызова обработчика, если при его работе вновь возникло то же самое исключение. При обработке аппаратных исключений во многих процессорах, в том числе

и в x86, на этот случай предусмотрено специальное исключение, называемое Double Fault (двойной отказ), — если при обработке, скажем, ошибки сегментации снова произойдет ошибка сегментации, будет вызван не тот же самый обработчик, а обработчик Double Fault, который, как правило, аварийно завершает работу ОС и рисует на экране регистры.

Интересно отметить, что Critical Error Handler в DOS не может быть вызван рекурсивно, потому что он не имеет права исполнять ни один из системных вызовов, которые могут привести к возникновению Critical Error.

12.4.3. Стековые обработчики ошибок и простая обработка исключений

Как я уже отмечал, главной проблемой простых обработчиков ошибок является недостаточность знаний обработчика о контексте исполнения основной нити программы. Эта проблема в довольно широких пределах может решаться за счет того, что регистрация обработчика ошибки — это, в сущности, дешевая операция, которая просто сводится к записи указателя на функцию-обработчик в определенную таблицу. Поэтому программа может легко и часто заменять эти обработчики; таким образом, каждый из возможных обработчиков может знать, из какого места программы он может быть вызван; если соответствующий объем кода невелик, можно даже с достаточно высокой достоверностью определить, в каком именно месте кода возникла данная ошибка.

Рассмотрим пример 12.1 (нам еще придется вернуться к нему в этом разделе, чтобы объяснить, что делают функции `setjmp/longjmp`, но пока нас интересует другой аспект данной программы). Установка обработчика сигнала делается функцией (на самом деле, это системный вызов) `signal`; этот вызов встречается в коде два раза.

Пример 12.1. Обработка исключения Floating Underflow
(антипереполнение при операциях с плавающей точкой)

```
#include <setjmp.h>
static jmp_buf fpe_retry;

void fpe_handler (int sig) {
    _fpreset();
    signal (SIGFPE, fpe_handler);
    longjmp(fpe_retry, -1);
}
```

```
int compare_pgms(Image * img0, Image * img1)
{
    int xsize=256, ysize=256;
    int i,j, p0, p1, pd;
    double avg, avgsq, scale, smooth;

    scale=(double)xsize*(double)ysize;
    avg = 0.0;
    avgsq = 0.0;

    /* Подавить возможные антипеперполнения */
    signal (SIGFPE, fpe_handler);

    for(i=0; i<ysize; i++) {
        smooth = (double)(img0->picture[i*xsize]-img1->picture[i*xsize]);
        for(j=0; j<xsize; j++) {
            p0=img0->picture[j+i*xsize];
            p1=img1->picture[j+i*xsize];
            pd=(p0-p1);
            if (setjmp(fpe_retry) == 0) {
                smooth = smooth*(1.0-SMOOTH_FACTOR)+(double)pd*SMOOTH_FACTOR;
                avg += smooth;
                avgsq += smooth*smooth;
            } else smooth=0.0;
        }
    }

    if (setjmp(fpe_retry) == 0)
        dispersion = avgsq/scale-avg*avg/(scale*scale);
    else dispersion = 0.0;
    signal (SIGFPE, SIG_DFL);
}
```

В этом примере подсчитывается разность двух черно-белых растровых изображений размером 256×256 точек и среднеквадратичное отклонение этой разности от нуля (по историческим причинам, код не совсем оптимален; впрочем, оптимизирующий компилятор заменяет инвариантные переменные константами и получает довольно приличный код). Среднее и среднеквадратичное отклонение вычисляются не от самой разности, а от грубо "сглаженного" значения, чтобы компенсировать присутствующие в изображениях шумы.

мы. Затем вычисляется квадрат дисперсии (поэтому название переменной *dispersion* не вполне соответствует действительности) и сохраняется в глобальной переменной.

Этот пример пытается обойти возможные "антипереполнения" (underflow) — выход числа с плавающей точкой за допустимый диапазон представления, но не в направлении слишком больших, а в направлении слишком малых (по абсолютному значению) чисел. В данном случае мы подсчитываем довольно грубую статистику и для нас результат антипереполнения ничем не отличается от нуля — поэтому мы можем обойти данную ошибку простым отождествлением такого результата с нулем (в общем случае это некорректно).

Видно, что обработчик исключения устанавливается перед входом в цикл и снимается после выхода. Мы знаем, в каком именно месте кода возникает антипереполнение (в действительности, в этом месте возникает *первое антипереполнение*, но мы пресекаем возможную цепочку ошибок у самого ее начала), и знаем также, что никаких других источников арифметических ошибок в этом коде нет — хотя там также присутствуют и деления, но их делители никогда не бывают нулевыми. Поэтому мы интерпретируем все возникающие SIGFPE как антипереполнение, возникшее в определенной точке нашего кода, и принимаем меры к обходу именно этого антипереполнения — при том, что эти меры привели бы к катастрофическим результатам при попытке компенсировать ими все остальные теоретически возможные арифметические ошибки.

Таким образом, мы сформулировали достаточно простой подход, позволяющий скомпенсировать обсуждаемый недостаток простых обработчиков ошибок. А именно, мы выбираем участки кода, в которых все ошибки одного типа могут быть обработаны одним и тем же способом — это может получаться как за счет того, что в этом участке только один источник таких ошибок, так и благодаря стечению каких-то других обстоятельств. Затем мы пишем обработчик, который реализует именно этот способ обработки, устанавливаем его в начале выбранного нами участка кода и снимаем в конце.

Очевидный недостаток этого подхода состоит в том, что если наш код вызывает какие-то другие процедуры, функции или методы, и мы не знаем, что происходит внутри них, то в таком виде этот подход неприемлем. Однако он допускает простое и очевидное усовершенствование. Если мы обрабатываем некоторую ошибку (тот же SIGFPE) и вызываем функцию, которая желает обрабатывать эту ошибку каким-то иным образом, то мы просто договариваемся, что эта функция может ставить все обработчики, которые ей нужны, при условии, что перед выходом она восстановит наши обработчики.

Иными словами, если при установке нашего обработчика мы будем запоминать предыдущий обработчик той же ошибки, а при снятии — восстанавлив-

вать его, то мы получим рекурсивную схему, которая хорошо соответствует порядку исполнения кода в языках с блочной структурой. Код в примере 12.1 не выполняет это соглашение (он предполагает, что до его вызова сигнал SIGFPE обрабатывался обработчиком по умолчанию), но вообще-то системный вызов `signal` имеет такое описание:

```
void (*signal)(int signo, void (*handler)(int))(int);
```

Требуется достаточно глубокое знание языка C, чтобы понять, что означает такая конструкция, поэтому в современной документации предпочитают более длинную, но более понятную запись:

```
typedef void (*sighandler_t)(int);
sighandler_t signal (int signum, sighandler_t handler);
```

Первая строка этой записи определяет тип `sighandler_t` — указатель на функцию с типом возвращаемого значения `void` (т. е. не возвращающую значения) и с параметром типа `int`. Вторая строка описывает функцию `signal`, которая возвращает значение этого типа, т. е. указатель на функцию. Эта вторая функция принимает два параметра, первый из них типа `int` (номер сигнала), второй — указатель на функцию того же типа, что и возвращаемое значение.

Я надеюсь, читатель уже догадался, что системный вызов `signal` принимает в качестве входного параметра новое значение обработчика, а возвращает старое значение, которое, скорее всего, следует запомнить и восстановить перед выходом из блока, в котором будет действовать наш обработчик. Таким образом, обработчики сигналов естественным образом выстраиваются в стековую структуру: каждая процедура может установить свой собственный обработчик на время своей работы и вернуться к старому обработчику перед возвратом.

Пример 12.2 демонстрирует еще один интересный прием, посредством которого обработчик исключения может вмешиваться в исполнение основной нити программы. А именно, он возвращает управление не в точку, в которой произошла ошибка, а в некоторую другую точку. Это достигается с помощью пары своеобразных функций, предоставляемых стандартной библиотекой языка C, называющихся `setjmp` и `longjmp`.

Функция `setjmp` сохраняет контекст исполнения процесса (или нити) в момент своего вызова. Во всяком случае, она сохраняет значение указателя стека (и других регистров, формирующих стековый кадр) и адрес возврата. Сохраненные значения помещаются в переменную `jmp_buf`. Затем эта функция возвращает управление вызвавшей процедуре так, как будто ничего не произошло.

Если в какой-то другой момент мы восстановим сохраненные значения регистров (именно это и делает функция `longjmp`), во многих отношениях это будет выглядеть так, будто функция `setjmp` вернула управление еще раз. Во всяком случае, управление будет передано в точку возврата из этой функции.

Видно, что пара функций `setjmp/longjmp` в определенных отношениях напоминает функцию переключения сопрограмм `ProcessSwitch`, которую мы обсуждали в *главе 8*, — она тоже запоминает состояние стека одной нити и потом, в какой-то другой момент, восстанавливает это состояние. Впрочем, между `setjmp/longjmp` есть некоторые важные отличия. В частности, при вызове `ProcessSwitch` состояние стека остановленной нити "консервируется", пока `ProcessSwitch` не вернет нити управление, состояние ее стека не может измениться. Напротив, после вызова `setjmp` исполнение нити продолжается обычным порядком, и состояние стека может меняться.

Из-за этого `longjmp` можно вызывать только из функций, которые были вызваны (возможно, через несколько уровней вложенности) из функции, сделавшей `setjmp`. При этом уничтожаются стековые записи активации всех промежуточных функций, но стековый кадр первой функции останется в стеке и исполнение программы сможет продолжаться нормально. Напротив, если первая функция к моменту вызова `longjmp` уже успеет возвратить управление, ее стековый кадр будет уничтожен, и восстановление указателей стека и стекового кадра приведет к полному разрушению стека.

Функции `setjmp` и `longjmp` совершают манипуляции над стеком и регистрами, которые невозможно реализовать на большинстве языков высокого уровня, поэтому их приходится реализовать с помощью ассемблерных вставок или просто на ассемблере. В примере 12.2 приведены исходные тексты этих функций из библиотеки EMX (эмуляция Unix-совместимого API для OS/2).

Пример 12.2. Исходный текст функций `setjmp/longjmp`

```
/ setjmp.s (emx+gcc) -- Copyright (c) 1990-1996 by Eberhard Mattes

#include <emx/asm386.h>

.globl _setjmp, _longjmp

.text
ALIGN

#define J_EBX 0
#define J_ESI 4
```

```
#define J_EDI 8
#define J_ESP 12
#define J_EBP 16
#define J_EIP 20
#define J_XCP 24

/ Слова со смещениями 28..44 зарезервированы

/ int setjmp (jmp_buf here)

_setjmp:
PROFILE_NOFRAME
movl 1*4(%esp), %edx /* здесь*/
movl %ebx, J_EBX(%edx)
movl %esi, J_ESI(%edx)
movl %edi, J_EDI(%edx)
movl %ebp, J_EBP(%edx)
movl %esp, J_ESP(%edx)
movl .0*4(%esp), %eax /* Адрес возврата */
movl %eax, J_EIP(%edx)
cmpb $0, __osmode /* OS/2? */
je 1f /* No -> skip */
fs
movl 0, %eax /* handler - Обработчик исключений */
movl %eax, J_XCP(%edx)
1: xorl %eax, %eax
EPILOGUE(setjmp)

ALIGN

/ void longjmp (jmp_buf there, int n)

_longjmp:
PROFILE_NOFRAME
cmpb $0, __osmode /* OS/2? */
je 2f /* нет -> пропустить */
movl 1*4(%esp), %eax /* там */
pushl J_XCP(%eax)
call __unwind2 /* восстановить обработчики сигналов */
addl $4, %esp
```

```

2: movl 1*4(%esp), %edx /* там */
   movl 2*4(%esp), %eax /* n */
   testl %eax, %eax
   jne 3f
   incl %eax

3: movl J_EBX(%edx), %ebx
   movl J_ESI(%edx), %esi
   movl J_EDI(%edx), %edi
   movl J_EBP(%edx), %ebp
   movl J_ESP(%edx), %esp
   movl J_EIP(%edx), %edx
   movl %edx, 0*4(%esp) /* адрес возврата */
EPILOGUE(longjmp) /* ну, ... */

```

Используя переустановку обработчиков сигналов и, когда это необходимо, `setjmp/longjmp`, можно реализовать достаточно сложные и адекватные задаче схемы обработки ошибок. Однако у теоретиков программирования этот механизм вызывает, в лучшем случае, смешанные чувства — слишком уж очевидно, что это не встроенное средство языка, а какая-то надстройка над ним или даже нащепка сбоку.

Обработка исключений в VMS

В VMS, а также в OS/2, Win32 и ряде других систем применяется схема обработки исключений, в ряде отношений существенно более совершенная, чем сигналы Unix. Рассмотрим подсистему CHF (Condition Handling Facility — средство обработки [нештатных] ситуаций) ОС VMS [VMSDOC CHF].

Ситуации (condition) VMS используются для обработки синхронно возникающих нештатных состояний. Для обработки асинхронно возникающих ошибок используются AST (Asynchronous System Trap — асинхронный системный обработчик), аналогичные сигналам Unix. Ситуации могут обнаруживаться аппаратурой (например, деление на 0 или ошибка защиты памяти) или системным программным обеспечением. Пользовательская программа также может самостоятельно генерировать ситуации посредством системных вызовов `LIB$SIGNAL` и `LIB$STOP`.

Ситуация описывается 32-битовым значением `condition value`, которое разбивается на четыре битовых поля (рис. 12.3). Значения такого формата используются в качестве кода возврата большинства системных процедур; кроме того, эти значения передаются обработчикам событий.

Младшие четыре бита характеризуют серьезность ситуации. Младший бит представляет собой логическое значение, 1 — успеху операции, 0 — неуспеху. Следующие три бита описывают собственно уровень серьезности в соответствии с табл. 12.1.



Рис. 12.3. Формат condition value в VMS

Таблица 12.1. Уровни серьезности ситуации в OpenVMS

Значение	Символ	Серьезность	Смысл
0	STSSK_WARNING	Warning	Исполнение может продолжаться, результат непредсказуем
1	STSSK_SUCCESS	Success	Исполнение может продолжаться, результат соответствует ожиданиям
2	STSSK_ERROR	Error	Исполнение может продолжаться, ошибка
3	STSSK_INFO	Information	Исполнение может продолжаться, выводится информационное сообщение
4	STSSK_SEVERE	Severe error	Исполнение прервано
5			Reserved for HP
6			Reserved for HP
7			Reserved for HP

Видно, что из восьми возможных кодов используются только пять, и лишь некоторые из них соответствуют собственно ошибкам. Системный вызов LIB\$SIGNAL может использоваться для сообщения ситуаций всех уровней серьезности, LIB\$STOP всегда генерирует сообщения уровня STSSK_SEVERE.

Далее следуют два 12-битовых поля. Старшее из этих полей (facility number — номер подсистемы) описывает подсистему, которая пытается сигнализировать о ситуации, а младшее (message number — номер сообщения) представляет собой код ситуации.

Старший бит facility number используется для разделения системных и пользовательских подсистем: 0 соответствует системным компонентам, 1 — пользовательским.

Значения message number, у которых старший бит равен 1, глобальны для всей системы. Напротив, коды ошибок с нулевым старшим битом специфичны для подсистем.

Старшие четыре бита называются control (управляющие). Из них реально используется только младший бит: он подавляет распечатку сообщения об ошибке стандартным обработчиком. Остальные биты зарезервированы и должны быть равны 0.

Таким образом, VMS предоставляет весьма обширное сегментированное пространство кодов, которые могут использоваться для обозначений как ошибок, так и штатных ситуаций.

Для обработки ситуаций пользовательская программа может зарегистрировать обработчик. Основным способом регистрации является запись указателя на точку входа процедуры-обработчика в зарезервированное слово в стековом кадре процедуры. Если процедура не регистрирует собственный обработчик, то соответствующее слово должно быть нулевым.

При возникновении ситуации система просматривает список обработчиков исключений, определенных в следующих местах:

- первичный вектор обработчиков исключений, устанавливаемый системой. В действительности, это четыре таблицы для каждого из четырех уровней привилегий процессора. Эти векторы используются системой и отладчиками, однако пользователь может выключать некоторые из этих обработчиков;
- вторичный вектор обработчиков. По умолчанию эта таблица пуста. Она используется некоторыми системными программами, пользовательским программам не рекомендуется заполнять ее;
- обработчики, определенные в стековых кадрах, в порядке, обратном порядку вызовов (т. е. последний из созданных стековых кадров обрабатывается первым);
- обработчики traceback и catch-all. Адреса этих обработчиков записаны в заголовке EXE-файла и устанавливаются при сборке программы;
- вектор обработчиков "последнего шанса" (last chance exception vector) — в действительности, это также четыре вектора для каждого из уровней привилегий. Пользовательский обработчик последнего шанса по умолчанию отсутствует, но отладчики обычно регистрируют такой обработчик. Уровни привилегий ядра и привилегированных процедур (executive) имеют обработчики последнего шанса, которые для нефатальных ситуаций генерируют записи в системном логе, а для фатальных инициируют останов системы.

Найдя обработчик, система вызывает его, передавая в качестве одного из параметров код ситуации. В зависимости от кода ситуации и контекста, обработчик может предпринять одно из следующих действий:

- попытаться продолжить выполнение программы. Ситуации делятся на ловушки (trap) и отказы (fault). Если ситуации была типа trap, исполнение продолжается с команды, непосредственно следующей за командой, породив-

шай эту ситуацию. Если ситуация была типа *fault*, то делается попытка повторить исполнение команды, породившей ситуацию. Для ошибок уровня *STS\$K_SEVERE* продолжение невозможно;

- повторный сигнал (*resignal*). Таким образом обработчик сообщает системе, что он не может обработать эту ситуацию. Система пытается продолжить поиск обработчика с той точки, на которой остановилась. Если *resignal* был возвращен обработчиком первичного или вторичного векторов, начинается поиск в стеке; если *resignal* был возвращен стековым обработчиком, система просматривает стек дальше и т. д. Обработчики последнего шанса, разумеется, не имеют права возвращать *resignal*;
- породить другую ситуацию вызовом *LIB\$SIGNAL* или *LIB\$STOP*;
- отмотать стек (*SYS\$UNWIND*). Обычно отмотка стека происходит либо к записи активации процедуры, зарегистрировавшей обработчик, либо к процедуре уровнем выше (т. е. к процедуре, вызвавшей ту процедуру, которая зарегистрировала обработчик). Исполнение возобновляется с точки возврата процедуры, внутри которой возникла ситуация. В действительности, обработчик может отмотать стек на произвольное количество записей активации, но в большинстве случаев обработчик недостаточно хорошо знает контекст возникновения ситуации, чтобы пользоваться этим;
- в OpenVMS для DEC/Compaq Alpha и IA64 (Intel Itanium) доступна также функция *SYS\$GOTO_UNWIND*, которая отматывает стек на определенное количество записей активации и передает управление в произвольную точку соответствующей процедуры.

Более сложные схемы обработки исключений, которые мы рассмотрим далее, представляют собой, в конечном итоге, лишь некоторое синтаксическое укращение описанных в этом разделе схем. Эти украшения хороши тем, что более понятны и защищают от некоторых грубых ошибок (например, от вызова *longjmp* в момент, когда сохраненный стековый кадр уже разрушен).

К сожалению, при всех преимуществах рассматриваемых далее средств обработки ошибок, все они привязаны к определенным языкам программирования. Если разработчики ОС пытаются обеспечить "языковую нейтральность" или, попросту говоря, хотят обеспечить исполнение программ, написанных на разных языках высокого уровня, они оказываются ограничены примитивными средствами информирования об ошибках, такими как коды ошибок и простые или, в лучшем случае, стековые процедуры-обработчики. Трансляция этих примитивных средств в высокоуровневые конструкции ЯВУ возлагается на среду исполнения языка — стандартную библиотеку, интерпретатор и т. д.

12.4.4. Обработка исключений в стиле PL/I

Ряд языков высокого уровня предоставляют своеобразную группу синтаксических конструкций, предназначенных для обработки ошибок. Сами эти кон-

струкции в разных языках различны и не имеют общепринятого родового названия; однако при взгляде на эти наборы конструкций становится очевидно их концептуальное родство. Я не могу точно проследить генеалогию этого подхода, но самый старый из относительно распространенных языков, в котором он последовательно реализован, — это PL/I, разработанный компанией IBM во второй половине 60-х годов XX века.

Из современных языков, аналогичный подход используется в диалектах языка Basic (Visual Basic, Lotus Script) и в некоторых менее известных языках, например в REXX. Рудиментарные формы этого подхода можно обнаружить даже в языках COMMAND.COM, CMD.EXE (командные процессоры в MS/DR DOS и OS/2/Windows NT соответственно) и их родственниках, таких как 4DOS/4OS2/4NT.

Рассмотрим обработку ошибок в языках Visual Basic и Lotus Script (все сказанное далее относится к обоим языкам). В этих языках описываемый механизм достиг, пожалуй, наивысшего своего развития, а сами языки достаточно распространены и широко известны. Необходимо отметить, что в самом современном диалекте языка — VB.Net — используется более совершенный синтаксис, похожий на применяемый в C++/Java/C#, который мы будем рассматривать в следующем разделе. Синтаксис, который мы будем рассматривать сейчас, применялся в VB версий 5 и 6, а в VB.Net он сохраняется только для совместимости со старым кодом.

Механизм позволяет обрабатывать ошибки, возникающие при работе стандартных функций интерпретатора (например, при работе с файлами) и при вызовах компонентов расширения, например объектов COM. Кроме того, программист может самостоятельно генерировать ошибки, вызывая оператор Error, точный синтаксис которого следующий:

```
Error Errno [Errmsg]
```

где Errno — целое число, номер ошибки. Errmsg — текстовая строка, сообщение об ошибке и, возможно, о контексте, в котором ошибка возникла. Параметр Errno обязательен, Errmsg может быть пропущен.

Для обработки ошибки программист должен исполнить оператор On Error. Синтаксис этого оператора таков:

```
On Error [Errno] statement
```

Errno — номер обрабатываемой ошибки. Если он не указан, оператор устанавливает универсальный обработчик для всех типов ошибок; если указан — то только для данного конкретного кода ошибки.

Statement — это оператор, указывающий на то, какие действия следует предпринять для обработки ошибки. Точный список допустимых операторов

сильно варьируется в зависимости от диалекта языка, но основные варианты следующие:

- Resume Next — очистить код ошибки и передать управление на оператор, следующий за вызвавшим ошибку. Если ошибка произошла во время вызова функции, то управление будет передано не внутрь этой функции, а на следующий за вызовом оператор функции, в которой настроен обработчик;
- Resume label — очистить код ошибки и передать управление на метку label. Метка должна быть определена в текущей подпрограмме;
- Goto label — перейти на метку label. При этом код ошибки не очищается, и можно получить информацию о том, какая именно ошибка произошла, в какой строке функции она произошла и каков текст сообщения об ошибке. Кроме того, при неочищенном коде ошибки можно вызывать операторы Resume label и Resume Next;
- Goto 0 — выключить обработку ошибок указанного типа и использовать стандартный обработчик, который состоит в прекращении исполнения программы и выдаче сообщения об ошибке в виде модального диалога с кнопкой OK;
- Goto -1 — игнорировать ошибку;
- Exit Sub — очистить код ошибки и вернуть управление из текущей процедуры или функции.

Функция или процедура может содержать несколько операторов On Error; они исполняются наравне с обычными операторами, и при возникновении ошибки производится действие, соответствующее последнему из выполненных операторов, относившихся к этому коду ошибки.

Если функция не содержит ни одного оператора On Error для возникшей ошибки, то происходит возврат из этой функции и вызывается оператор On Error вызвавшей функции.

Видно, что механизм, при всей его простоте, достаточно гибок и позволяет реализовать достаточно сложные сценарии обработки ошибок и восстановления после них. Одно из остроумных его применений приводится в примере 12.3 — так называемый "отладчик". Такой обработчик ошибок повторно генерирует ошибку, которую не может исправить, но добавляет к сообщению об ошибке название своей функции и строку, в которой она произошла. Если все процедуры, функции и методы в программе содержат такие обработчики, то полное сообщение об ошибке, которое будет в конце концов выдано пользователю, будет содержать полный стек вызовов на момент возникновения ошибки, вместе с содержательным сообщением, порожденным при генерации

этой ошибки. Разумеется, пользователю эта информация не очень-то интересна, но для разработчика она может оказаться неоценимой важной.

Пример 12.3. Пример обработчика ошибок — "отладчик" на языке Lotus Script

```
Sub Foo ()  
    On Error Goto Handler  
    ...  
' Код программы  
    ...  
' Оператор Exit Sub необходим, чтобы не исполнить обработчик ошибки  
' при нормальном завершении процедуры  
    Exit Sub  
Handler:  
    Error Err(), Error()+"at line "+cstr(erl())+" of Sub Foo"  
End Sub
```

При всей простоте и привлекательности этот синтаксис плохо подходит для языков с блочной структурой и, особенно, для объектно-ориентированных языков с блочной структурой, таких как C++ и Java/C#. Действительно, при исполнении оператора *Goto*, метка которого определена внутри блока, мы должны исполнить конструкторы всех переменных, определенных в этом блоке (и его родительских блоках), до метки, на которую мы переходим. Напротив, при выходе из блока мы должны выполнить деструкторы всех таких переменных. При обработке ошибок такое поведение может оказаться нежелательным, но чтобы контролировать этот аспект поведения, мы должны сделать операторы обработки ошибок блочными.

12.4.5. Исключения C++/Java/C#

В конце предыдущего раздела я указал на важную проблему, которую необходимо решить при синтаксической поддержке обработки ошибок в блочных и, особенно, в объектно-ориентированных блочных языках: чтобы точно знать, какие объекты существуют к моменту исполнения обработчика ошибок, а какие — должны быть к этому моменту уничтожены, сами операторы обработки ошибок должны иметь блочную структуру.

Для решения этой проблемы в C++ был введен оператор *try/catch*. В действительности, описываемый подход, как и обработка исключений в стиле PL/I, имеет давнюю и почтенную историю и восходит, как минимум, к языку Simula 67, спецификации которого были опубликованы в 1967 году. Однако я буду описывать реализацию этого подхода в современных языках. Синтаксис оператора *try/catch* таков:

```
try-block :  
    try compound-statement handler-list  
handler-list :  
    handler handler-listopt  
handler :  
    catch ( exception-declaration ) compound-statement  
exception-declaration :  
    type-specifier-list declarator  
    type-specifier-list abstract-declarator  
    type-specifier-list  
;
```

Здесь `compound-statement` обозначает составной оператор языка C++, т. е. блок операторов, заключенных в фигурные скобки.

Сами ошибки генерируются оператором `throw`, синтаксис которого следующий:

```
throw-expression :  
    throw assignment-expressionopt
```

При исполнении оператора `throw` создается промежуточный объект, вообще говоря, произвольного типа. Если программист хочет, чтобы код был осмысленным, этот объект должен содержать описание произошедшей ошибки в той или иной форме.

Затем делается попытка найти вверх по стеку вложенных блоков и вызовов оператор `catch`, в описании которого указан объект совместимого типа, т. е. такого, к которому можно привести объект описания ошибки в соответствии с правилами неявного преобразования типов в C/C++. При этом необходимо помнить, что C++ допускает полиморфизм типов. Если в `catch` был указан родительский класс, а в `throw` — наследник этого класса, то типы будут признаны совместимыми. Знатоки языка C++ должны отметить, что на самом деле все несколько сложнее, но точное описание требует подробного изложения принципов наследования и преобразования типов в C++, которое увело бы нас далеко в сторону от обсуждаемой темы. Правильное и точное описание всех деталей семантики этих операторов может быть найдено в книгах по C++, например в [Страуструп 1999].

Если такой оператор найден, описание ошибки присваивается переменной, определенной в `exception-declaration` этого блока, и блок исполняется. Перед этим "отматывается стек" (*stack unwind*), т. е. исполняются деструкторы всех объектов с классом памяти `auto`, определенных во всех блоках, из которых нам придется выйти, прежде чем мы достигнем нужного оператора `catch`.

Если такого оператора `catch` не найдено, управление перехватывается средой исполнения C++, которая запускает деструкторы объектов, определенных в функции `main`, далее запускает деструкторы статических объектов и завершает программу.

Поскольку к моменту исполнения `catch`-блока стек уже отмотан, то никакого эквивалента оператору `Resume Next` операторы `try/catch` не предлагают и предложить не могут.

Пример использования этой конструкции показан в примере 12.4.

Пример 12.4. Использование операторов `try/catch` в языке C++

```
// exceptions_trycatchandthrowstatements.cpp,
// цит. по [MSDN C++ exception]
#include <iostream>

using namespace std;
int main()
{
    char *buf;
    try
    {
        buf = new char[512];
        if( buf == 0 )
            throw "Memory allocation failure!";
    }
    catch( char * str )
    {
        cout << "Exception raised: " << str << '\n';
    }
    // ...
    return 0;
}
```

Таким образом, каждая ошибка идентифицируется двумя параметрами или, точнее, одним параметром и группой параметров — типом объекта, порожденного в момент `throw`, и значениями атрибутов этого объекта. Каждый оператор `try` может иметь несколько `catch`-операторов; если внутри блока `try` произойдет ошибка, то именно по типу `throw`-объекта и будет определяться, какой из `catch`-блоков будет вызван.

Видно, что операторы `try/catch` вынуждают программиста писать довольно много кода — возможно, меньше, чем при использовании кодов ошибок, но больше, чем при использовании операторов `Error/On Error`.

В Java синтаксис оператора `try/catch` несколько усложнен: введена ветвь `finally`, в которой можно написать обработчик всех ошибок, тип которых не совместим ни с одним из операторов `catch`. Сами объекты описания ошибок при попадании в ветвь `finally` отбрасываются.

Кроме того, в прототипе (или, как любят говорить Java-разработчики, в контракте) каждого метода Java должен указываться список всех типов исключений, которые этот метод порождает. Компилятор проверяет, что если наш метод вызывает какие-то другие методы, то либо мы сами должны обработать порождаемые ими исключения, либо указать эти типы исключений в контракте нашего метода.

На первый взгляд, это шаг в сторону закрытого списка ошибок. В действительности, ограничение накладывается только на типы объектов описания ошибок, но не на значения атрибутов этих объектов. Кроме того, никто не мешает порождать в качестве объектов описания ошибок экземпляры классов-наследников. Это может создать некоторые сложности — например, в C++ может потребоваться использование нестандартных средств RTTI (Run-Time Type Information — информация о типах [доступная] во время исполнения), — но также и позволяет передавать информацию о новых типах ошибок без потери совместимости со старым API.

В реализациях C++ и C для Win32, во всяком случае в MS Visual C и Borland C, реализованы аналогичные операторы `_try` и `_except`, которые позволяют перехватывать исключения Win32. Этот механизм известен как SEH (Structured Exception Handling — структурированная обработка исключений). Вместо произвольных объектов, которые можно использовать в стандартных исключениях C++, в операторе `_except` должна использоваться специальная встроенная функция `filter`, позволяющая задать список обрабатываемых кодов ситуаций и получить код текущего исключения. Кроме того, в отличие от стандартных исключений C++, при передаче управления в оператор `_except` не происходит отмотки стека, так что можно попытаться продолжить исполнение с точки, в которой возникло исключение.

Синтаксис `_try/_except` запатентован компанией Borland, поэтому разработчики свободно распространяемых компиляторов, таких как GCC, не могут его реализовать.

Вопросы для самопроверки

1. Можете ли вы привести точное формальное определение ошибки?
2. Какие способы обнаружения ошибок вы можете назвать?
3. Приведите примеры синхронных и асинхронных ошибок.
4. Почему во многих случаях единственной разумной реакцией на ошибку оказывается аварийное завершение процесса или даже аварийное завершение работы системы? Почему при ошибках, вообще говоря, невозможно сохранять данные?
5. Сравните преимущества и недостатки открытого и закрытого списка обрабатываемых ошибок.
6. Каковы недостатки и ограничения кодов ошибок? Почему современные языки программирования отказываются от этого метода?
7. Каковы недостатки простых обработчиков ошибок?
8. Почему обработка ошибок в стиле PL/I нашла лишь ограниченное применение? Почему в современных языках программирования более популярна структурированная обработка исключений?
9. Как исключения VMS можно использовать для реализации обработки ошибок в стиле PL/I? Можно ли использовать исключения VMS для реализации операторов `try/catch` языка C++?
10. Пример 12.2 представляет собой реализацию функций `setjmp/longjmp` для OS/2. Можно ли по коду этой реализации понять структуру стекового кадра, которую использует компилятор GCC для OS/2? Совпадает ли структура стекового кадра в OS/2 и в системах семейства Unix? (для ответа на этот вопрос вам, возможно, придется найти исходные тексты `setjmp/longjmp` для Linux, BSD или Solaris). Если нет, в чем состоит отличие?
11. Безопасно ли совместное использование исключений C++ и `setjmp/longjmp`? Можно ли вообще использовать `setjmp/longjmp` в программах на C++? Что говорит по этому поводу документация среды программирования, которой вы пользуетесь?



ГЛАВА 13

Безопасность

Just cause you're paranoid
Don't mean they aren't after you.

K. Cobain

То, что ты параноик,
Не значит, что они тебя не преследуют.

К. Кобэйн

По мере компьютеризации общества в электронную форму переносится все больше и больше данных, конфиденциальных по своей природе: банковские счета и другая коммерческая информация, истории болезни и т. д. Проблема защиты пользовательских данных от нежелательного прочтения или модификации встает очень часто и в самых разнообразных ситуациях — от секретных баз данных Министерства обороны до архива писем к любимой женщине.

Причин, по которым пользователь может желать скрыть или защитить свои данные от других, существует очень много, и в подавляющем большинстве случаев эти причины достойны уважения.

Совершенствование средств доступа к данным и их совместного использования всегда порождает и дополнительные возможности несанкционированного доступа. Наиболее ярким примером являются современные глобальные сети, которые предоставляют доступ к огромному богатству информационных сред. Эти же сети представляют серьезную угрозу безопасности подключющихся к сети организаций. Основная специфика угрозы заключается в том, что в таких сетях злоумышленнику значительно легче обеспечить свою анонимность даже в случае обнаружения факта "взлома". Поэтому в наше время глобальных открытых информационных систем вопросы безопасности приобретают особую важность.

13.1. Формулировка задачи

А мальчик-юзер пошел пить пиво со злыми интернетчиками, а те ему и говорят: "Да какой ты крутой хаксер. Почту ты сломал, факт. А вот этот сервер попробуй сломать". И в окно показывают. А напротив пивняка стоит здание, с надписью "Почта". И мальчик-юзер пошел почту ломать. А на почте, видать, сниффер стоял, так что через пять минут менты-модераторы приехали и так отмодерили мальчика-юзера, что с тех пор о нем ничего не известно.

Vinar Velazquez

Идеальная система безопасности должна обеспечивать полностью прозрачный санкционированный доступ к данным и непреодолимые трудности при попытках доступа несанкционированного. Кроме того, она должна предоставлять легкую и гибкую систему управления санкциями; во многих случаях бывает также полезно отслеживать все попытки несанкционированного доступа.

К сожалению, несмотря на огромную практическую важность, единой и связной теории безопасности вычислительных систем на сей день не разработано. Отчасти это объясняется сложностью и многосторонностью задачи: несанкционированный доступ к данным, например, может быть получен не только путем удаленного "взлома" вычислительной системы, но и посредством физического похищения компьютера или носителей данных, или с помощью подкупа сотрудника организации, имеющего доступ к данным. Следовательно, идеальная система безопасности должна предусматривать средства защиты от всех физически реализуемых способов несанкционированного доступа.

Хорошо спроектированная система попутно выполняет также и другие полезные функции, защищая данные от ошибочных модификаций (и, таким образом, в известной мере защищая пользователей системы от самих себя), внешние каналы — от посторонних людей, забредших по ошибке и т. д., но все-таки о конечной цели ее создания никогда не следует забывать.

Для того чтобы понять, способна ли система безопасности к выполнению своей основной функции, ее разработчик и администратор хотя бы время от времени должны становиться на точку зрения противника и искать в своей системе потенциальные точки атаки, которые одновременно являются слабыми местами в защите.

Атаки на систему безопасности можно разделить на различные категории по разным принципам. С точки зрения целей, атаки можно разделить на три класса:

- DoS (Denial of Service — дословно, отказ в сервисе), т. е. приведение атакуемой системы или одной из ее подсистем в неработоспособное состояние;

- собственно несанкционированный доступ к данным или другим ресурсам атакуемой системы;
- захват контроля над системой, обычно, но не всегда, с целью атаки на какие-то другие системы.

Разумеется, наиболее опасны атаки третьего типа — прежде всего потому, что система, уязвимая для таких атак, представляет проблему не только для ее владельца, но и для окружающих, особенно для тех, кто этому владельцу в том или ином отношении доверяет. В современных условиях, по ряду причин, защите от атак такого типа уделяется слишком мало внимания, непропорционально мало по отношению к их опасности.

Возможно, положение могло бы исправиться, если бы владелец использованной таким образом системы нес хотя бы гражданско-правовую ответственность за причиненный его бездействием ущерб. Однако введение подобных принципов в законодательство привело бы к тому, что большинство владельцев Windows-систем оказались бы вынужденны выплачивать значительные суммы (трудно найти машину под управлением Windows, которая ни разу не оказывалась заражена вирусом или червем или не была включена в ботнет). По-видимому, не следует вводить такие законы, не возлагая хотя бы часть ответственности на разработчика уязвимой системы, но это противоречит бизнес-моделям ряда поставщиков ПО, которые продают программы без контракта на их поддержку — так называемое "коробочное" ПО (*shrink-wrap software*), в первую очередь Microsoft. Поэтому такие законы натолкнутся на сильное лоббистское противодавление.

Атаки DoS сами по себе, как правило, менее опасны, чем атаки двух других классов, но им следует уделять большое внимание по нескольким причинам.

Во-первых, для осуществления такой атаки достаточно найти неадекватно обрабатываемую ошибку в коде или настройках атакуемого сервиса, либо найти ресурс, который нужен этому сервису и использование которого не ограничено квотами. В этом смысле, любая фатальная ошибка в коде сервиса или любой неквотируемый ресурс, а особенно — ресурс, который можно расходовать, не аутентифицируясь в системе (например, описатели сетевых соединений), представляет собой потенциальную точку DoS. Поэтому для осуществления DoS взломщику обычно требуется гораздо меньше собственных ресурсов — как материальных, так и интеллектуальных, чем для доступа к данным или внедрения троянского кода.

Во-вторых, вероятность обнаружения взломщика после успешного DoS часто гораздо ниже, чем после успешного доступа к данным. Например, взломщика банковской системы, который перевел деньги себе на счет, можно поймать при попытке физического снятия денег ("путаница следов" усложняет задачу идентификации преступника, но не делает ее невыполнимой), взломщик же,

который ограничился тем, что однажды нарушил работу сервера транзакций, не вступает более ни в какие взаимодействия с системой и поэтому не всегда может быть прослежен.

Из-за этой и предыдущей причин многие взломщики из спортивного интереса часто удовлетворяются осуществлением атаки DoS над целевой системой. Взломщики, осуществляющие попытку вторжения в систему, часто используют DoS-атаки для отвлечения внимания администраторов атакуемой системы или при "заметании следов" после основной атаки. Большую роль в современных условиях играют DoS-атаки против средств активной безопасности — снiffeров, антивирусных пакетов и т. д.

В-третьих, отказ критически важного для организации сервиса хотя и меньшее зло, чем систематическая враждебная деятельность по анализу конфиденциальных данных, но все равно явление весьма неприятное и способное привести к значительным финансовым потерям.

В современных условиях большую опасность представляют сетевые DoS-атаки затоплением (flood attack), при которых на атакуемую систему отправляется большое количество запросов на соединение с различных узлов сети. Для организации такой атаки используются ботнеты или сетевые черви, а иногда и просто спуфинг (подделка исходящего адреса у пакетов с запросами). Известно несколько попыток организовать такую атаку социальными инженерными средствами: так, в 2003 году, когда Олег Кубаев (автор известной в русском Интернете серии мультфильмов про Масяню) судился с владельцами домена mult.ru, во многих сетевых форумах распространялся призыв организовать атаку на сайт mult.ru, посыпая на него запросы ping (ICMP ECHO); впрочем, ко времени выхода этой книги успешных крупномасштабных атак такого типа еще не было известно.

Аналогичные явления могут возникать и без злого умысла. Так, в Сети известно явление, называемое slashdotting: когда на популярном новостном сайте www.slashdot.org публикуют ссылку на какой-либо ресурс, размещенный на маломощном сервере или на сервере с низкоскоростным подключением к сети, резкое увеличение количества запросов может надолго парализовать этот сервер; серверы под управлением Windows нередко даже падают в "синий экран".

Классификация атак с точки зрения средств — гораздо более сложная задача; я склоняюсь к тому, что исчерпывающая классификация по этому параметру невозможна. Тем не менее следует упомянуть несколько типов атак.

- Атаки, требующие физического доступа к системе, начиная от физического захвата жестких дисков или системного блока целиком и включая кратковременный несанкционированный доступ к терминалу. В данной книге мы практически не будем обсуждать эти атаки и средства защиты от них.

К счастью, опасность таких атак очевидна даже наиболее технически неграмотным пользователям, поэтому большинство реальных систем в основном удовлетворительно защищены от них.

Отдельного внимания требуют атаки, которые предполагают лишь кратковременный доступ и могут осуществляться скрыто — доступ к незащищенному терминалу, установка подслушивающих устройств и т. д. Далее в этом разделе мы вкратце рассмотрим некоторые особенности защиты от таких атак.

- Социальная инженерия, т. е. провоцирование пользователей системы на совершение действий, которые нужны злоумышленнику. Без особых наложений к этой же категории можно отнести атаки, основанные на знании того, как пользователи обычно нарушают требования безопасности — например, словарную атаку на систему парольной аутентификации (*см. разд. 13.3*). В данной книге мы также практически не будем обсуждать эти типы атак — вовсе не из-за того, что они не заслуживают внимания, а напротив, потому, что эта тема слишком обширна и важна, а краткое обсуждение может создать у читателя впечатление ложного знания. Неплохой обзор социально-инженерных приемов приводится в книге [Шнайер 2003].
- Атаки, основанные на ошибках в коде или настройках системы. В зависимости от типа ошибки, эти атаки могут требовать разного начального уровня доступа к системе — например, срыв буфера в публичном сетевом сервисе позволяет получить полный доступ к системе любому пользователю Интернета; ошибкой в списке контроля доступа может воспользоваться только тот, кто уже имеет учетную запись в системе, и т. д. Поэтому для таких атак возможна довольно естественная классификация по уровню их опасности — от осуществимых любым абонентом Интернета (наиболее опасные) до осуществимых лишь привилегированным пользователем системы (наименее опасные).
- Троянская программа, т. е. помещение в атакуемую систему написанного взломщиком кода. Код при этом размещается таким образом, чтобы его рано или поздно должны были запустить. Поскольку код системы обычно так или иначе защен, внедрение такой программы часто само по себе является атакой.

Разумеется, на практике встречаются гибридные (то есть относящиеся к нескольким классам одновременно) и комбинированные (то есть состоящие из отдельных шагов, относящихся к различным классам) атаки.

Сама возможность комбинированных атак несколько обесценивает упомянутую ранее классификацию ошибок по их опасности: так, если ошибка в раздаче прав дает возможность зарегистрированному пользователю системы прочитать те данные, которых он читать не должен, но пользователь добро-

совестный и сам их читать не будет, то проблему можно считать не очень серьезной. Но если злоумышленник средствами социальной инженерии спровоцирует этого добросовестного пользователя срочно уйти в другую комнату, а сам сядет за оставленный без присмотра терминал или, пользуясь какой-то другой ошибкой в системе, получит к ней доступ от имени этого пользователя, опасность такой ошибки резко повышается.

Вообще, когда я говорил, что современные вычислительные системы удовлетворительно защищены от несанкционированного физического доступа, я имел в виду, что оборудование обычно удовлетворительно защищено от кражи и от ситуаций, когда злоумышленник остается наедине с системным блоком на несколько минут — время, за которое можно загрузиться с компакт-диска и прочитать данные с диска в обход средств защиты ОС. К сожалению, современная вычислительная техника уязвима для воздействий, которые требуют гораздо меньше времени и могут осуществляться незаметно — например таких, как уже упоминавшийся доступ к оставленному без присмотра терминалу.

Перехват клавиатурного ввода

Часто недооцениваемая опасность — подключение к компьютеру устройств, которые перехватывают и запоминают вводимые с клавиатуры данные (keylogging — запись клавиатуры). Современные технологии позволяют собрать такое устройство в габаритах клавиатурного разъема. Кейлоггеры с довольно большим объемом флэш-памяти и/или с передатчиком беспроводного доступа производятся серийно и доступны по весьма умеренным ценам. Причем доступны они не только на черном рынке: некоторые компании устанавливают такие устройства на рабочие станции сотрудников, чтобы иметь возможность контролировать их действия. Нередко это имеет смысл — например, на рабочем месте операциониста в банке.

Подключение кейлоггера к компьютеру может быть осуществлено за несколько секунд, особенно если злоумышленник имеет доступ к задней панели системного блока. Так, нередко приходится видеть в приемных многих контор компьютер секретаря, стоящий на столе; при этом, разумеется, передней панелью он обращен к секретарю, а задней — к посетителю.

Один из моих студентов, фамилию которого я по понятным причинам не могу раскрыть, рассказывал мне о самодельном устройстве для записи клавиатуры, которое он собрал, когда еще учился в школе. Его отец пытался контролировать доступ сына к домашнему компьютеру, чтобы пресечь чрезмерное увеличение компьютерными играми. Контроль осуществлялся с помощью пароля BIOS, без ввода которого система отказывалась загружаться.

Сын знал, что пароль состоит из одних цифр. Он разобрал старый калькулятор и клавиатуру и припаял клавиши с цифрами к соответствующим клавишам калькулятора. Затем он собрал клавиатуру и прилепил калькулятор изолентой к нижней части корпуса клавиатуры. Таким образом, после включения компьютера все вводимые цифры оставались на дисплее калькулятора. Разумеется, объем дисплея был недостаточно велик, чтобы запомнить все вводившиеся цифры, но ведь требовалось запомнить только первые введенные!

О том, было ли устройство обнаружено и если да, то что последовало за его обнаружением, история умалчивает.

Идеальная система безопасности, способная защитить от всех вышеперечисленных (а также от возможных, но не упомянутых) типов атак, по-видимому, физически не реализуема. В лучшем случае удается исключить отдельные способы, но для идеального решения задачи это нужно сделать по отношению ко всем способам получения несанкционированного доступа. На практике обычно исходят из требований "разумной достаточности" или экономической целесообразности: с одной стороны, стоимость установки и эксплуатации систем безопасности не должна превосходить ценности защищаемых данных. С другой, "экономически идеальная" система безопасности должна быть достаточно сложной, для того чтобы выгоды потенциального взломщика от получения доступа были ниже затрат на преодоление защиты.

Практическое применение этого критерия требует оценки стоимостей и их сравнения. Использование как показателя цен сопряжено с серьезной методологической сложностью, которую вкратце можно описать следующим образом: рыночная цена любого предмета — это цена, по которой обладатели такого предмета, желающие его продать, реально могут его продать, а желающие купить — соответственно, могут купить. В нашем же случае обладатель защищаемого объекта часто вовсе не намерен его продавать, а те, от кого он защищается, не планируют его покупать. Даже если полный эквивалент защищаемого объекта может быть куплен, требуемая для этого сумма является лишь нулевым приближением для оценки реального ущерба пострадавшего или прибыли взломщика.

И в том случае, когда охраняемый объект — это деньги (например, база данных со счетами клиентов банка), потери от его похищения или нарушения целостности часто значительно превосходят непосредственно потерянную при этом сумму денег. Так, банк, пострадавший от взлома системы управления счетами, теряет не только переведенную на счет взломщика сумму, но и доверие клиентов.

Многие из объектов, с которыми вынуждены иметь дело разработчики систем безопасности, вообще не могут быть куплены. Все примеры конфиденциальных данных, перечисленных в начале этой главы, относятся именно к этой категории. Если охраняемый объект в принципе не покупается и не продается, то его цена попросту не определена (что, впрочем, не означает, что такой объект заведомо нельзя оценить).

Ожидаемую прибыль взломщика также бывает сложно оценить. Например, польза от прочтения и анализа данных в одном домашнем компьютере обычно очень невелика; однако, захватив контроль над большим количеством таких компьютеров, взломщик может получать некоторые, порой неожидан-

ные, и порой довольно значительные в денежном выражении преимущества. Так, статистика посещений Интернет-сайтов одним человеком практически никому не интересна, но такая статистика, собранная по достаточно большой выборке пользователей, представляет собой ценные данные для маркетологов и организаций, занимающихся размещением рекламы.

Захваченный компьютер может применяться как средство при атаке на какую-то другую, более ценную систему — например, банк. Это позволяет взломщику сохранить свою анонимность. В современных условиях *ботнеты* (botnet, искаженное сокращение от robot net, сеть удаленно управляемых захваченных компьютеров) используются для рассылки почтового спама, распределенных атак затоплением, запуска вирусов и червей. Сеть из тысячи "ботов", пригодных для рассылки спама, продается на черном рынке за несколько сотен долларов.

Критерием принятия решения в такой ситуации могут быть лишь: субъективное решение владельца охраняемого объекта, сравнивающего стоимости объекта и стоимости охранных систем или те или иные представления о субъективной системе ценностей самого взломщика, оценивающего возможные прибыли и издержки.

Формулируя такие представления, разработчик архитектуры безопасности вынужден принимать во внимание также и людей, для которых вскрытие чужих систем безопасности является самоцелью, своего рода спортом. Обсуждение вопроса о том, можно ли охарактеризовать систему ценностей таких людей как извращенную, уведет нас далеко от темы книги. Для целей дальнейшего обсуждения важно отметить, что классификация этих людей как извращенцев и даже заочная постановка им того или иного мозговедческого диагноза никак не может защитить нас от них самих и результатов их деятельности.

Таким образом, заказчик системы безопасности вынужден принимать решение, в основе которого лежат не поддающиеся строгому обоснованию и, возможно, неверные представления о системах ценностей других людей. Задача, стоящая перед ним, находится в близком родстве с задачей, которую решает предприниматель, пытающийся оценить рыночные перспективы того или иного товара. Как и предприниматель, заказчик системы безопасности в случае принятия неверного решения рискует понести серьезные материальные потери, поэтому принятие решения о структуре и стоимости системы безопасности вполне можно считать разновидностью предпринимательского решения.

Принятие предпринимательских решений в общем случае является неформализуемой и неалгоритмизуемой задачей. Во всяком случае, для ее формализации и алгоритмизации необходимо, ни много ни мало, создать полную уни-

версальную формальную алгоритмическую модель человеческой психики, которая полностью описала бы поведение не только предпринимателя, но и всех его потенциальных деловых партнеров (а в случае системы безопасности — и всех потенциальных взломщиков этой системы). Даже если эта задача в принципе и разрешима, то, во всяком случае, она находится далеко за пределами возможностей современной науки и современных вычислительных систем. Возможно, этот факт и является фундаментальной причиной, по которой теория систем безопасности представляет собой логически несвязанное сочетание эмпирических, теоретических или "интуитивно очевидных" рекомендаций разного уровня обоснованности.

Несмотря на все методологические и практические сложности, с которыми сопряжено применение экономической оценки к системам безопасности, по крайней мере, некоторые практически важные рекомендации этот подход нам может дать.

Во-первых, мы можем утверждать, что хорошая система безопасности должна быть сбалансированной, причем прежде всего с точки зрения взломщика: стоимости всех мыслимых путей получения доступа к системе должны быть сопоставимы. И, наоборот, должны быть сопоставимы стоимости мероприятий по обеспечению защиты от проникновения разными способами. Бесмысленно ставить бронированные ворота в сочетании с деревянным забором.

Это соображение, впрочем, ни в коем случае не должно удерживать нас от применения мер, которые при небольшой стоимости необычайно затрудняют несанкционированный доступ. Принцип сбалансированности необходимо применять в первую очередь к дорогостоящим, но при этом умеренно эффективным мероприятиям.

Вторая полезная рекомендация: если это оправдано по стоимостным показаниям, делать системы безопасности многослойными или, используя военную терминологию, эшелонированными. Пройдя внешний слой защиты (например, преодолев файрволл и установив прямое соединение с одним из серверов приватной сети компании), взломщик должен получать доступ не непосредственно к данным, а лишь к следующему слою защиты (например, средствам аутентификации ОС или серверного приложения).

Третья рекомендация в известной мере противоречит двум предыдущим и гласит, что, учитывая компоненты стоимости эксплуатации системы безопасности, нельзя забывать о тех действиях, которые эта система требует от сотрудников организации. Если требования безопасности чрезмерно обременительны для них, то они могут попытаться осуществить операцию, которую в неоклассической экономике называют экстернализацией издержек (например, выдвинуть ультимативное требование — либо вы снимаете наиболее

раздражающие из требований, либо повышаете зарплату, либо мы всеуволимся).

Более неприятная для разработчика системы безопасности перспектива — это отказ (не всегда сознательный) от выполнения слишком обременительных (или воспринимающихся как таковые) требований, создающий неконтролируемые дыры в системе безопасности. Например, ключи могут оставляться под половиком, пароли — записываться на прилепленных к монитору бумагах, конфиденциальные данные — копироваться на недостаточно защищенные настольные или переносные компьютеры. В сетях многих "режимных" предприятий и государственных учреждений доступ к Интернету разрешен только с некоторых выделенных компьютеров; в современных условиях это нередко приводит к тому, что сотрудники устанавливают на своих рабочих станциях несанкционированные средства для доступа к Интернету, например через сотовые телефоны с поддержкой GPRS.

В дальнейшем в этой главе мы будем обсуждать то подмножество задачи обеспечения безопасности, с которым практически сталкиваются разработчики программного обеспечения и системные администраторы. А именно, в большинстве случаев мы будем предполагать, что помещение, где размещена вычислительная система, защищено от несанкционированного проникновения, а персоналу, который имеет прямой или удаленный доступ к системе, в определенной мере можно доверять. Причина, по которой мы принимаем эти предположения, совершенно прозаична: в большинстве современных организаций за обеспечение такой безопасности отвечает не *системный администратор*, а совсем другие люди. На практике, хотя системный администратор и не обязан быть по совместительству специалистом в вопросах охраны и подбора персонала, но он должен так или иначе взаимодействовать с этими людьми, вырабатывая согласованную и сбалансированную политику защиты организации.

Не следует думать, что эти допущения позволяют решить перечисленные задачи идеально или, тем более, что они могут быть решены идеально. Не означают эти предположения также и того, что без разрешения данных задач системный администратор вообще не может приступить к работе — более того, мы рассмотрим и некоторые подходы к решению нашей задачи в ситуации, когда эти предположения выполняются лишь частично или не выполняются совсем. Большинство таких подходов основано на шифровании и электронной подписи данных.

Даже в такой ограниченной формулировке задача обеспечения безопасности вовсе не проста и требует дополнительной декомпозиции. Задачу управления доступом к данным и операциям над ними разбивают на две основные подзадачи: *автентификацию* (проверку, что пользователь системы действительно

является тем, за кого себя выдает) и *авторизацию* (проверку, имеет ли тот, за кого себя выдает пользователь, право выполнять данную операцию).

13.2. Сессии и идентификаторы пользователя

Петли дверные
Многим скрипят, многим поют:
"Кто вы такие,
Вас здесь не ждут!"
B. Высоцкий

Как правило, далеко не каждая авторизация отдельных операций сопровождается актом аутентификации. Чаще всего используется принцип *сессий* работы с вычислительной системой. В начале работы пользователь устанавливает соединение и "входит" в систему. При "входе" происходит его аутентификация.

Для того чтобы быть аутентифицированным, пользователь должен иметь *учетную запись* (*account*) в системной базе данных. Затем пользователь проводит сеанс работы с системой, а по завершении этого сеанса аннулирует регистрацию.

Одним из атрибутов сессии является *идентификатор пользователя* (*user id*) или *контекст доступа* (*security context*), который и используется при последующих авторизациях. Обычно такой идентификатор имеет две формы: числовой код, применяемый внутри системы, и мнемоническое символьное имя, используемое при общении с пользователем.

Сессии в Unix

Например, в системах семейства Unix пользователь идентифицируется целочисленным значением *uid* (*user identifier*). С каждой задачей (процессом) связана два идентификатора пользователя: реальный и эффективный. В большинстве случаев эти идентификаторы совпадают (ситуации, когда они не совпадают, подробно обсуждаются в разд. 13.5.4). Таким образом, каждая задача обязательно исполняется от имени того или иного пользователя, имеющего учетную запись в системе.

Пользователь может иметь также символьное имя. В старых Unix-системах соответствие между символьным и числовым идентификаторами устанавливалось на основе содержимого текстового файла */etc/passwd*. Каждая строка этого файла описывает одного пользователя и состоит из семнадцати полей, разделенных символом ':'. В первом поле содержится символьное имя пользователя, во втором — числовой идентификатор в десятичной записи. Остальные поля содержат другие сведения о пользователе, например, его полное имя.

Пользовательские программы могут устанавливать соответствие между числовым и символьным идентификаторами самостоятельно, путем просмотра файла */etc/passwd*, или использовать библиотечные функции, определенные стан-

дартом POSIX. Во многих реализациях эти функции используют вместо /etc/passwd индексированную базу данных, а сам файл /etc/passwd сохраняется лишь для совместимости со старыми программами.

В современных системах семейства Unix библиотеки работы со списком пользователей имеют модульную архитектуру и могут использовать различные, в том числе и распределенные по сети базы данных. Интерфейс модуля работы с конкретным типом БД называется *PAM (Pluggable Authentication Module — подключаемый модуль аутентификации)* (рис. 13.1).

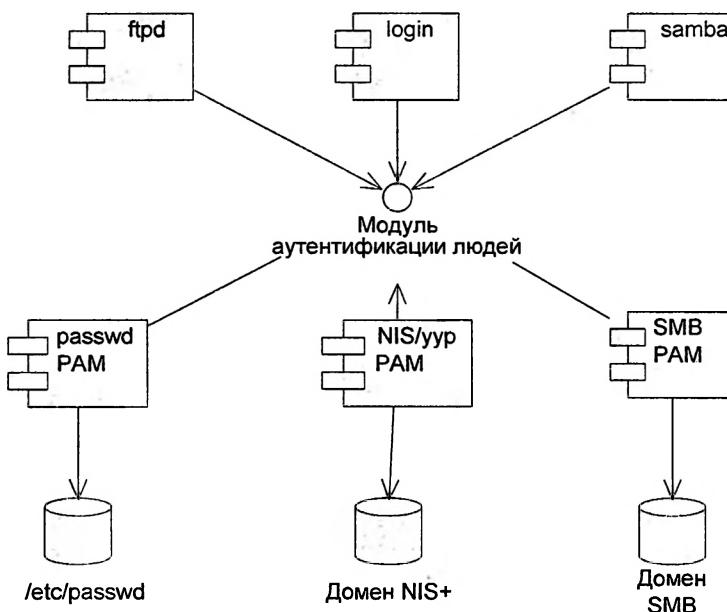


Рис. 13.1. PAM и различные базы учетных записей

Нужно отметить, что соответствие между символьным и числовым идентификаторами в Unix не является взаимно однозначным. Одному и тому же числовому идентификатору может соответствовать несколько имен. Кроме того, в Unix разрешено создавать объекты с числовым uid, которому не соответствует никакое символьное имя.

Большинство современных ОС позволяют также запускать задания без входа в систему и создания сессии. Так, практически все системы разделения времени (Unix, VMS, MVS-OS/390-z/OS) предоставляют возможность пользователям запускать задачи в заданные моменты астрономического времени или периодически, например, в час ночи в пятницу каждой недели. Каждая такая задача исполняется от имени определенного пользователя — того, кто запросил запуск задачи. Для управления правами доступа в таких ситуациях идентификатор пользователя ассоциируется не с сессией, а с отдельными зада-

ниями, а обычно даже с отдельными задачами. В Windows NT/2000/XP задачи, которые могут запускаться и работать без входа пользователя в систему, называются *сервисами*. По умолчанию, сервисы запускаются от имени специального [псевдо]пользователя System, но в свойствах сервиса можно указать, от чьего имени он будет запускаться. Кроме того, некоторые комплектации системы (Terminal Server Edition, Citrix ICA, не включенная в домен Windows XP) допускают одновременную или поочередную интерактивную работу нескольких пользователей. Чтобы обеспечить разделение доступа во всех этих случаях, каждый процесс в системе имеет *контекст доступа* (*security context*), соответствующий той или иной учетной записи.

13.3. Аутентификация

Вот еще: однъятесь свеже, и на выставке в Манеже
К вам приблизится мужчина, с чемоданом будет он.
Спросит — хочете черешни? — вы ответите — конечно.
Он вам даст батон с взрывчаткой, принесете мне батон.
В. Высоцкий

Понятно, что если права доступа выделяются на основе машинного идентификатора пользователя, то возникает отдельная проблема установления соответствия между этим машинным идентификатором и реальным человеком. Такое соответствие не обязано быть взаимно однозначным: один человек может иметь несколько идентификаторов или наоборот, несколько человек могут пользоваться одним идентификатором. Тем не менее способ установления такого соответствия необходим.

По-английски процесс входа в систему называется *login* (*log in*) и происходит от слова *log*, которое обозначает регистрационный журнал или процесс записи в такой журнал. В обычном английском языке такого слова нет, но в компьютерной лексике слова *login* и *logout* прижились очень прочно.

Наиболее точным переводом слова *login* является регистрация. Соответственно, процесс выхода называется *logout*. Его точная русскоязычная калька — разрегистрация — к сожалению, очень неблагозвучна.

Теоретически можно придумать много разных способов идентификации, например, с использованием механических или электронных ключей или даже тех или иных биологических параметров, например рисунка глазного дна. Однако подобные способы требуют специальной и зачастую довольно дорогой аппаратуры. Наиболее широкое распространение получил более простой метод, основанный на символьных паролях.

Пароль (*parole* — фр. слово) представляет собой последовательность символов. Предполагается, что пользователь запоминает ее и никому не сообщает.

Этот метод хорош тем, что для его применения не нужно никакого дополнительного оборудования — только клавиатура, которая может использоваться и для других целей. Но этот метод имеет и ряд недостатков.

Использование паролей основано на следующих трех предположениях:

- пользователь может запомнить пароль;
- никто не сможет догадаться, какой пароль был выбран;
- пользователь никому не сообщит свой пароль.

Последнее предположение кажется разумным: если человек может добровольно кому-то сообщить свой пароль, с примерно той же вероятностью этот человек может сам сделать пакость. Впрочем, человека можно заставить выдать пароль. При этом речь не обязательно идет о каких-то серьезных физических угрозах; в соответствии с результатами анонимного опроса, проводившегося в 2004 году компанией BBC, 70% англичан согласны рассказать свой пароль в обмен на плитку шоколада [bbc.co.uk password] (организаторы опроса не сообщают, проверяли ли они полученные таким образом пароли и, если да, то каковы были результаты проверки). Так или иначе защита от подобных ситуаций требует мер, которые не могут быть обеспечены на уровне операционной системы.

Если вдуматься, первые два требования отчасти противоречат друг другу. Количество легко запоминаемых паролей ограничено и перебрать все такие пароли оказывается не так уж сложно. Этот вид атаки на систему безопасности известен как *словарная атака* (*dictionary attack*) и при небрежном отношении пользователей к выбору паролей и в ряде других ситуаций, которые будут обсуждаться далее, представляет большую опасность.

Словарная атака столь успешна потому, что при выборе пароля человек использует не весь свой активный словарный запас (который обычно измеряется несколькими тысячами или даже десятками тысяч слов), а слова, которые, во-первых, имеют достаточную длину, а во-вторых, в определенном смысле представляются "необычными". Практика показывает, что количество таких "необычных" слов в действительности невелико.

Словарная атака в исполнении червя Morrisa

Возможности словарной атаки впервые были продемонстрированы в 1987 году молодым тогда студентом Робертом Моррисом [КомпьютерПресс 1991]. Разработанная им программа — "червь Морриса" — использовала словарную атаку и последующий вход в систему с подобранным паролем как один из основных способов размножения. Червь использовал для распространения и другие приемы, продемонстрировавшие не только специфические изъяны тогдашних ОС семейства Unix, но и несколько фундаментальных проблем компьютерной безопасности, поэтому мы еще несколько раз будем возвращаться к обсуждению этой программы.

Червь Морриса использовал при подборе паролей следующие варианты:

- входное имя пользователя;
- входное имя с символами в обратном порядке — "задом наперед";
- компоненты полного имени пользователя — имя (first name), фамилию (last name) и другие элементы, если они есть;
- те же компоненты, но задом наперед;
- слова из заранее определенной таблицы, содержащей 400 элементов.

Большая часть успешно подобранных паролей была взята из таблицы. После поимки червя таблица была опубликована с тем, чтобы пользователи не повторяли старых ошибок. Эта таблица приводится в [КомпьютерПресс 1991]. По моему мнению, включение ее в данное издание нецелесообразно, поскольку она рассчитана на англоязычных пользователей и не очень актуальна в России.

С первого взгляда видно, что таких "легко запоминаемых" вариантов довольно много — никак не меньше нескольких сотен. Набор их вручную занял бы очень много времени, но для компьютера несколько сотен вариантов — это доли секунды.

Применение словарных атак резко затрудняется, если вместо отдельных слов использовать словосочетания и предложения. Во-первых, благодаря этому резко возрастает количество возможных комбинаций. Во-вторых, значительно увеличивается полезный объем словаря — при выборе компонентов для словосочетания пользователь не сталкивается ни с ограничениями на длину отдельных слов, ни с пожеланием их "необычности". Поэтому в современной англоязычной документации, особенно при описании приложений с высокими требованиями к качеству пароля, вместо слова *password*, которое содержит довольно-таки явный для англоязычного читателя намек, что речь идет об отдельном слове, предпочитают употреблять слово *passphrase*, которое столь же явно намекает, что речь идет о словосочетании или полном предложении.

Первый слой защиты от подбора пароля заключается именно в том, чтобы увеличить время подбора. Обнаружив неправильный пароль, наученные горьким опытом современные системы делают паузу, прежде чем позволят повторную попытку входа с того же терминального устройства. Такая пауза может длиться всего одну секунду, но даже этого достаточно, чтобы увеличить время подбора пароля от долей секунды до десятков минут.

Второй слой защиты заключается в том, чтобы усложнить пароль и тем самым увеличить количество вариантов. Даже очень простые усложнения сильно увеличивают перебор. Так, простое требование использовать в пароле буквы и верхнего, и нижнего регистров увеличивает перебор в 2^n раз, где n — длина пароля. В большинстве современных систем пароль обязан иметь длину не менее восьми символов, т. е. количество вариантов увеличивается в

256 раз. Требование использовать в пароле хотя бы один символ, не являющийся буквой, увеличивает число вариантов еще в $8 \times 43 = 344$ раза (в наборе ASCII 43 небуквенных графических символов). Вместо десятков минут подбор пароля, который содержит буквы разных регистров и хотя бы один спецсимвол, займет много дней.

Но если взломщику действительно нужно попасть в систему, он может подождать и несколько дней, поэтому необходим третий слой защиты — ограничение числа попыток. Все современные системы позволяют задать число неудачно набранных паролей, после которого имя блокируется. Это число всегда больше единицы, потому что пользователь — это человек, а людям свойственно ошибаться, но в большинстве случаев такой предел задается не очень большим — обычно 5—7 попыток. Однако этот метод имеет и оборотную сторону — злоумышленник может использовать его для блокировки пользователей.

Интересный вариант того же метода заключается в увеличении паузы между последовательными неудачными попытками хотя бы в арифметической прогрессии.

Наконец, последний слой защиты — это оповещение пользователя (а иногда и администратора системы) о неудачных попытках входа. Если пользователь сам только что нажал на ту кнопку, он при входе увидит, что была одна неудачная попытка, и не будет волноваться; однако, если есть сообщения о дополнительных неудачных попытках, время побеспокоиться и разобраться, что же происходит.

Современная техника выбора паролей обеспечивает достаточно высокую для большинства практических целей безопасность. В тех случаях, когда эта безопасность представляется недостаточной, — например, если следует все-рьез рассматривать опасность принуждения пользователя к раскрытию пароля — можно применять альтернативные методы идентификации, перечисленные ранее, т. е. основанные на механических или электронных ключах или биометрических параметрах. Впрочем, как уже говорилось, такие методы требуют применения специальной аппаратуры.

Тем не менее систематическое несоблюдение правил выбора паролей во многих организациях приводит к проблемам. Так, в Интернете и в сетях университетских городков встречаются черви, размножающиеся по протоколу SSH с помощью низкоскоростного подбора паролей по словарю. Скорость размножения таких червей, разумеется, слишком низка, чтобы привести к пандемии, похожей на пандемию червя Морриса в 1987 году или червя Sapphire в 2003, но достаточно высока, чтобы поддерживать жизнеспособные популяции.

При использовании парольной аутентификации в распределенных вычислительных системах и в сетях возникает опасность перехвата пароля злоумыш-

ленником путем простого прослушивания сети или атаки "человек в середине", когда злоумышленник тем или иным способом имитирует систему, которая должна принимать аутентификацию. Решение этой проблемы требует применения криптографических механизмов или, точнее, сложных механизмов, включающих в себя криптографию как необходимую составную часть. Протоколы сетевой аутентификации обсуждаются в книге [Иргегов 2004] и в ряде других книг, посвященных сетевым технологиям: [Ганненбаум 2002, Ганненбаум/Ван-Стеен 2004] и др.

При использовании паролей возникает отдельная проблема безопасного хранения базы данных со значениями паролей. Как правило, даже администратор системы не может непосредственно получить значения паролей пользователей. Можно привести несколько соображений в пользу такого ограничения.

- Если бы администратор системы знал пользовательские пароли, то взломщик, сумевший выдать себя за администратора, так же получал бы доступ к этим паролям. После этого взломщик мог бы легко маскироваться под других пользователей, что сильно усложнило бы обнаружение взлома.
- Если администратор знает только административный пароль, для лишения его административных привилегий достаточно сменить этот пароль. Если же бывший администратор имел доступ ко всей базе данных о паролях, он по-прежнему будет иметь возможность доступа к системе, что может оказаться нежелательным.

Для обеспечения секретности паролей обычно используют одностороннее шифрование, или *хэширование*, при котором по зашифрованному значению нельзя восстановить исходное слово. При этом программа аутентификации кодирует введенный пароль и сравнивает полученное значение (*хэш*) с хранящимся в базе данных. Существует много алгоритмов хэширования, при использовании которых узнать реальное значение пароля можно только путем полного перебора всех возможных вариантов и сравнения зашифрованной строки со значением в базе данных.

В старых системах семейства Unix пароль использовался в качестве ключа шифрования фиксированной строки алгоритмом DES (см. главу 1). Этот алгоритм ограничивает длину ключа 64 битами, соответственно, пароли в таких системах могут содержать не более 8 символов (сам пароль может быть более длинным, но при аутентификации проверяются только первые восемь символов). Современные системы используют алгоритм MD5 [RFC 1321] и другие криптостойкие хэш-функции, которые допускают пароли практически неограниченной длины. Эти алгоритмы специально разрабатывались с целью максимального усложнения задачи построения сообщения с заданным значением хэша.

Отцам-основателям Unix такой механизм обеспечения секретности показался настолько надежным, что они даже не стали ограничивать доступ обычных пользователей к хэшам чужих паролей и выделили для их хранения поле в общедоступном для чтения файле /etc/passwd.

Однако "червь Морриса" продемонстрировал, что для подбора паролей не нужно перебирать все возможные комбинации символов, поскольку пользователи склонны выбирать лишь ограниченное подмножество из них. Процесс, имеющий возможность непосредственно читать закодированные значения паролей, может осуществлять подбор, не обращаясь к системным механизмам аутентификации, т. е. делать это очень быстро и практически незаметно для администратора.

После осознания опасности словарных атак разработчики систем семейства Unix перенесли значения паролей в недоступный для чтения файл /etc/shadow, где пароли также хранятся в односторонне зашифрованном виде. Это значительно усложняет реализацию словарной атаки, поскольку перед тем, как начать атаку, взломщик должен получить доступ к системе с привилегиями администратора. В наше время все способы получения несанкционированного доступа к парольной базе данных считаются "дырами" в системе безопасности.

Практически все современные системы хранят данные о паролях в односторонне зашифрованном виде в файле, недоступном для чтения обычным пользователям. Поставщики некоторых систем, например Windows NT/2000/XP, даже отказываются публиковать информацию о формате этой базы данных, хотя это само по себе вряд ли способно помешать квалифицированному взломщику.

13.4. Авторизация

Иннокентий на кухне, он пьет молоко,
Таракана завидев во мраке.
На душе его чисто, светло и легко,
Он не хочет впускать Полтораки.

Б. Гребеников

Механизмы авторизации в различных ОС и прикладных системах различны, но их трудно назвать разнообразными. Два основных подхода к авторизации — это ACL и полномочия.

ACL (Access Control List, список управления доступом) или, что то же самое, *список контроля доступа/полномочия (capability)*.

Список контроля доступа ассоциируется с объектом или группой объектов и представляет собой таблицу, строки которой соответствуют учетным записям пользователей, а столбцы — отдельным операциям, которые можно осущест-

вить над объектом. Перед выполнением операции система ищет идентификатор пользователя в таблице и проверяет, указана ли выполняемая операция в списке его прав.

Реализация списков управления доступом вполне прямолинейна и не представляет непреодолимых сложностей. Разработчики системы безопасности, впрочем, могут (и часто бывают вынуждены) предпринимать достаточно сложные меры для сокращения ACL, предлагая те или иные явные и неявные способы объединения пользователей и защищаемых объектов в группы.

Полномочие представляет собой абстрактный объект, наличие которого в контексте доступа задачи позволяет выполнять ту или иную операцию над защищаемым объектом или классом объектов, а отсутствие — соответственно, не позволяет. При реализации такой системы разработчик должен гарантировать, что пользователь не сможет самостоятельно сформировать полномочие.

Например, полномочие может быть реализовано в виде ключа шифрования или электронной подписи. Невозможность формирования таких полномочий обеспечивается непомерными вычислительными затратами, которые нужны для подбора ключа.

13.4.1. Списки контроля доступа

— Что это? — вытянул шею Гмык, хмуро глядя на мои карты. — Но тут же только...

— Минутку, — вмешался игрок слева от него. — Сегодня вторник. Выходит, его единороги дикие.

— Но в названии месяца есть "М"! — въякнул еще кто-то. — Значит, его великан идет за половину номинальной стоимости!

— Но у нас четное число игроков...

— Вы все кое-что упускаете. Эта партия — сорок третья, а Скив сидит на стуле лицом к северу!

Приняв за указание стоны и все заметнее вырисовывавшееся выражение отвращения на лицах, я сгреб банк.

P. Асприн

В общем случае совокупность всех ACL в системе представляет собой трехмерную матрицу, строки которой соответствуют пользователям, столбцы — операциям над объектами, а слои — самим объектам. С ростом количества объектов и пользователей в системе объем этой матрицы быстро растет, поэтому, как уже говорилось, разработчики реальных систем контроля доступа предпринимают те или иные меры для более компактного представления матрицы.

Хотя ухищрения для сокращения ACL дают определенный эффект, в большинстве случаев список имеет всего лишь несколько записей (рис. 13.2), на-

ложение ограничений на его размер часто считают неприемлемым или устанавливают такие ограничения весьма высокими, например в несколько тысяч записей. Это приводит к тому, что с файлом в ФС, поддерживающих списки контроля доступа, кроме основного массива данных, оказывается связан еще один массив, обычно уступающий в размерах основному, но, в принципе, способный достигать очень большого объема.

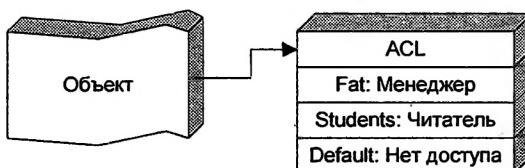


Рис. 13.2. Список контроля доступа

Впрочем, соответствующее усложнение файловой системы не так уж велико. Многие ФС позволяют хранить в дополнительных блоках данных не только записи ACL, но и другие сущности — расширенные атрибуты, ресурсную ветвь и т. д. (см. гл. 11).

Есть три основных подхода, используемых для сокращения ACL:

- использование прав по умолчанию;
- группирование пользователей и/или объектов;
- ограничение комбинаций прав, которыми пользователи и группы могут реально обладать.

Права по умолчанию дают наибольший эффект тогда, когда большая часть прав большинства пользователей на большинство объектов одинакова. Чаще всего рекомендуют при установлении прав исходить из принципа "запрещено все, что не разрешено [явным образом]" — при последовательном его применении должно получаться, что большинство пользователей не имеет прав на подавляющую часть объектов в системе. Исходя из этого, в большинстве систем пользователи, явно или неявно не перечисленные в ACL объекта, не имеют на объект никаких прав. Многие системы также предоставляют специальную запись в ACL, соответствующую пользователям, которые не перечислены явно.

Объединение пользователей в группы представляет собой более универсальное средство, которое полезно не только для сокращения ACL, но и для других целей. Группа вводит дополнительный уровень косвенности (ACL ссылается на пользователя не прямо, а косвенно, через группу в систему установления прав и управления ими, и это дает дополнительную гибкость, полезную во многих практических случаях) (рис. 13.3).

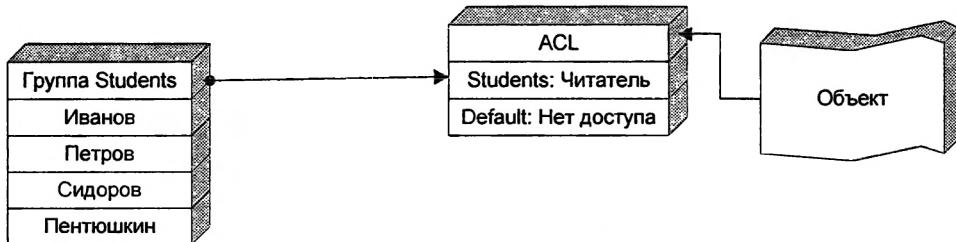


Рис. 13.3. Группы пользователей

Так, в большинстве случаев человека наделяют правами доступа к вычислительной системе не за то, что он такой хороший, а потому, что они нужны ему для выполнения служебных обязанностей. Изменение служебных обязанностей (постоянное, при переходе на другую должность, или временное, например, при замене заболевшего или ушедшего в отпуск сотрудника) сопровождается изменением прав, которыми пользователь должен обладать.

В этой ситуации целесообразно создать группу, соответствующую той или иной должности или организационной роли, и выдавать права на объекты этой группы. Назначение человека на должность сопровождается включением его в соответствующую группу, а снятие — исключением из нее. Без использования групп эти операции потребовали бы явной модификации ACL всех объектов, права на которые изменяются, что во многих случаях совершенно нерационально.

Группы являются практически обязательным элементом систем управления правами на основе списков. Большинство систем предоставляет возможность создания вложенных групп (рис. 13.4). Ряд современных систем (Novell Netware 4.x, Windows 2000, Solaris 6) предоставляют также иерархические структуры БД учетных записей, почему-то называемые *службами каталогов* (directory service). Эти службы называются NDS — Netware Directory Service в Netware, Active Directory в Windows, NIS+ — Network Information Service+ в Solaris. Впервые служба каталогов была реализована в сетевой ОС VINES (Virtual NEtwork Services) фирмы Bayyan Systems в конце 80-х годов XX века.

Иерархическая структура особенно удобна для больших организаций, потому что она не только обеспечивает "естественную" структуру вложенных групп, но и облегчает просмотр учетных записей и управление ими.

Иерархические БД учетных записей, главным образом, используются в распределенных системах, разделяемых между несколькими компьютерами, а часто контролирующих доступ и аутентификацию во всей корпоративной сети [Иртегов 2004].

При активном использовании групп пользователей может возникнуть специфическая проблема: пользователь может состоять в нескольких группах и

иметь собственную запись в ACL и, таким образом, получать права на объект несколькими путями (рис. 13.5). Строго говоря, проблемой это не является, важно лишь описать, что будет происходить с правами в этом случае.

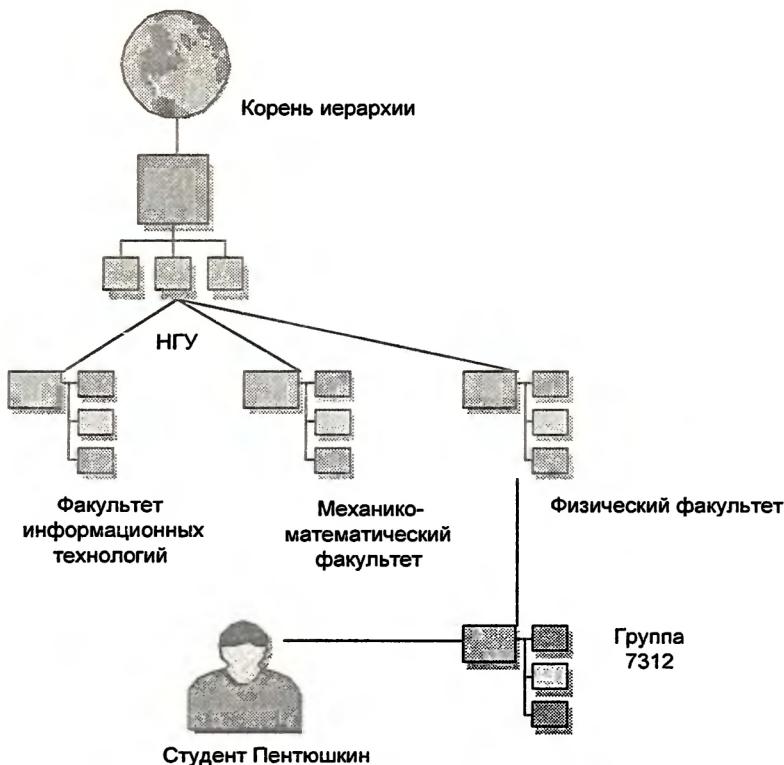


Рис. 13.4. Вложенные группы и структура организации

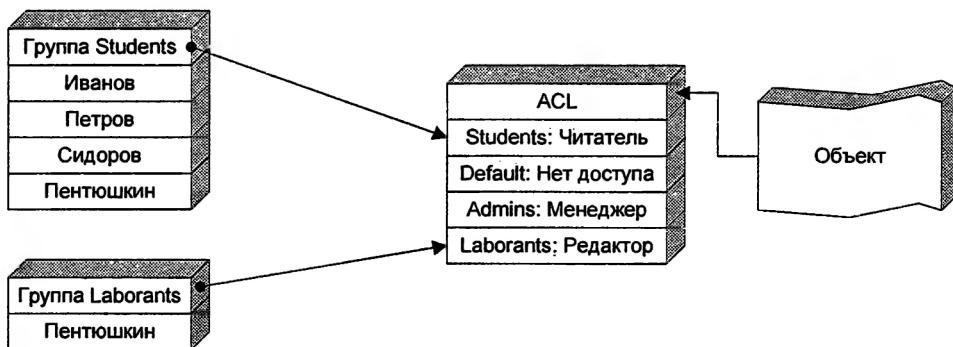


Рис. 13.5. Получение прав из нескольких групп

В различных системах используются почти все мыслимые варианты поведения.

Чаще всего права, полученные разными путями, просто складываются. Бывают системы, в которых отдельные права тем или иным образом ранжируются, и пользователь получает список прав, соответствующий той записи в ACL, которая содержит наивысшее право. Очень часто некоторые записи в ACL обладают особым статусом. Так, если человек имеет явную (соответствующую его пользовательскому идентификатору) запись в ACL, записанные в ней права оказываются "сильнее" всех прав, которые он получает как член группы. Запись с правами по умолчанию часто рассматривается как более "слабая", чем явные и групповые записи, и при наличии у пользователя прав, полученных из записей по умолчанию, вообще игнорируется.

Каждое из этих правил по отдельности обычно преследует цель облегчить формирование списков, предоставляющих требуемые комбинации прав, но в результате полное описание семантики ACL многих распространенных ОС напоминает вынесенные в эпиграф фрагменты правил игры в "драконий покер".

Группирование объектов используется несколько реже, но также является мощным средством управления правами и сокращения общего объема ACL в системе. Для файловых систем естественным средством группирования является иерархия каталогов.

Наследование прав на файлы в Novell Netware

По-видимому, наибольшей сложности группирование объектов достигло в системе Novell Netware. Рассмотрим схему установления прав на файлы в этой ОС.

Запись файлового ACL в Netware представляет собой битовую маску, значения разрядов которой перечислены в табл. 13.1. Видно, что некоторые из прав имеют смысл только для файлов, а некоторые — только для каталогов.

Каталоги и файлы в этой системе наследуют права доступа от родительских каталогов. Если пользователь или группа не перечислены явно в ACL объекта, их эффективные права будут определяться записями в ACL родительских каталогов (рис. 13.6). Если пользователь перечислен в ACL родительского и дочернего каталогов, его эффективные права будут равны сумме прав, указанных в обеих записях. Таким образом, по мере спуска по дереву каталогов, эффективные права могут только возрастать.

Причем пользователь не обязан иметь права на каталог, чтобы видеть его файлы и подкаталоги. Поэтому система управления доступом Netware предполагает выдачу прав как можно ближе по дереву каталогов к тем файлам, права на которые необходимы. Права на корневые каталоги томов обычно выдаются только администратору системы.

В случае если все-таки потребуется изменить принцип расширения прав по мере спуска по дереву, каталоги и файлы, кроме ACL, имеют дополнительный атрибут, называемый *IRF* (*Inherited Rights Filter* — фильтр наследуемых прав).

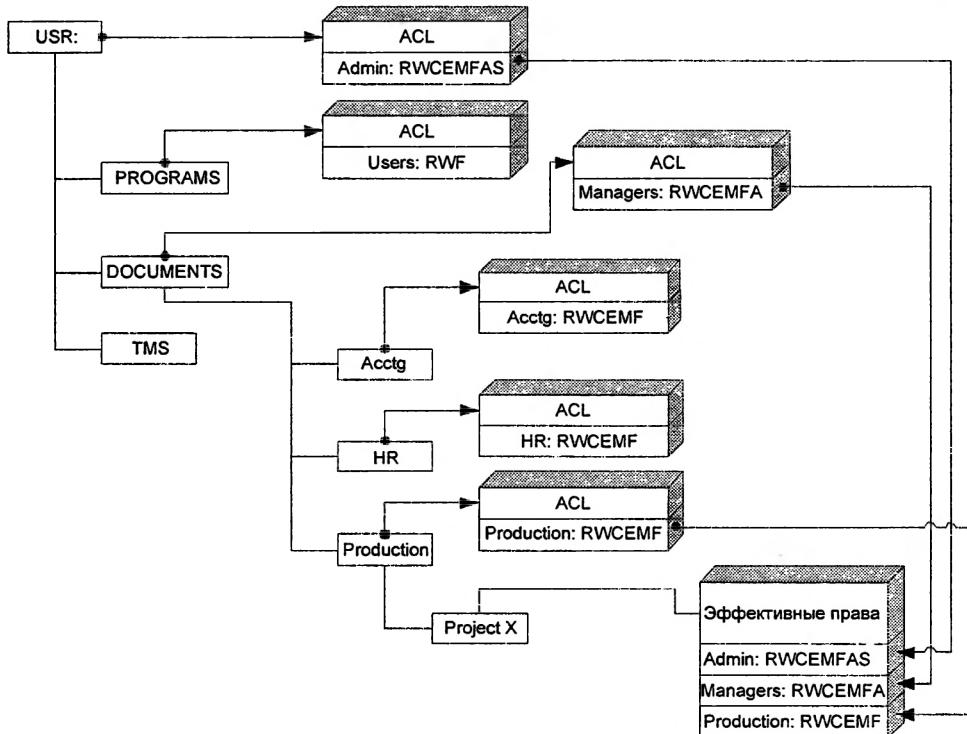


Рис. 13.6. Наследование прав на каталоги в Novell Netware
(обозначения прав соответствуют табл. 13.1)

Этот атрибут представляет собой битовую маску, биты которой (кроме бита S — он не может быть отфильтрован) соответствуют битам записи ACL (см. табл. 13.1). Установка бита в этой маске приводит к блокировке наследования соответствующего права (рис. 13.7).

Запрет на фильтрацию права супервизора обусловлен тем, что его включение по ошибке приведет к потере администратором прав на эту иерархию. Избавиться от такого поддерева можно было бы только переразметкой тома.

В пользовательской базе данных, которая, начиная с Netware 4.x, также имеет иерархическую структуру, и наследование, блокирование супервизорских прав разрешено, поэтому можно по ошибке "даровать суверенитет" ветви дерева (рис. 13.8). Это одна из распространенных ошибок начинающих администраторов. В административных утилитах Netware 4.11 даже была введена специальная проверка, не позволяющая отфильтровать право супервизора, если ни у кого нет явно выданных супервизорских прав на соответствующий контейнер или объект.

Альтернативой этим хитростям является третий из упомянутых путей — ограничение комбинаций прав, которые реально могут быть выданы. С одним

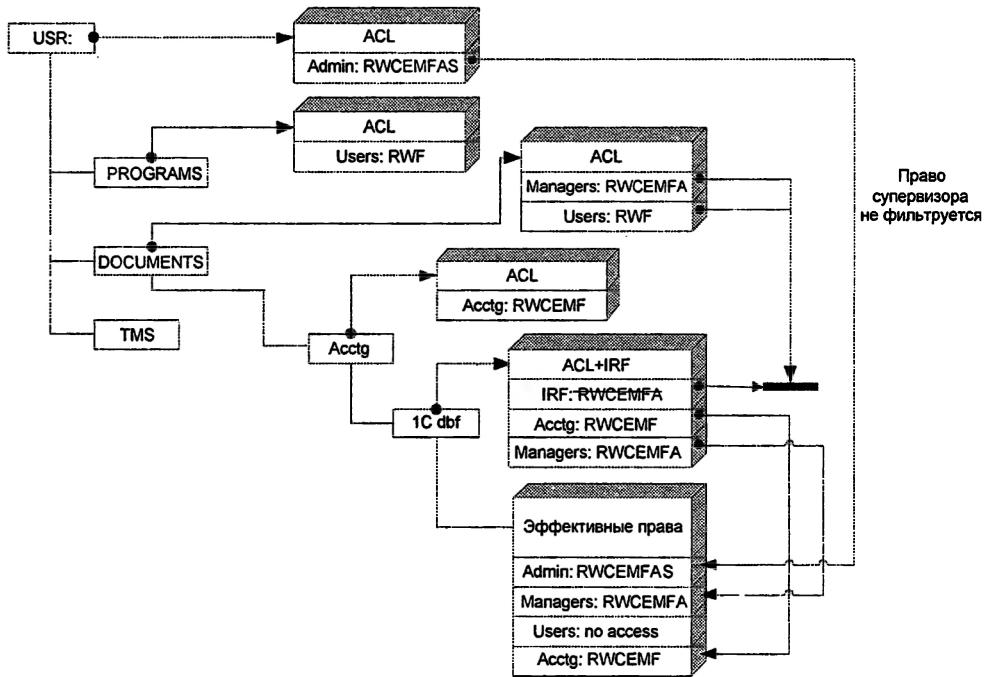


Рис. 13.7. Фильтр наследуемых прав в Novell Netware

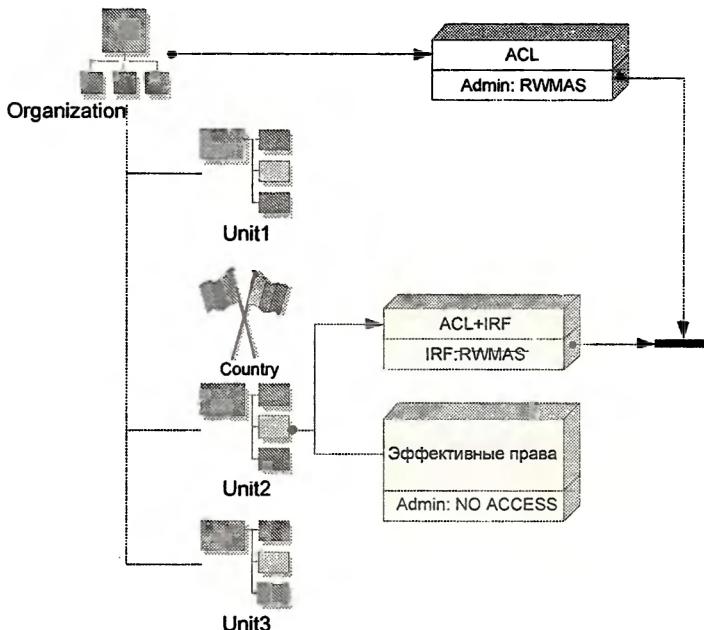


Рис. 13.8. Дарование суверенитета ветви дерева каталогов

Таблица 13.1. Права доступа к файлу в Novell Netware 3.x и выше

Бит	Обозначение	Описание
S	Supervisor	Право осуществлять любые операции над файлом или каталогом
A	Access	Право модифицировать ACL файла или каталога
R	Read	Право читать файл
C	Create	Право создавать файлы в каталоге
W	Write	Право записи в файл
E	Erase	Право удалять файл или каталог
M	Modify	Право изменять атрибуты файла или каталога
F	Find	Право на поиск файлов в каталоге

из примеров такого ограничения мы сталкивались в *разд. 5.1*: диспетчер памяти VAX имеет четыре уровня привилегий, каждый из которых может иметь право чтения и записи в страницу памяти. Все возможные комбинации прав в этих условиях кодируются восемью битами, но наложение требования о том, что каждый более высокий уровень обязан иметь хотя бы те же права, что и более низкие, позволяет нам обойтись пятнадцатью допустимыми комбинациями и четырьмя битами для их кодирования.

При разработке такой системы мы сталкиваемся с нетривиальной задачей: нам нужно выработать такие ограничения, которые не только обеспечивали бы компактное представление ACL, но и позволяли так или иначе реализовать комбинации прав, необходимые для практической эксплуатации вычислительных систем.

Замечательным примером ограниченного ACL, структура которого выдержала 30-летнюю проверку практикой, является модель безопасности в системах семейства Unix.

Авторизация в Unix\label{UnixACL}

В этих системах ACL состоит ровно из трех записей (рис. 13.9):

- права хозяина файла (пользователя);
- права группы;
- права по умолчанию.

Как правило, права хозяина выше прав группы, а права группы выше прав по умолчанию, но это не является обязательным требованием и никем специально не проверяется. Пользователь может принадлежать нескольким группам одновременно, файл всегда принадлежит только одной группе.

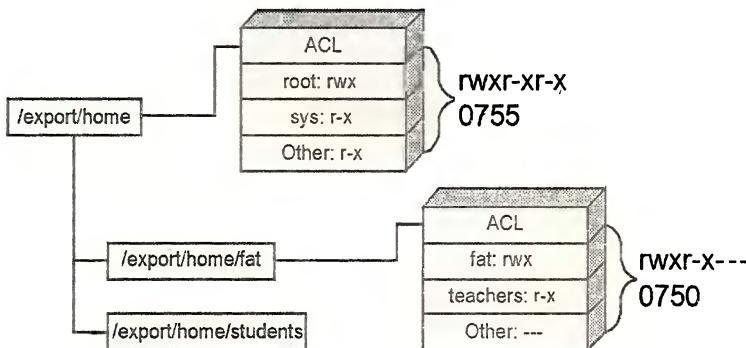


Рис. 13.9. Установление прав в системах семейства Unix

Бывают три права: чтения, записи и исполнения. Для каталога право исполнения означает право на открытие файлов в этом каталоге. Каждое из прав обозначается битом в маске прав доступа, т. е. все три группы прав представляются девятью битами или тремя восьмеричными цифрами.

Права на удаление или переименование файла не существует; вообще, в Unix не определена операция удаления файла как таковая, а существует лишь операция удаления имени `unlink`. Это связано с тем, что файл в Unix может иметь несколько имен, и собственно удаление происходит только при уничтожении последнего имени (подробнее см. главу 11). Для удаления, изменения или создания нового имени файла достаточно иметь право записи в каталог, в котором это имя содержится.

Удаление файлов из каталога, в действительности, можно в определенных пределах контролировать: установка дополнительного бита (маска прав содержит не девять, а двенадцать бит, с семантикой еще двух из них мы познакомимся в разд. 13.4.3) запрещает удаление из каталога чужих файлов. Обладатель права записи в такой каталог может создавать в нем файлы и удалять их, но только до тех пор, пока они принадлежат ему.

Кроме прав, перечисленных в маске, хозяину файла разрешается изменять права на файл: модифицировать маску прав и передавать файл другой группе и, если это необходимо, другому пользователю (в системах с дисковыми квотами передавать файлы обычно запрещают).

Еще один обладатель прав на файл, не указанный явно в его ACL, — это администратор системы, пользователь с идентификатором, равным 0. Этот пользователь традиционно имеет символическое имя `root`. Его полномочия по отношению к файлам, другим объектам и системе в целом правильнее описать даже не как обладание всеми правами, а как возможность делать с представленными в системе объектами что угодно, не обращая внимания на права.

В традиционных системах семейства Unix все глобальные объекты — внешние устройства и именованные программные каналы — являются файлами (точнее, имеют имена в файловой системе) и управление доступом к ним выполняется файловым механизмом. В современных версиях Unix адресные пространства исполняющихся процессов также доступны как файлы в специальной файловой

(или псевдофайловой, если угодно) системе proc. Файлы в этой ФС могут быть использованы, например, отладчиками для доступа к коду и данным отлаживаемой программы (рис. 13.10). Управление таким доступом также осуществляется стандартным файловым механизмом.

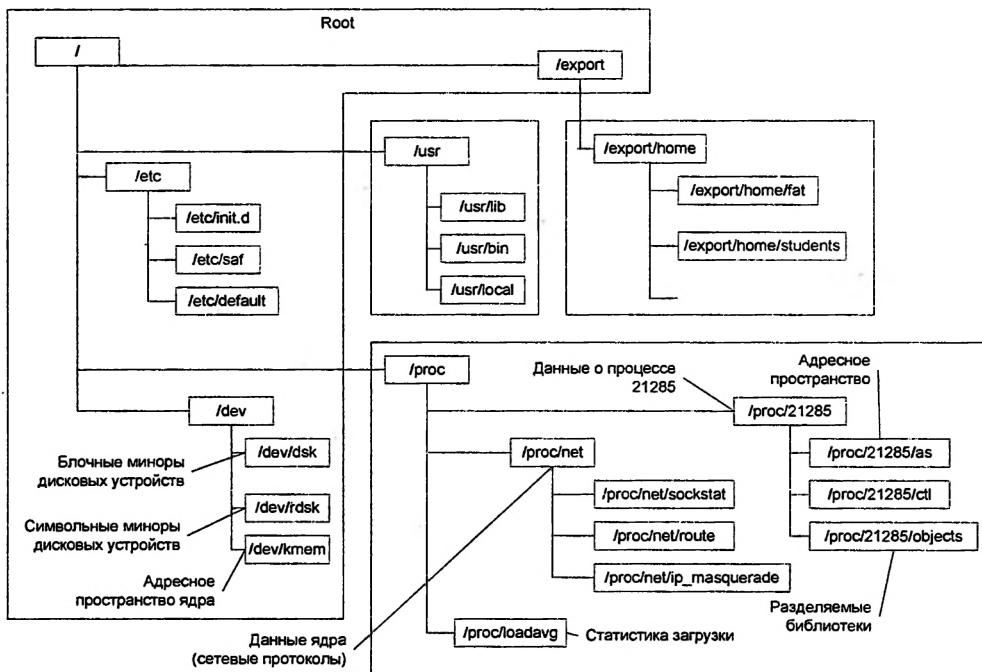


Рис. 13.10. Дерево каталогов Unix

В Unix System V появились объекты, не являющиеся файлами и идентифицируемые численными ключами доступа вместо имен, а именно средства межпроцессного взаимодействия: это семафоры, очереди сообщений и сегменты разделяемой памяти. Каждый такой объект имеет маску прав доступа, аналогичную файловой, и доступ к нему контролируется точно так же, как и файлам.

Основное преимущество этого подхода состоит в его простоте. Фактически это наиболее простая из систем привилегий, пригодная для практического применения. Иными словами, более простые и ограниченные системы установления привилегий, по-видимому, непригодны вообще.

Многолетний опыт эксплуатации систем, использующих эту модель, показывает, что она вполне адекватна подавляющему большинству реальных ситуаций. Впрочем, ряд современных файловых систем в ОС семейства Unix предоставляет произвольного вида список для управления доступом.

13.4.2. Полномочия

Ты пришла ко мне утром, ты села на кровать,
Ты спросила, есть ли у меня разрешение дышать
И действителен ли мой пропуск,
Чтобы выйти в кино.

Б. Гребенников

В чистом виде модель авторизации доступа к оперативной памяти на основе полномочий реализована в системах Burroughs и Intel 432, которые подробнее рассматриваются в *разд. 5.1 и 5.2*. Полномочия доступа к сегментам единого адресного пространства в этих системах реализованы в виде полей селектора сегмента или мандатов, а невозможность самостоятельной их генерации пользователем обеспечена теговой архитектурой и запретом арифметических операций над сегментной частью указателя.

Системы с моделями безопасности, основанные на одних лишь полномочиях, не имели коммерческого успеха. Однако концепция полномочий распространена гораздо шире, чем это принято думать. Так, реализация модели безопасности на основе ACL возможна лишь постольку, поскольку системные модули имеют полномочия делать что угодно, не обращаясь ни к каким ACL (в частности, и полномочия выполнять перечисленные в ACL операции), а пользовательский код таких полномочий не имеет. В силу этого, пользователь может выполнять необходимые ему операции лишь посредством обращения к системным модулям.

Только в этих условиях проверка ACL перед входом в системный модуль имеет смысл — если бы пользователь мог бы выполнить операцию сам или вызывать системные подпрограммы без ограничений, обход любого ACL выполнялся бы очевидным образом.

В частности, поэтому в системах с открытой памятью невозможны сколько-нибудь эффективные средства управления доступом: пользователь имеет возможность выполнять любые операции самостоятельно, а при использовании криптографической защиты может похитить ключ или модифицировать алгоритм шифрования.

Типичная практически используемая архитектура управления доступом предоставляет пользователю и системному администратору управление правами в форме списков контроля доступа и содержит один или несколько простых типов полномочий, чтобы предотвратить доступ в обход этих списков. Простейшей структурой таких полномочий является разделение пользовательского и системного режимов работы процессора и исполнение всего не пользующегося доверием кода в пользовательском режиме.

Системный режим процессора является полномочием или, во всяком случае, может применяться в качестве такового: обладание им позволяет выполнять

операции, недопустимые в пользовательском режиме, и этот режим не может произвольно устанавливаться. Он позволяет реализовать не только ACL, но и дополнительные полномочия: пользователь не имеет доступа в системное адресное пространство, поэтому система может рассматривать те или иные атрибуты дескриптора пользовательского процесса как полномочия (рис. 13.11).

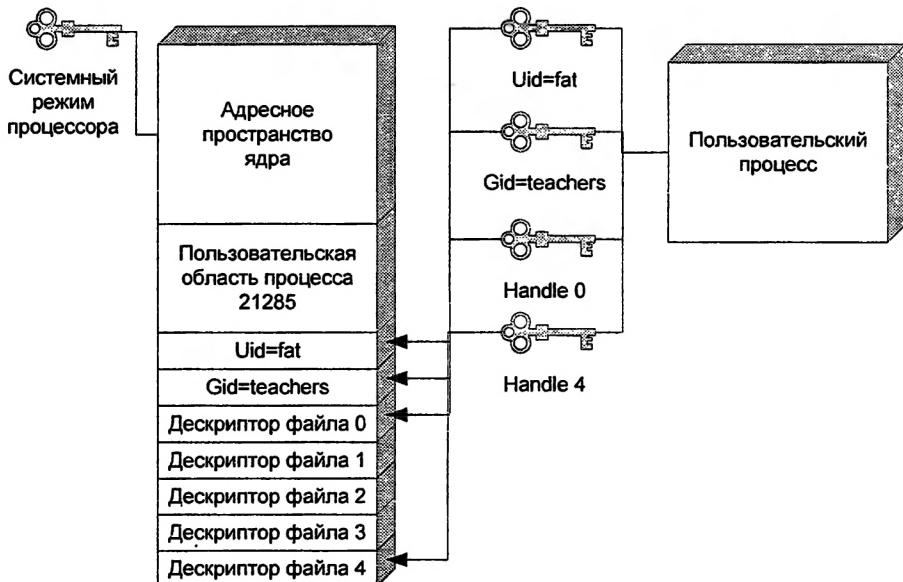


Рис. 13.11. Хранение полномочий в системном адресном пространстве

Идентификатор пользователя, устанавливаемый при аутентификации и хранящийся в дескрипторе процесса в адресном пространстве ядра, также может рассматриваться как полномочие. Для того чтобы этот идентификатор действительно можно было использовать таким образом, необходимо ввести весьма жесткие ограничения на то, кто и каким образом может производить аутентификацию.

Два подхода к решению данной задачи — это осуществление аутентификации модулями ядра и введение специального идентификатора пользователя (или специального типа процессов).

Включение средств аутентификации в ядро кажется весьма привлекательным: мы можем быть уверены, что никто не получит чужие полномочия, иначе как пройдя штатную процедуру проверки идентичности. С другой стороны, каждый процесс, считающий, что ему следует произвести смену идентичности (например, сервер, исполняющий запрос от имени конкретного

пользователя), может запросить у пользователя имя и пароль и переаутентифицироваться без каких-либо трудностей.

Проблема при таком подходе состоит в том, что мы ограничиваем используемые в системе способы аутентификации и форматы пользовательской базы данных.

Разработка модулей, реализующих альтернативные способы аутентификации (например, применяющих папиллярный детектор или сканер глазного дна вместо запроса пароля) или нестандартные способы хранения списков пользователей резко усложняются — теперь это должны быть не простые разделяемые библиотеки, а модули ядра.

Осуществление аутентификации в пользовательском режиме требует введения специального [псевдо]пользователя, которому разрешено становиться другими пользователями (говоря точнее, приобретать их идентичность), или специального типа процессов, которые могут делать это. Привилегия входить в систему и запускать процессы с таким идентификатором должна жестко контролироваться: ведь обладатель таких полномочий может стать кем угодно и, таким образом, выполнять любое действие, которое кому-либо разрешено.

Аутентификация в Unix

В системах семейства Unix процессы с эффективным пользовательским идентификатором, равным 0, имеют право выполнять системные вызовы `setuid` и `setgid` и, таким образом, становиться другими пользователями. Другие пользователи не имеют такой возможности, поэтому все процессы, в той или иной форме осуществляющие аутентификацию и смену идентичности, как с применением стандартных системных средств (файлов `/etc/passwd` и `/etc/shadow` в старых системах и разделяемых библиотек РАМ в современных), так и самостоятельно, обязаны исполняться от имени этого пользователя.

Поскольку пользователь с идентификатором 0 (`root`) может приобретать идентичность других пользователей, он, как уже говорилось, фактически обладает всеми правами, которые есть у кого бы то ни было другого в системе. Признав этот факт, разработчики Unix явным образом дали пользователю `root` вообще все права, в том числе и такие, которых не имеет никто другой.

Мы уже отмечали в разд. 13.4.1, что `root` может производить любые операции над файлами, безотносительно к тому, кому они принадлежат и какие права у них установлены. `root` также имеет возможность перезагружать систему, в ОС с динамической загрузкой модулей ядра — загружать, выгружать и перенастраивать эти модули, запускать процессы с классом планирования реального времени, посылать сигналы чужим процессам (простые смертные могут это делать только со своими процессами) и выполнять множество других операций, недоступных другим пользователям. Фактически, все действия, так или иначе затрагивающие систему в целом, являются исключительной прерогативой пользователя `root` [Хевиленд/Грей/Салама 2000].

В небольших и средних организациях это не представляет проблемы, потому что все функции, требующие привилегий `root`, — резервное копирование и

восстановление данных, регистрация пользователей и управление их полномочиями и собственно настройка системы исполняются одним человеком или командой более или менее взаимозаменяемых людей, должность которых называется системный администратор.

В крупных организациях перечисленные функции могут исполняться разными людьми: *администратором учетных записей (account manager)*, *администратором данных (data administrator)* (или администратором резервных копий) и собственно *системным администратором* (рис. 13.12).

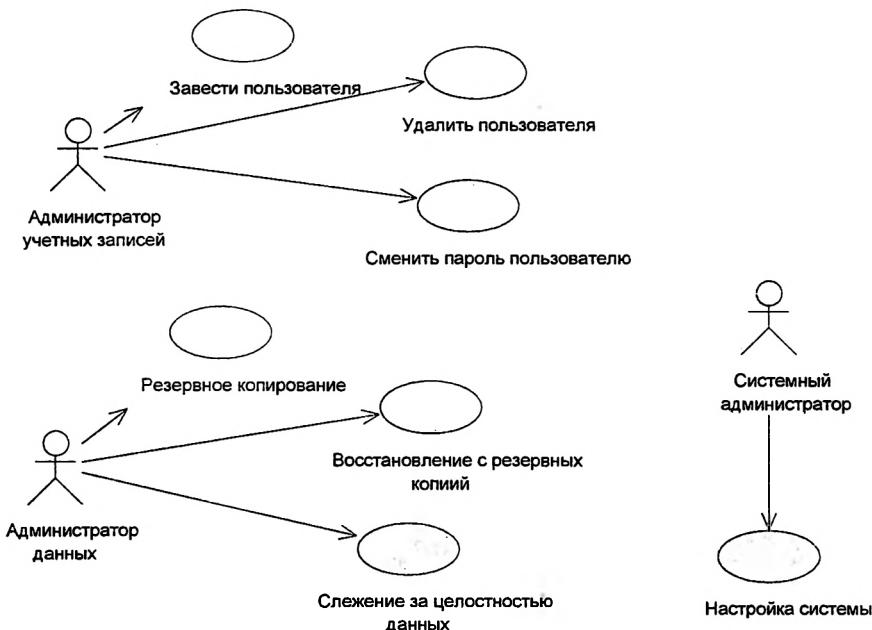


Рис. 13.12. Разделение полномочий администраторов системы

Механизм, предоставляемый системами семейства Unix, который может быть использован для реализации такого разделения полномочий, будет обсуждаться в разд. 13.4.3. Несколько забегая вперед, скажем, что решение состоит в том, чтобы разрешить администраторам учетных записей и данных запуск с полномочиями `root` только определенных программ (например, утилит управления пользователями и резервного копирования), но не запуск произвольных программ и не выполнение произвольных действий.

Многие поставщики коммерческих Unix-систем предоставляют комплектации ОС с повышенным уровнем безопасности, содержащие средства для упомянутого выше разделения полномочий между администраторами. Аутентификация для полноправного входа в систему с именем `root` в таких системах обычно предполагает ввод нескольких паролей, которые, по замыслу, должны быть известны разным людям, так что для выполнения всех действий, требующих полных административных полномочий, необходимо собирать комиссию. Как пра-

вило, такая комиссия состоит из одного или нескольких технических специалистов и одного или нескольких представителей руководства компаний.

ОС в такой комплектации применяются на центральных серверах крупных компаний, в банковских системах и других приложениях, где ведется работа с ценными и высоко конфиденциальными данными.

Многие ОС предоставляют и полномочия доступа к отдельным объектам — для того, чтобы не проверять ACL объекта при каждой операции. Так, в системах семейства Unix права доступа к файлам проверяются только в момент открытия. При открытии необходимо указать желаемый режим доступа к файлу: только чтение, только запись или чтение/запись. После этого пользователь получает "ручку" — индекс дескриптора открытого файла в системных таблицах.

Ручка представляет собой целое число и не имеет смысла в отрыве от соответствующего ей дескриптора, зато дескриптор является типичным полномочием: он недоступен пользовательскому коду непосредственно, потому что находится в системном адресном пространстве. Дескриптор может быть сформирован только системным вызовом open и допускает только те операции над файлом, которые были запрошены при открытии. Во время выполнения операций проверка прав доступа не производится (хотя, конечно, проверяется их физическая выполнимость: наличие места на устройстве и т. д.), так что если мы изменим ACL файла, это никак не повлияет на права процессов, открывших файл до этой модификации.

13.4.3. Изменение идентификатора пользователя

В некоторых случаях у разработчиков приложений или ОС возникает потребность или, во всяком случае, желание разрешить контролируемую смену идентичности для исполнения одной операции или группы тесно связанных операций над данными. Требование контролируемости означает, что нельзя разрешать пользователю свободное выполнение таких операций: например, в системе документооборота пользователю может быть разрешено поставить под документом подпись, что он с ним ознакомился, но при этом не дано права изменять содержание документа или удалять ранее поставленную подпись. Если предоставляемый системой список прав на документ либо позволяет редактирование документа целиком, либо опять-таки целиком запрещает его редактирование, то мы сталкиваемся с неразрешимой проблемой.

Авторизация доступа к полям документа в Notes

Программный комплекс Lotus Notes, широко признанный как лучшая основа для реализации систем документооборота, позволяет устанавливать права на отдельные поля или группы полей документа, а стандартная техника реализации вышеприведенных требований состоит в том, чтобы сначала предоставить

пользователю право записи в поле, где должна быть подпись, а потом — после подписания — это право тут же снять. Впрочем, при реализации сложных циклов документооборота, включающих много взаимосвязанных документов, нередко возникают ситуации, нераразрешимые с помощью только этих средств [Керн/Линд 2000].

Можно привести и другой пример, более тесно связанный с темой книги: для изменения информации о пользователе необходим доступ для записи в соответствующую базу данных, но не во всю, а только в определенную запись. Вполне естественно и даже необходимо дать пользователю возможность менять пароль, не обращаясь к администратору. С другой стороны, совершенно недопустимо, чтобы один пользователь, не являясь администратором учетных записей, мог сменить пароль другому.

Одним из решений было бы хранение пароля для каждого из пользователей в отдельном файле, но это во многих отношениях неудобно. Другое решение может состоять в использовании модели "клиент-сервер" с процессом-сервером, исполняющимся с привилегиями администратора, который является единственным средством доступа к паролям. Например, в Win32 весь доступ к пользовательской базе данных осуществляется через системные вызовы, т. е. функции процесса-сервера исполняет само ядро системы.

В системах семейства Unix для этой цели был предложен оригинальный механизм, известный как *setuid* (*setting of user id* — установка [эффективного] идентификатора пользователя). По легенде, идея этого механизма возникла именно при решении задачи о том, как же пользователи смогут менять свои пароли, если их хэши будут храниться в одном файле.

Установка идентификатора пользователя в Unix

В Unix каждая задача имеет два пользовательских идентификатора: реальный и эффективный. Реальный идентификатор обычно совпадает с идентификатором пользователя, запустившего задание. Для проверки прав доступа к файлам и другим объектам, однако, используется эффективный идентификатор. При запуске обычных задач реальный и эффективный идентификаторы совпадают. Несовпадение может возникнуть при запуске программы с установленным признаком *setuid*. При этом эффективный идентификатор для задачи устанавливается равным идентификатору хозяина файла, содержащего программу.

Признак *setuid* является атрибутом файла, хранящего загрузочный модуль программы. Только хозяин может установить этот признак, таким образом передавая другим пользователям право выполнять эту программу от своего имени. При модификации файла или при передаче его другому пользователю признак *setuid* автоматически сбрасывается.

Например, программа */bin/passwd*, вызываемая для смены пароля, принадлежит пользователю *root*, и у нее установлен признак *setuid*. Любой пользователь, запустивший эту программу, получает задачу, имеющую право модификации пользовательской базы данных, — файлов */etc/passwd* и */etc/shadow*. Однако, прежде чем произвести модификацию, программа *passwd* проверяет

допустимость модификации. Например, при смене пароля она требует ввести старый пароль (рис. 13.13). Сменить пароль пользователю, который его забыл, может только root.

Другим примером setuid-программы может служить программа /bin/ps (process status — состояние процессов) в старых системах. До установления стандарта на псевдофайловую систему /proc, Unix не предоставляла штатных средств для получения статистики об исполняющихся процессах. Программа /bin/ps в таких системах анализировала виртуальную память ядра системы, доступную как файл устройства /dev/kmem, находила в ней список процессов и выводила содержащиеся в списке данные. Естественно, только привилегированная программа может осуществлять доступ к /dev/kmem.

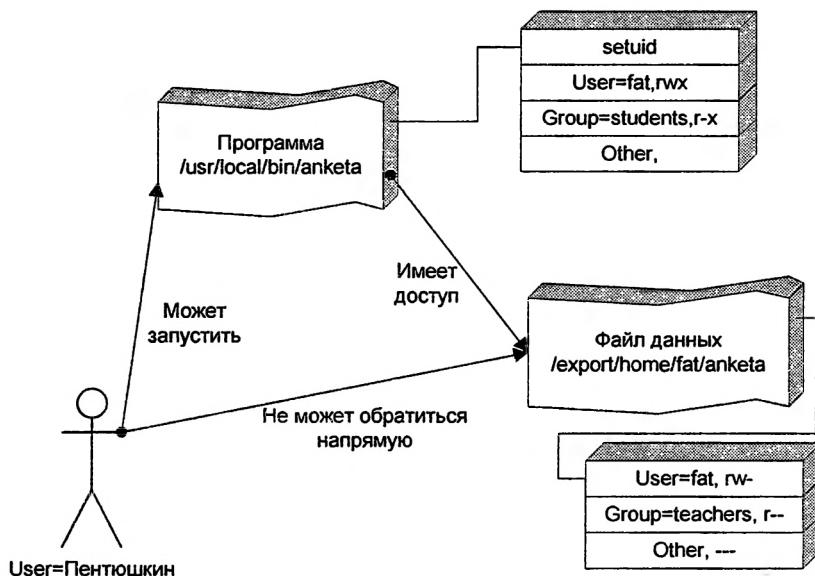


Рис. 13.13. Смена идентификатора пользователя в Unix

Механизм setuid был изобретен одним из отцов-основателей Unix, Деннисом Ритчи, и запатентован компанией AT&T в 1975 г. Через несколько месяцев после получения патента ему был дан статус public domain. Официально AT&T объяснила это тем, что плату за использование данного патента нецелесообразно делать высокой, а сбор небольшой платы с большого числа пользователей неудобен и тоже нецелесообразен.

Механизм setuid позволяет выдавать привилегии, доступные только суперпользователю, как отдельным пользователям (при этом setuid-программа должна явным образом проверять реальный идентификатор пользователя и сравнивать его с собственной базой данных), так и группам (при этом достаточно передать setuid-программу соответствующей группе и дать ей право

исполнения на эту программу), таким образом, отчасти компенсируя недостаточную гибкость стандартной системы прав доступа и привилегий в системе Unix.

Серверные агенты Lotus Notes/Domino

До совершенства механизм setuid доведен в Lotus Notes (этот пакет относится к прикладным программам, а не операционным системам, но реализует собственные схемы аутентификации и авторизации). В этой системе все объекты базы данных, в том числе и агенты (хранимые процедуры), снабжены электронной подписью. Серверные агенты исполняются не от имени пользователя, инициировавшего операцию, а от имени того, кем агент подписан.

В данном случае подпись кода означает, что либо пользователь сам занимался разработкой агента, либо он явным образом согласился взять на себя ответственность за результаты его работы. Подпись подделать практически невозможно (используется алгоритм RSA с 631-битным ключом) [redbooks.ibm.com sg245341].

sudo в современных Unix-системах

В современных системах семейства Unix широкое распространение получил механизм делегирования полномочий sudo, в конечном итоге основанный на механизме setuid. Программа sudo принадлежит пользователю root и имеет установленный бит setuid, так что она исполняется с эффективными правами root. Благодаря этому же обстоятельству, sudo может сменить идентичность и исполнить команду от имени любой другой учетной записи в системе.

Конфигурация утилиты sudo размещается в файле /etc/sudoers. В этом файле перечислены пользователи и группы пользователей, обладающие особыми правами в системе. При этом могут использоваться как общесистемные группы, перечисленные в /etc/group, так и логические группы sudo, состав которых также описывается в /etc/sudoers.

Для каждой группы указано, какими именно правами члены этой группы обладают, в виде списка команд, которые они могут выполнять, и учетной записи, от имени которой эти команды будут запущены. Команды можно указывать вместе с параметрами; в именах команд и параметров можно использовать шаблоны. Наивысший уровень полномочий допускает запуск командного интерпретатора и исполнение произвольных команд от имени root.

Файл /etc/sudoers обладает довольно сложным синтаксисом, который позволяет задавать сложные комбинации прав: например, пользователь может обладать разными полномочиями в разное время (так, оператору системы можно давать права только на период его дежурства) или при входе с разных терминалов (так, пользователю, зашедшему с консоли, можно дать право монтировать и размонтировать дискеты и другие сменные носители); реализация разных прав может требовать разных механизмов аутентификации.

sudo поддерживает три механизма аутентификации: NOPASSWD, PASSWD и ROOTPW. При использовании NOPASSWD sudo проверяет только свой реальный UID. В этом случае sudo можно использовать в фоновом режиме, например из запускаемых по расписанию заданий. Однако есть опасность, что злоумышленник может реализовать полномочия пользователя, сев за его оставленный без присмотра терминал.

PASSWD — предпочтительный механизм, при котором sudo спрашивает пароль запустившего программу пользователя. В этом режиме sudo можно использовать только интерактивно, но гарантируется, что операция действительно за- прошена именно тем пользователем, которому даны полномочия. sudo хранит историю аутентификаций и, как правило, в течении нескольких минут после успешной аутентификации не переспрашивает пароль, так что пользователь может исполнить серию команд, не вводя пароль каждый раз.

ROOTPW предполагает аутентификацию с помощью пароля `root`, а не собственного пароля пользователя. Честно говоря, я не понимаю, какие преимущества дает использование sudo в таком режиме по сравнению с более примитивными средствами, такими как команда `su` или вход в систему из-под `root`. На практике этот режим используется очень редко.

Способы использования sudo разнообразны. Так, администратору БД можно дать право перегружать сервер СУБД и изменять его конфигурацию; оператору можно дать право управлять очередями печати, монтировать и размонтировать запоминающие устройства со сменными носителями, "убивать" сессии пользователей и выключать систему и т. д.

В простейшем случае, sudo используется для того, чтобы дать некоторым людям полные административные права, не сообщая каждому из них пароль `root` — это приблизительно соответствует добавлению пользователя в группу Local Administrators в Windows NT. Даже в этом режиме sudo дает большие преимущества по сравнению с разделяемым паролем `root`: резко упрощается управление паролем супервизора (можно вообще запретить прямые регистраций в системе из-под `root`) для каждого действия, совершенного от имени `root`, остается информация в логах о том, кто именно из администраторов совершил это действие, так что значительно упрощается расследование инцидентов, связанных с ошибками администрирования.

Механизмы, функционально аналогичные sudo, предоставляются также многими пользовательскими графическими оболочками — KDE, Gnome, Mac OS X и др.

User Access Control в Windows Vista

В Windows NT механизмы контролируемой смены идентичности, аналогичные setuid, отсутствуют. Для смены идентичности процесс обязан предъявлять ядру пароль или другие аутентификационные данные учетной записи, от имени которой будет осуществляться доступ. В поставку Windows входит программа `runas.exe`, которую можно рассматривать как аналог программы `su` — она требует интерактивного ввода пароля администратора. Аналоги программы sudo, осуществляющей привилегированный доступ на основе других механизмов авторизации, в Win32 невозможны. Использование программы `runas` сопряжено со значительными неудобствами; так, запущенная из-под `runas` программа теряет сетевые соединения и работает с профилем другого пользователя. Хотя Microsoft и предлагал `runas` как средство ограничить доступ пользователя при нормальной работе, большинство пользователей это решение не устраивает и они продолжают работать с системой из-под локального администратора.

В Windows Vista была предпринята попытка решить многочисленные проблемы, связанные с постоянной работой пользователя с правами локального админи-

стратора. Эта попытка была включена в релиз ОС под названием UAC (User Access Control — управление доступом пользователя). Пользователь в Vista по умолчанию лишается прав локального администратора. Программы для Windows Vista имеют флаг в заголовке, который указывает, с какими правами программа хотела бы исполняться. Если эти права включают в себя права локального администратора, при запуске такой программы система обращается к специальному сервисному процессу, AIS (Application Information Service — служба информации о приложениях). Этот сервисный процесс работает из-под учетной записи SYSTEM, права которой эквивалентны правам локального администратора. Он выдает пользователю модальный диалог, запрашивающий подтверждение на выполнение программы с административными правами, а в некоторых случаях и административный пароль. Если подтверждение дано, сервис запускает программу от своего имени.

Причины, по которым Microsoft не включил в Vista аналог setuid и sudo, мне не вполне понятны. Патент AT&T на setuid имеет статус public domain, поэтому не может быть препятствием. В некоторых публикациях Microsoft встречается агрессивная (и обычно ошибочная) критика концепции setuid, из чего можно сделать вывод, что кто-то из влиятельных технических специалистов просто неверно понимает эту идею. Не исключено также, что переделка ядра Windows и добавление к дескрипторам процессов второго (эффективного) контекста доступа не по силам нынешнему поколению разработчиков Microsoft.

Для старых программ, требующих административных привилегий, но не помеченных как таковые, предлагается ряд более или менее неудачных решений. В частности, Vista использует ряд эвристических приемов для определения инсталляторов приложений и считает такие программы требующими административных полномочий.

Теоретически, UAC дает пользователю контроль над тем, что и как исполняется с правами локального администратора, и должен был бы пресечь большинство каналов внедрения malware в систему. На практике, диалоги UAC выдаются слишком часто, в том числе при попытках исполнения операций, вроде бы совершенно безобидных, например при попытке открыть диалог изменения даты/времени для того, чтобы посмотреть календарь. Кроме того, важно понимать, что диалог UAC выдается до запуска программы; поэтому он содержит имя программы и информацию о том, кем она подписана (если подписана), но не информацию о том, что эта программа будет делать и почему именно она хочет получить административные привилегии. То есть в этом диалоге не содержится самой важной для принятия решения информации. Поэтому в опросах пользователей UAC занимает одно из первых мест в рейтингах самых раздражающих программ, обычно наравне или с небольшим отрывом после "скрепки" Microsoft Office. По-видимому, это мнение разделяется и многими сотрудниками Microsoft, потому что в системе есть не очень хорошо документированные, но вполне штатные средства "заткнуть" UAC.

К тому же, опыт общения с приложениями Microsoft научил большинство пользователей нажимать OK (а если это возможно, то и "больше не показывать этот диалог") во всех модальных диалогах, не читая сообщение в этом диалоге и не вдумываясь в него. Так или иначе, опыт эксплуатации Windows Vista на время выхода второго издания книги не позволяет судить об эффективности UAC, но общие соображения заставляют предположить, что она будет довольно низка.

13.5. Ресурсные квоты

Мне больно видеть белый свет,
Мне лучше в полной темноте.
Я очень много-много лет
Мечтаю только о сне.

А. Князев

Все ресурсы, управление которыми осуществляет операционная система, — дисковое пространство, оперативная память, время центрального процессора, пропускная способность внешних соединений и т. д. — с одной стороны, конечно и, таким образом, исчерпаемы, а с другой — стоят денег.

Необоснованное расходование того или иного ресурса пользователем может привести к его исчерпанию и лишению других пользователей системы доступа к нему. Если ресурс необходим для работы системы, в результате последняя станет неработоспособной. Такое расходование может происходить как просто по ошибке, так и в рамках целенаправленной атаки на вычислительную систему. Промежуточное положение между этими случаями занимает использование корпоративных вычислительных ресурсов в личных целях: даже если политика компании и допускает это, те или иные пределы такому использованию необходимо установить именно в силу ограниченности ресурса.

Стандартным способом предотвратить исчерпание ресурсов в масштабах системы является введение *квот* (*quota*) на эти ресурсы: ограничений количества ресурса, которое конкретный пользователь может занять. Иногда квоты выделяются не отдельным пользователям, а группам, но — из-за возникающей при этом проблемы разрешения ресурсных конфликтов между пользователями группы — это делается очень редко. Практически всегда квота выделяется отдельному пользователю.

Чаще всего квотированию подвергается дисковое пространство. В системах семейства Unix, в которых большинство файловых систем используют статические таблицы инодов, квотированию подлежит также количество файлов, которые могут принадлежать данному пользователю.

В системах коллективного пользования квоты устанавливаются практически на все ресурсы: объем адресного пространства и рабочего множества страниц отдельной пользовательской задачи, количество одновременно запущенных процессов, время исполнения отдельной задачи или суммарное занятое время центрального процессора и т. д. Объясняется это тем, что любой ресурс, который пользователь может занимать неограниченно, представляет собой потенциальную точку атаки на безопасность системы.

На практике квоты многих ресурсов могут устанавливаться весьма высокими, так что пользователь при разумном применении системы может никогда с ними не столкнуться.

Особую роль квоты играют в вычислительных системах, совместно используемых несколькими организациями, — в этом случае квоты ресурсов, выделяемых пользователям соответствующих организаций, могут быть пропорциональны сумме денег, которые организация вносит за пользование системой. Предельным случаем систем такого рода являются *хостинг-провайдеры* (*hosting* — глагол от англ. *host* — хозяин, т. е. дословно "бытие хозяином" — в смысле "приглашение гостей"), которые предоставляют ресурсы (внешние каналы и телефонные линии, дисковое пространство для почтовых ящиков и домашних страниц, ресурсы для исполнения более сложных веб-приложений — CGI-скриптов, интерфейсов к СУБД и самих этих СУБД и др.) мелким и средним организациям и индивидуальным пользователям. Организованные по тем же принципам *ASP* (*Application Service Providers*, поставщики услуг приложений) на время написания этой книги не имели большого успеха, однако есть много оснований предполагать значительное развитие этого бизнеса в будущем.

В разд. 13.1 я уже оговаривал, что вопросы физической защиты информационной системы и защиты от "социально-инженерных" атак в этой книге затрагиваться не будут — фактически, мы ограничили постановку задачи предположением, что эти вопросы решены и от воздействий данных типов обеспечена удовлетворительная защита.

13.6. Типичные уязвимые места

— Ты намерен потопить корабль? — уточнил он.
На лице Смерти отразился ужас.

— РАЗУМЕЕТСЯ, НЕТ. БУДЕТ ИМЕТЬ МЕСТО СОЧЕТАНИЕ НЕУМЕЛОГО УПРАВЛЕНИЯ КОРАБЛЕМ, НИЗКОГО УРОВНЯ ВОДЫ И ПРОТИВНОГО ВЕТРА.

T. Пратчетт

Современные системы общего назначения имеют развитую систему безопасности, основанную на сочетании ACL и полномочий. Методы получения и передачи полномочий в этих ОС, как правило, теоретически корректны в том смысле, что доказана невозможность установления полномочия для пользователя или процесса, который не должен этого полномочия получать (сказанное не относится к системам Windows 95/98/ME, в которых эффективные средства безопасности отсутствуют по проекту).

В таких системах существует пять основных источников проблем безопасности.

- Недостаточная аккуратность пользователей в выборе паролей, создающая условия для успешной словарной атаки, а также утечки паролей, обусловленные другими причинами, начиная от пресловутых "паролей на бумаж-

ках" и кончая шантажом пользователей и прослушиванием сети злоумышленниками.

- Запуск пользователями вирусов и других троянских программ.
- Ошибки администратора в формировании ACL (с некоторой натяжкой сюда же можно причислить неудачные комбинации принятых в системе прав по умолчанию).
- Наличие в сети ОС и приложений с неадекватными средствами обеспечения безопасности.
- Ошибки в модулях самой ОС и работающих под ее управлением приложениях.

Первый источник проблем преодолим только организационными мерами, и лишь некоторые из них — например, проведение с пользователями воспитательной работы — находятся в сфере компетенции системного администратора. Исключение составляет борьба с прослушиванием сети: предотвращение технической возможности несанкционированного подключения к сети также обычно находится на грани области компетенции системного администратора, но использование шифрованных сетевых протоколов или хотя бы таких, в которых имя и пароль передаются в хэшированном виде, может значительно уменьшить пользу прослушивания для злоумышленника.

Второй источник также следует преодолевать организационными мерами. Разумной политикой, по-видимому, следует считать не полный запрет компьютерных игр, а постановку процесса под контроль путем создания легального более или менее централизованного хранилища этих игр, систематически проверяемого на предмет "заразы". Поддержание этого хранилища может выполняться как самим системным администратором, так и на общественных началах.

Третий источник проблем находится преимущественно в голове самого системного администратора. Он должен знать точную семантику записей в ACL используемой ОС, исключения из правил, хотя бы наиболее распространенные стандартные ошибки, а также то, какие и кому права даются по умолчанию.

Большую помощь в формировании оптимальных и безошибочных ACL и групп оказывает хорошо документированная организационная структура компании, из которой ясно, какие служебные обязанности требуют того или иного доступа к тем или иным ресурсам и почему и на основании каких распоряжений тот или иной уровень доступа должен быть изменен. Сверх этого можно надеяться только на аккуратность и привычку к систематической мыслительной деятельности.

Ни в коем случае не следует полагаться на то, что права, раздаваемые системой или прикладным пакетом по умолчанию, адекватны и могут быть остав-

лены без изменения. Так, в Windows NT/2000/XP на разделяемый дисковый ресурс по умолчанию даются права Everyone: Full Control (т. е. всем пользователям, явно не перечисленным в ACL, даются все права, в том числе и на изменение прав).

Четвертый источник, как правило, находится вне контроля системного администратора: хотя он и может иметь право голоса в решении вопроса о судьбе таких систем, но обычно его голос не оказывается решающим. Если отсутствие или неадекватность средств безопасности в операционной системе настольного компьютера часто можно скомпенсировать, исключив хранение на нем чувствительных данных, не запуская на нем доступных извне сервисов и — в пределе — сведя его к роли малоинтеллектуального конечного устройства распределенной системы, а его локальный диск — к роли кэша программ и второстепенных данных, то с приложениями все гораздо хуже.

Используемые в организации приложения и, что самое главное, стиль их использования обычно обусловлены ее бизнес-процессом, поэтому недостаточно продуманные попытки не только миграции в другое приложение, а иногда и установка "заплат" или изменения настроек могут привести к нарушениям в бизнес-процессе. Тщательное же продумывание и аккуратная миграция представляют собой сложный, весьма дорогостоящий и все-таки рискованный процесс, на который руководство организации всегда — по очевидным причинам — идет крайне неохотно. Это в равной мере относится как к мелким конторам с "документооборотом" на основе MS Office, так и к организациям, в которых основное бизнес-приложение представляет собой самостоятельно разрабатываемый и поддерживаемый программно-аппаратный комплекс, сохраняющий преемственность по данным с самим собой с конца XIX века (механизированные системы обработки данных, табуляторы Холлерита, появились уже тогда).

Не менее чудовищна модель безопасности приложений пакета Microsoft Office, в которых документ может содержать макропрограммы, в том числе и способные модифицировать файлы, не имеющие отношения к документу. Указанные приложения являются идеальной средой для распространения вирусов и других троянских программ. Антивирусные пакеты ни в коем случае не могут считаться адекватной мерой, потому что обнаруживают только известные вирусы, обнаружить же новый вирус или троянскую программу, написанную специально для доступа к данным вашей компании, они принципиально не способны. Я не в состоянии предложить эффективного способа действий при использовании в компании этих приложений и могу лишь посоветовать все-таки принудительно выключить макросы в приложениях Office и научить пользователей не открывать незнакомые файлы, пришедшие по почте.

Наконец, пятая причина — ошибки в модулях ОС и приложениях — хотя и находится за пределами непосредственной сферы влияния системного администратора, но заслуживает более подробного обсуждения, особенно потому, что мы предназначаем нашу книгу не только эксплуатационщикам, но и разработчикам программного обеспечения.

13.7. Ошибки программирования

Исчерпывающая классификация всех встречающихся ошибок программирования, разумеется, невозможна. Однако некоторые ошибки достаточно опасны и в то же время достаточно распространены, чтобы обсудить их отдельно.

13.7.1. Срыв буфера

Максим достал из пачки папиросу
И сильно дунул в ейное нутро.
Могучий муж не поспился на усилие
И выдул весь табак из ейного нутра.

В. Шинкарев

Одна из наиболее опасных — и в то же время довольно распространенная в современных программах — ошибка приведена в примере 2.6. Рассмотрим этот код еще раз (пример 13.1).

Пример 13.1. Пример программы, подверженной срыву стека

```
/* Фрагмент примитивной реализации сервера SMTP (RFC 0822) */
int parse_line(FILE * socket)
{
    /* Согласно RFC 0822, команда имеет длину не более 4 байт,
     а вся строка — не более 255 байт */
    char cmd[5], args[255];
    fscanf(socket, "%s %s\n", cmd, args);
    /* Остаток программы нас не интересует */
```

Видно, что наша программа считывает из сетевого соединения строку, которая должна состоять из двух полей, разделенных пробелом, и заканчиваться символом перевода строки. В соответствии со спецификациями протокола SMTP, первое поле (команда) не может превышать четырех символов (к сожалению, не определено в протоколе более длинных команд), а строка целиком не может быть длиннее 255 байт.

Если наш партнер на другом конце соединения полностью соответствует требованиям [RFC 0822], наш код будет работать без проблем. Проблемы —

причем серьезнейшие — возникнут, если нам передадут строку, которая этим требованиям не соответствует.

Превышение допустимой длины кодом команды не представляет большой опасности: лишние байты будут записаны в начало массива args и потеряны при записи в него его собственного поля. Настоящая опасность — это превышение длины всей строки. Для нашего партнера не представляет никаких сложностей сгенерировать последовательность из более чем 255 символов, не содержащую переводов строки (рис. 13.14).

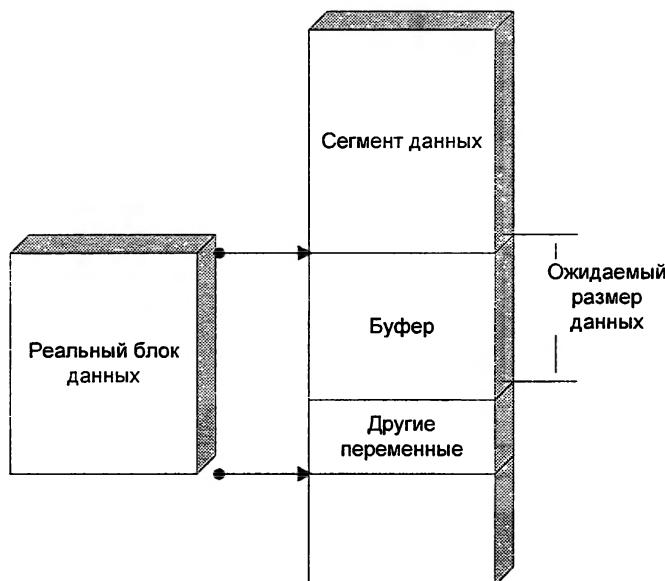


Рис. 13.14. Срыв буфера

Ошибки такого рода называются *переполнениями буфера* (*buffer overrun*) или *срывами буфера*.

Примечание

Обычно эти термины применяются для описания приемов, которые используются диверсантами для причинения ущерба системам, но важно понимать, что атаки такого типа основаны на ошибке в коде атакуемой системы. Если бы ошибки не было, диверсант не был бы в состоянии что-либо сделать (или, во всяком случае, был бы не в состоянии сделать именно это).

В зависимости от размещения переполняемого буфера, срывы буфера подразделяют на *срывы стека* (наш пример относится именно к этому классу), *срывы пула* или *срывы статического буфера*. Срывы стека наиболее опасны, потому что в большинстве современных систем программирования они обес-

печивают злоумышленнику возможность исполнить произвольный код в контексте процесса, который исполняет ошибочную программу.

Действительно, в нашем примере за буфером `arg` в памяти следует, ни много, ни мало, заголовок стекового кадра, в котором содержится адрес возврата нашей подпрограммы. Важно подчеркнуть, впрочем, что даже если бы там и находились другие переменные, это не было бы для нашего диверсанта препятствием — ему достаточно просто передать более длинный блок данных, чтобы добраться до заголовка кадра.

Если диверсант достаточно квалифицирован, он может передать нам вместо команды кусок кода и поддельный стековый кадр, который в качестве адреса возврата содержит адрес переданного кода (конечно, для этого надо точно знать, где у нашей программы находится стек, но в большинстве систем программирования это одно и то же или легко предсказуемое место). При попытке возвратить управление наша программа передаст управление на подставленный ей код (рис. 13.15). Количество гадостей, которые этот код может содержать, превосходит всякое воображение.

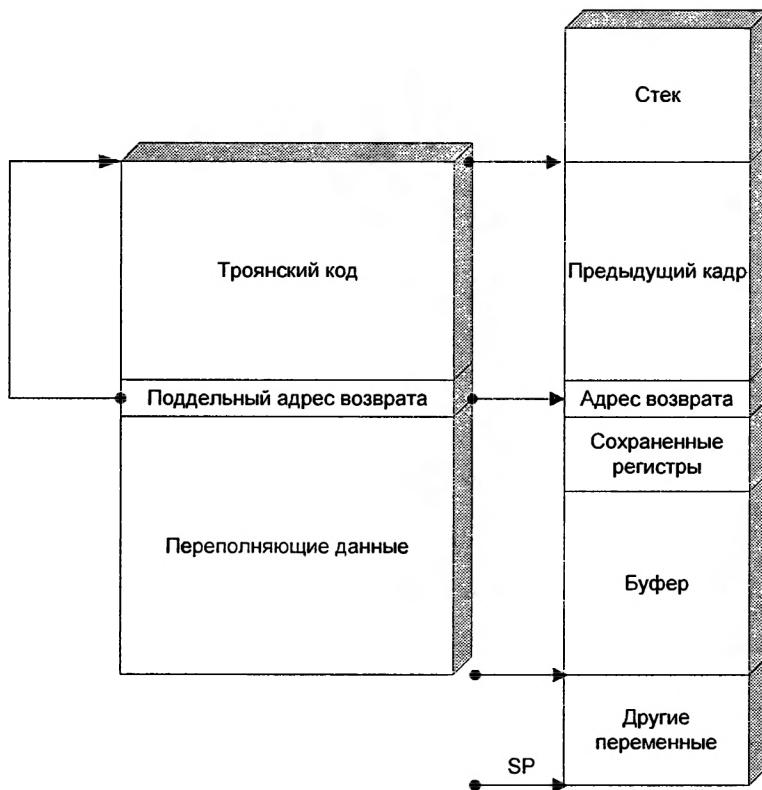


Рис. 13.15. Срыв стека с передачей троянского кода

При этом нужно отдавать себе отчет в том, что начальный переданный код не обязан быть таким уж большим. Как мы видели в главе 4, загрузочного сектора объемом не более 512 байт достаточно, чтобы загрузить целую операционную систему. Так и в нашем случае, нескольких сотен или даже десятков байт кода достаточно, чтобы открыть сетевое соединение с компьютером взломщика и затащить в нашу систему тело червя, бота или другого троянского агента.

Даже если вредитель не может передать и исполнить код (например, потому, что не знает адреса стека или потому, что стек защищен от исполнения), порчи стекового кадра достаточно, чтобы аварийно завершить исполнение сетевого сервиса, а это тоже неприятно.

Популярен миф, что срывы стека с исполнением кода опасен только для архитектур, в которых стек растет вниз, таких как x86 (направление движения стека определяется не произволом разработчика компилятора, а тем, в какую сторону сдвигает стек команда CALL, которая на всех современных процессорах сохраняет адрес точки возврата в стеке). По-видимому, источником этого мифа является статья [Karger/Shell 1974], в которой открытым текстом сказано, что срывы стека безопасны для процессоров, у которых стек растет вверх. При всем моем уважении к авторам этой статьи, я должен отметить, что это совершенно неверно.

Рассмотрим еще раз пример 2.6 Срываемый буфер в этом примере размещен в записи активации функции `parse_line`, но сам выход за границы буфера происходит в функции `fscanf`. То есть в момент переполнения в стеке находится еще один адрес возврата, и для активизации переданного кода нам достаточно перезаписать его — только и всего.

Срывы пула и статических буферов значительно менее опасны с точки зрения исполнения кода, чем срывы стека, поэтому даже существует миф, что такие срывы не могут быть использованы для внедрения и исполнения враждебного кода. Это также не совсем верно; действительно, современные хакеры предпочитают использовать срывы стека, но не потому, что через срыв статического буфера этого сделать нельзя, а лишь потому, что через срыв стека это сделать намного легче.

В действительности, в сегментах данных типичной современной программы находится немало указателей на код — таблицы виртуальных методов C++, PLT загрузчика ELF и многие другие структуры динамического связывания, а часто и просто указатели на функции. Переполнение статических или динамически размещаемых буферов позволяет — во всяком случае, теоретически — модифицировать любую из этих структур. Разумеется, взломщику необходимо проследить дальнейшую логику работы атакуемой программы — ведь ему нужно также гарантировать, что по модифицированному им указа-

телю рано или поздно произойдет передача управления, поэтому злоупотребление срывом статического буфера оказывается существенно сложнее, чем в случае срыва стека.

Если между срываемым буфером и ближайшими указателями на код расположены защищенные от записи области памяти, могут использоваться и более изощренные приемы. Например, взломщик может модифицировать указатель на данные, по которому программа вскоре будет производить запись, так, чтобы он указывал на PLT или на запись активации процедуры в стеке. Разумеется, это дополнительно усложняет атаку. Однако важно понимать, что сложность атаки в данном случае вовсе не означает ее технической невозможности.

Весьма эффективной защитой от внедрения кода с помощью срыва буфера могла бы быть защита данных от исполнения, а кода — от записи. К сожалению, диспетчеры памяти наиболее распространенных на сегодня процессоров не поддерживают защиты данных от исполнения: структура прав доступа страницы в большинстве реализаций x86 такова, что любую страницу, которую можно прочитать, можно и исполнить. Бит защиты страницы от исполнения появился только в процессорах Athlon и Pentium IV последних серий; ко времени выхода второго издания книги из всех ОС для x86 этим битом могли пользоваться только Windows XP SP2 и Windows Server 2003 SP 1.

Необходимо также отметить, что даже аппаратная защита данных от исполнения, вообще говоря, не гарантирует полной защиты от исполнения кода при срыве буфера. Имея возможность модифицировать указатели на код, взломщик может вызвать в атакуемой программе кусок кода, который делает то, что ему нужно, или сформировать своего рода "шитый код", состоящий из серии вызовов различных подпрограмм, например функций стандартной библиотеки ввода/вывода. Имея возможность модифицировать стек, взломщик может передать этим функциям нужные параметры. При этом, как уже отмечалось, для успешной атаки, вообще-то, не требуется сложный код — достаточно записать тело троянской программы в файл и тем или иным способом запустить этот файл. Годится как запуск в отдельном процессе, так и путем загрузки этого файла в качестве динамической библиотеки. В крайнем случае, можно просто подложить этот файл в такое место, где его с высокой вероятностью запустят, или зарегистрировать его в каком-либо скрипте или списке автоматического запуска.

Срывы буфера возможны не только в сетевых сервисах, но и в приложениях, просточитывающих файлы и, с другой стороны, не только в высокоуровневых сетевых сервисах, но и в драйверах сетевых протоколов нижнего уровня — последний тип ошибок особенно опасен, потому что атаке подвергается модуль ядра ОС. Особенную опасность представляет срыв буфера в модулях,

осуществляющих парольную авторизацию: в этом случае злоумышленник может даже не разрушать стековый кадр, ему достаточно лишь модифицировать переменную, которая сигнализирует, что пароль успешно проверен.

Срыв буфера в GDI+ Win32

В 2004 году в графической подсистеме Win32 был обнаружен срыв буфера в библиотеке, работавшей с графическими файлами формата JPG [MS04-028]. Впечатляет один только список продуктов, оказавшихся уязвимыми, — он занимает более страницы и включает все поддерживающиеся на то время версии Windows (ME, 2000, XP, 2004) и все без исключения популярные прикладные продукты Microsoft, в том числе Internet Explorer и все программы пакета Microsoft Office. К счастью, "заплата" также требовала обновления только одной библиотеки, поэтому установка обновления для любого из уязвимых продуктов в большинстве случаев приводит к полному закрытию "дыры". Тем не менее Microsoft была вынуждена выпустить специальную утилиту, которая сканирует всю систему в поиске уязвимых версий библиотеки.

Наиболее велика опасность срыва буфера в ситуациях, когда спецификация сетевого протокола или формата файла гласит, что длина того или иного поля или пакета не может превышать определенного количества байтов, однако нарушение этого правила физически возможно. Возможность такого нарушения может возникать как из-за того, что используется не счетчик байтов в пакете, а маркер конца пакета, так и из-за того, что разрядность счетчика байтов позволяет представлять значения, превышающие установленный протоколом предел.

При программировании на языке С основные источники ошибок такого рода — это использование стандартных процедур `gets`, `scanf/fscanf`, `sprintf` и некоторых других функций стандартной библиотеки С, прежде всего `strcpy` и `strcat`. Процедура `gets` лечению не подлежит — ей невозможно указать размер буфера, выделенного для приема данных, поэтому она в принципе не способна проконтролировать его заполнение. Вместо нее современные версии библиотек С предоставляют функцию `fgets`, которой размер буфера передается в качестве параметра. Настоятельно рекомендуется использовать именно ее.

Процедура `fscanf` в нашем случае лечится. Вместо

```
fscanf(socket, "%s %s\n", cmd, args);  
нам следует написать  
fscanf(socket, "%4s %255s\n", cmd, args);
```

Однако автоматизировать проверку того, что в каждом случае все форматные спецификаторы указаны правильно, невозможно, и поэтому использовать процедуру `fscanf` и другие процедуры того же семейства не рекомендуется.

Распространение червя Морриса через срыв буфера

Широко известен срыв буфера в программе fingerd, входившей в систему BSD Unix. На процессорах VAX и MC68000 отсутствует защита страниц памяти от исполнения, поэтому любая страница памяти, доступная для чтения, может быть исполнена.

"Червь Морриса" [КомпьютерПресс 1991] пользовался этой дырой, передавая кусок кода и поддельный стековый кадр, передававший управление этому коду. Код в свою очередь запускал на целевой системе копию командного интерпретатора, исполнявшуюся с привилегиями суперпользователя. Затем червь использовал полученный командный интерпретатор для втягивания в систему своего тела.

Ошибка была обнаружена в 1987 г. и вскоре была исправлена. Практически все современные системы используют безопасную в этом отношении версию fingerd. Впрочем, в современных системах этот сервис обычно вообще выключают, потому что спаммеры могут использовать его для получения списка почтовых адресов пользователей.

Червь Code Red, пандемия которого произошла в августе 2001 г., использовал срывы буфера в IIS (флагманский сетевой продукт Microsoft, сервер FTP/HTTP и ряда других протоколов для Windows NT/2000/XP). Как выяснилось, в данном случае речь шла не об одной, а о целой группе ошибок, потому что за выпуском patch, защищавшего от Code Red, последовало несколько других атак червей и поливалентных (т. е. использующих несколько каналов размножения) вирусов, часть из которых также привела к пандемиям. 19 сентября 2001 г. аналитическая компания Gartner Group выпустила доклад [www3.gartner.com 101034], в котором настоятельно рекомендовалось как можно скорее отказаться от использования IIS и высказывалась крайне пессимистическая оценка способности Microsoft исправить положение в обозримом будущем.

Срывы буфера являются одним из наиболее распространенных уязвимых мест: так, в базе данных [www.cert.org] они составляют 27% от общего количества уязвимостей уровня critical (наиболее высокая из категорий опасности по классификации Cert Advisories).

При использовании других языков программирования, например C++ с библиотекой классов для работы со строками или Java, реализовать программу, подверженную срыву буфера, несколько сложнее, однако возможности человеческие неисчерпаемы, и многим программистам это удается. Впрочем, для программирующих на Java есть одно утешение: Java Virtual Machine использует более сложную схему управления памятью, чем компилируемые алголоподобные языки, и разрушить стековый кадр посредством переполнения буфера в программах, написанных на Java, невозможно, поэтому данный прием не может применяться для передачи и исполнения вредоносного кода. Однако неправильно обработанное исключение при ошибке индексации в буфере может привести к аварийной остановке приложения Java с ничуть не меньшим успехом, чем ошибка доступа к памяти в C/C++.

Необходимо помнить также, что значительная часть среды исполнения таких языков, как Java, в действительности реализована на C/C++. В ядре JVM неоднократно обнаруживали срывы буфера; как и в случае с большинством других ошибок, то, что обнаруженные ошибки исправлены, вовсе не означает, что они исправлены все. Кроме того, поскольку развитие среды исполнения Java продолжается, возможно возникновение новых ошибок этого типа.

В тесном концептуальном родстве со срывами буфера находятся ошибки, срабатывающие при использовании во входном потоке данных недопустимых величин смещения (рис. 13.16). Такие ошибки встречаются при анализе входного потока, который содержит взаимосвязанные структуры данных, связи между которыми реализованы в виде смещений в потоке (я ссылаюсь на структуру данных, которая последует через 50 байт после этой точки). Практически важный пример такого протокола — система квитирования (посылки подтверждений) со скользящим окном, используемая в транспортном протоколе TCP [RFC 0793]. Адресация посредством смещений широко применяется также при работе с последовательными файлами, поэтому драйверы файловых систем и сетевые файловые серверы также могут содержать такие ошибки.

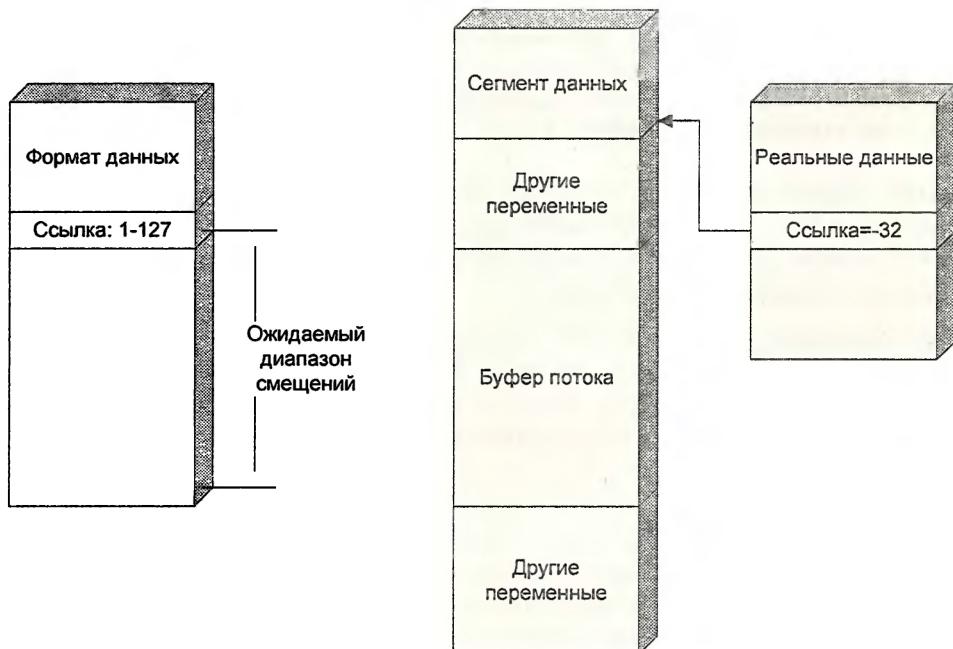


Рис. 13.16. Использование недопустимых смещений

Задание смещений, превосходящих размер буфера анализирующей программы, или недопустимых по каким-либо другим правилам — например, отрицательных, если по протоколу допустимы только положительные — может приводить к формированию указателей за пределы анализируемого буфера. Модификация данных по этим указателям может приводить к разнообразным последствиям — например, таким способом можно попытаться убедить файловый сервер, что файл, открытый для чтения, в действительности открыт для записи. Даже если в пределах досягаемости сформированного таким образом указателя и нет критически важных данных, практически всегда его можно использовать для разрушения целостности переменных состояния атакованного модуля и осуществления DoS.

13.7.2. Внедрение скриптов и SQL

В многослойных приложениях, использующих реляционные СУБД для хранения данных, часто встречается ошибка внедрения SQL (SQL insertion или SQL injection). Эта ошибка редко встречается в тиражируемых продуктах, но очень распространена в программах, разрабатываемых по индивидуальным заказам, и может быть почти столь же опасна, как срыв буфера, поэтому она также заслуживает отдельного обсуждения.

Рассмотрим типичный элемент кода трехслойного приложения, обращающегося к СУБД (пример 13.2).

Пример 13.2 Обращение к реляционной СУБД из скрипта

```
string sql= "select Id from users where name='"+username+"'";
```

Как правило, этот код ведет себя вполне разумно: при обращении к базе будет получена запись, соответствующая имени пользователя. Но что, если злоумышленник тем или иным способом передаст username, такой как в примере 13.3?

Пример 13.3 Возможное значение переменной username

```
user'; insert into users value ('hacker', 'password', 'Administrator') --
```

Строка -- в конце текста в большинстве распространенных диалектов SQL обозначает начало комментария, поэтому конец оригинального запроса исполнен не будет и не сможет привести к нарушению синтаксиса SQL.

Существует ряд способов затруднить подобные действия, но все они малоэффективны. Например, рекомендуется устанавливать соединение с СУБД от

имени учетной записи, которая не имеет права модифицировать таблицы. К сожалению, это не всегда возможно — ведь многие приложения требуют сбора вводимых пользователем данных. Кроме того, иногда считывание данных из таблиц тоже может быть ценным для взломщика, например если таблица содержит номера кредитных карт или другие конфиденциальные данные. Тем не менее при тщательном продумывании подход, основанный на ограничении доступа скрипта к БД, может быть достаточно эффективен — так, можно добиться, чтобы весь доступ к БД происходил через представления и хранимые процедуры, а прямой доступ к таблицам был бы запрещен. К сожалению, требование тщательного продумывания сводит на нет основное достоинство скриптовых языков — низкую стоимость разработки приложений.

Также, можно было бы сканировать значение `username` в поисках кавычек. В языке PHP для этого даже есть специальная синтаксическая поддержка — так называемая "магическая кавычка" (magic quote). Рассмотрим пример 13.4.

Пример 13.4 Магическая кавычка в PHP

```
$sql="SELECT Id FROM users WHERE login='$login' AND paswd='$paswd'";
```

В языке PHP, в теле строк, ограниченных двойными кавычками, производится подстановка значений переменных, так что `$login` будет заменено на значение переменной `$login` в момент вычисления оператора и т. д. При этом, поскольку значение переменной `$login` заключено в одинарные кавычки, интерпретатор PHP заменит все вхождения символа одинарной кавычки в значении переменной на последовательность `\'` (в SQL эта последовательность используется для включения символа кавычки в строковые литералы). Кроме того, большинство других специальных с точки зрения SQL символов также будут заэкранированы. Это сделано специально для упрощения формирования корректных запросов SQL.

Так или иначе, корректный анализ входных данных довольно сложен и тоже может привести к значительному росту стоимости разработки.

Более эффективное средство предоставляется рядом современных средств доступа к реляционным СУБД — параметризованные команды. При работе с такими средствами подстановка параметров в запросы происходит не до компиляции текста запроса, а во время ее. При этом компилятор уже знает, что `username` — это не произвольный кусок текста, который может содержать как значения, так и код, а значение, которое необходимо подставить в запрос с уже определенным синтаксисом [Ховард/Лебланк 2004].

Аналогичные проблемы могут возникать в других скриптовых языках, особенно в тех, которые предоставляют функцию `evaluate` (вычислить — в дей-

ствительности, исполнить параметр этой функции как код). Используя ошибки при формировании таких строк, злоумышленник может заставить приложение выполнить произвольный код. Соответственно, эти ошибки называются внедрением скрипта (*script injection*).

Например, во всех вариантах входного языка *shell* (командных процессоров систем семейства Unix) строка, ограниченная символами `` (обратная кавычка), всегда исполняется, и на ее место подставляется текст, выведенный запущенными при исполнении командами. При этом практически невозможно обратиться к строковой переменной, не спровоцировав выполнение размещенных в ней таким образом команд — невозможно даже проверить строку на наличие таких символов! Поэтому, несмотря на то, что в ряде учебников по *shell* приводятся примеры CGI-скриптов на этом языке, необходимо проявлять очень большую осторожность, давая публичный доступ к таким программам.

Червь phpBB Santy

phpBB — популярный "движок" для реализации веб-форумов [www.phpbb.com], распространяемый с открытыми исходными текстами. Осенью 2004 года в нем была найдена ошибка внедрения скрипта, которую можно было использовать как для внедрения SQL-запросов, так и для исполнения произвольного кода на языке PHP от имени веб-сервера; в том числе, командой *system* можно было выполнить произвольную команду *shell*. Практически сразу после обнаружения ошибки была выпущена обновленная версия *phpBB* 2.0.11.

Однако администраторы многих форумов оказались недостаточно расторопны и 21 декабря 2004 года уязвимые форумы были поражены сетевым червем, известным как Perl.Santy. Этот червь находил уязвимые форумы с помощью поискового сервера Google и запускал на них скрипт на языке Perl, который начинал искать и заражать другие уязвимые форумы. Поскольку весь червь был реализован на скриптовых языках, он с равным успехом поражал серверы, работающие на всех ОС и аппаратных платформах, где поддерживался *phpBB* [isc.sans.org Santy].

Чрез некоторое время после внедрения в систему, червь пытался заменить все страницы сайта (или, точнее, все файлы с расширениями .php, .asp, .htm, .shtml) на короткую HTML-страницу:

This site is defaced!!!

NeverEverNoSanity WebWorm generation x.

На момент, когда распространение червя было заблокировано (для этого администраторам Google пришлось заблокировать запрос, посредством которого червь искал жертвы), количество зараженных сайтов оценивалось приблизительно в 40 000.

13.7.3. Другие ошибки

В многопоточных сервисах распространены ошибки соревнования. Для их срабатывания необходима определенная последовательность и временнóе

согласование запросов к сервису. В большинстве случаев, такие ошибки приводят к падению сервиса и/или порче данных, но есть ряд примеров, когда такие ошибки могли использоваться для несанкционированного доступа к данным.

В некоторых типах сервисов встречаются свойственные им ошибки. Так, во многих серверах HTTP была обнаружена ошибка подъема по каталогам (directory traversal bug), когда злоумышленник, запрашивая URI, содержащие последовательности '..', мог подняться по файловой системе выше корневого каталога HTTP-сервера и, таким образом, считать или даже модифицировать файлы, не входящие в иерархию HTML-документов (рис. 13.17). Аналогичные ошибки встречаются и в сетевых файловых серверах.

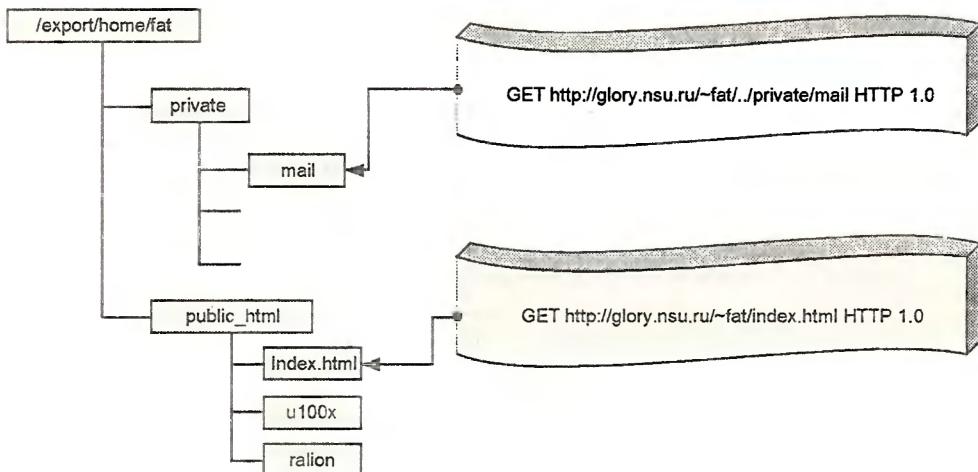


Рис. 13.17. Ошибка подъема по каталогам

Общим правилом, позволяющим, если не искоренить ошибки такого рода, то, во всяком случае, уменьшить вероятность их совершения, является недоверие к входным потокам данных. Если спецификации протокола или формата гласят, что то или иное условие обязано выполняться, мы не можем просто считать, что оно выполняется, а обязаны как минимум вставить явную проверку того, что оно выполняется. Программа тестирования программного комплекса должна включать не только проверку правильной обработки допустимых входных данных в каждом из модулей, но и осмысленную реакцию на недопустимые входные данные.

13.8. Троянские программы

Бойся Данаицев, все время дары приносящих,
Нам для даров этих вскорости места не хватит,
Им — то, Danaizam, все пофиг: несут себе, гады,
Мы же загнуться рискуем в стесненном про-
странстве.

Д. Филиппов

Название этого типа атак происходит от известной легенды о статуе коня, которую греки использовали для проникновения в стены города Троя во время воспетой Гомером Троянской войны.

Троянские программы или, короче, "трояны" (*troyan*) представляют очень большую опасность, потому что исполняются они с привилегиями тех пользователей, которые имели несчастье их запустить, и имеют доступ ко всем данным этого пользователя. Если троянскую программу удается запустить от имени системного сервиса, то автор трояна может получить полный контроль над системой.

Вред, который может причинить троянская программа, ограничен только фантазией ее разработчика. Из наиболее неприятных возможностей следует упомянуть полное уничтожение всех доступных данных или их анализ и пересылку результатов анализа автору или заказчику трояна. В некоторых случаях результат деятельности троянской программы выглядит как целенаправленная диверсия со стороны пользователя, что может повести расследование инцидента по ложному пути.

Пример троянской программы

Простой и по-своему элегантный пример троянской программы приводится во многих учебниках по командному языку систем семейства Unix, например в [Керниган/Лайк 1992]. В указанной работе эта программа приводится для объяснения того, почему в Unix путь поиска исполняемых программ по умолчанию не включает текущего каталога, и почему включать текущий каталог в этот путь крайне нежелательно.

Программа из примера 13.5 работает следующим образом:

1. Вредитель помещает ее в общедоступный каталог под именем `ls`.
2. Пользователь входит в этот каталог и исполняет команду `ls` (просмотр текущего каталога).
3. Вместо системной команды `/bin/ls` исполняется троянская программа.
4. Троянская программа совершает вредоносное действие и запускает настоящий `/bin/ls`, позаботившись о том, чтобы отфильтровать свою запись из листинга каталога.

Важная часть программы — модуль, который удаляет файл `ls` из листинга текущего каталога, — опущен из-за его сложности, ведь он должен анализировать параметры команды `ls` и производить удаление своей записи различными способами в зависимости от требуемого формата вывода команды.

Пример 13.5. Троянская программа на языке shell

```
#!/bin/sh
# Разместите эту программу в общедоступном каталоге и назовите ее ls
# Скопировать себя в домашний каталог пользователя;
# на этом месте может стоять и другая вредоносная операция
cp $0 ~
/bin/ls $# | /home/badguy/filter_ls $#
```

Троянская программа может быть реализована не только в виде самостоятельного загрузочного модуля, но и в виде разделяемой библиотеки. Так, в Windows NT 4.0 вплоть до выхода Service Pack 5 присутствовала ошибка, позволявшая зарегистрировать DLL, совпадающую по имени с любой из системных, причем так, чтобы при разрешении ссылок из других модулей использовалась вновь зарегистрированная библиотека.

Аналогичный пример для Win32

В 2001 году произошла пандемия вируса Nimda, один из приемов распространения которого аналогичен приведенному в примере 13.5: обнаружив каталог, в котором лежат файлы данных MS Office, вирус помещает туда файл riched20.dll. При сборке программы в момент запуска Windows просматривает текущий каталог до перечисленных в PATH, поэтому вместо модуля Office загружается троянский код вируса. В отличие от систем семейства Unix, в Win32 порядок просмотра каталогов при сборке жестко задан и не поддается контролю со стороны системного администратора, поэтому перекрыть данный путь распространения заразы можно лишь модификацией ядра Windows.

Многие троянские программы образуются модификацией присутствующих в системе загрузочных модулей, разделяемых библиотек и даже модулей ядра.

Такая программа не обязательно должна представлять собой бинарный код — это может быть также интерпретируемый код или последовательность команд макропроцессора. Широко распространены макровирусы, распространяющиеся с файлами данных пакета Microsoft Office.

Способы внедрения троянских программ (и программ, которые по своему воздействию трудноотличимы от троянских) в систему отличаются большим разнообразием и заслуживают отдельного подробного обсуждения. В зависимости от способа внедрения, трояны можно разделить на четыре основных класса.

- Встраиваемые в систему при ее разработке. В зависимости от намерений программиста их можно подразделить еще на три класса:

- "закладки", т. е. сознательно внедряемый вредоносный код;
 - "задние двери" (backdoor) — отладочные инструментальные средства, которые позволяют разработчику получать привилегии нештатными способами;
 - ошибки программирования.
- Добавляемые в систему при ее распространении. От этих типов троянских программ особенно сильно страдают бесплатные и условно-бесплатные программные пакеты, распространяемые по Интернету. Взломав сервер, с которого распространяется дистрибутив программы, злоумышленник может внедрить в него троянский код.
- Внедряемые в уже работающую систему путем физического доступа к этой системе, в том числе и путем установки в систему дополнительного оборудования. Воздействия такого рода относительно сложно реализуемы, потому что физической безопасности вычислительных систем обычно уделяется адекватное внимание и, как правило, они достаточно эффективно защищены от таких воздействий. Кроме того, идентификация злоумышленника при обнаружении такого воздействия обычно достаточно проста, а доказательство злого умысла бесспорно и очевидно.
- Внедряемые в уже работающую систему с помощью несанкционированного удаленного доступа. Поскольку обычно код работающей системы так или иначе защищается от модификации, для того чтобы что-то подобное могло произойти, в системе уже должны присутствовать какие-то проблемы с безопасностью.
- Внедряемые в уже работающую систему с согласия пользователя (часто при этом пользователя целенаправленно вводят в заблуждение, так что он не совсем ясно представляет себе, на что именно он согласился). Троянская программа из примера 13.5 относится к этому классу. Многие вирусы, в действительности, также относятся к этой категории.

Возможна другая классификация троянских программ — по тому, что они делают или пытаются делать. Как уже отмечалось, потенциальные действия троянской программы ограничены только фантазией ее разработчика. Поэтому исчерпывающей классификации программ по этому признаку я привести не смогу, но попробую перечислить функции наиболее распространенных типов практически применяемых программ.

- Сбор конфиденциальной информации из системы, в которую внедрен троян. Крупнотиражные трояны используются для сбора информации, которая с высокой вероятностью может присутствовать на многих компьютерах — паролей (в том числе паролей доступа к сетям интернет-провайдеров), приватных ключей и сертификатов SSL, номеров кредитных карт и

других банковских реквизитов (так называемый *phishing*), регистрационной информации для условно-бесплатных и защищенных от копирования программ. При целенаправленных атаках на конкретного человека или компанию злоумышленник может написать программу, которая извлекает конкретную информацию из информационной системы жертвы. Сбору может подвергаться как хранящаяся в системе, так и вводимая пользователем информация (так называемый *keylogging*).

- Модификация защищенных данных. Крупнотиражных троянов такого типа не встречается, но при целенаправленной атаке на конкретную жертву, конечно, ничто не помешает злоумышленнику разработать специальную программу для такой задачи, и часто это делается. Впрочем, ряд вирусов при определенных условиях уничтожают пользовательские данные, поэтому без особой натяжки могут быть отнесены к этой категории.
- Сбор статистики и другой информации, которая сама по себе не конфиденциальна, но может представлять ценность. Например, похитив паспортные данные жертвы, злоумышленник может в определенных пределах представляться жертвой при взаимодействиях с государственными учреждениями и частными предприятиями (так называемая "кражи идентичности"). Статистика обращения к интернет-сайтам может быть использована в маркетинговых целях; для сбора такой статистики существует целый обширный класс троянских программ, объединенных названием *SpyWare*. Эти программы, пожалуй, следует признать самыми безобидными из троянов. Впрочем, даже они потребляют ресурсы зараженного компьютера и доставляют неприятности пользователям.
- Передача пользователю рекламной информации. В действительности, бизнес-модель распространителей таких программ, как правило, имеет лишь косвенное отношение к собственно рекламе: в самом деле, если реклама внедряется вопреки активному противодействию пользователя, вряд ли она создаст у него положительное представление о заказчике этой рекламы, а положительное впечатление о рекламодателе — необходимое условие успешной торговли. Обычно авторы *AdWare* зарабатывают на обмане рекламодателей, которые платят за каждый показ их баннера: троян открывает баннеры со множества различных клиентских компьютеров, что позволяет обойти большинство распространенных схем борьбы с мошенничеством при подсчете показов.
- Использование в качестве "первичного загрузчика", который позволяет заразить систему каким-то другим троянским агентом. Таким образом обычно используются ненамеренно возникшие вредоносные программы — задние двери, ошибки программирования и т. д.
- Использование зараженной системы в качестве промежуточного агента при атаке какой-то другой системы. Я уже упоминал использование про-

грамм такого типа для усложнения отслеживания источника атаки, применение ботнетов для атак затоплением (сетевая атака отказа сервиса, когда жертва просто затапливается потоком запросов или похожих на запрос пакетов) и некоторые другие приемы. В действительности, сценарии такого использования троянских программ слишком разнообразны, чтобы их можно было исчерпывающим образом описать или классифицировать. Например, захватив контроль над рабочей станцией в сети компании-жертвы, злоумышленник может использовать ее для атаки на серверы этой же компании. Захватив контроль над сервером сетевой службы именования (DNS) или маршрутизатором, злоумышленник может перенаправлять запросы пользователей на свой сайт и собирать аутентификационную и другую конфиденциальную информацию.

- Распространение троянских программ. Формально это частный случай использования зараженной системы для атаки другой системы, но трояны этого типа — *черви* и *вирусы* — достаточно распространены и опасны, чтобы борьба с ними заслуживала выделения в специальное направление деятельности.
- Использование троянской программы в качестве агента удаленного управления. Такая контролируемая извне система (*зомби* или *бот*) может использоваться для всех вышеперечисленных целей.

Пожалуй, главная мораль, которую можно извлечь из данной классификации, — это соображение, что безопасность вычислительной системы, особенно системы, подключенной к сети — не одно только личное дело владельца этой системы. Даже если сама система не содержит никаких ценных данных, ее сетевой интерфейс и адрес представляют собой ценные ресурсы, которые могут быть захвачены и использованы злоумышленником для причинения вреда другим людям и организациям. Владелец системы, не предпринимающий адекватных действий для обеспечения ее безопасности, опасен не только (а часто даже и не столько) для себя, но и для окружающих.

Чтобы понять, какие меры можно и следует принять для защиты от троянских программ, необходимо подробнее обсудить, что это такое и откуда они берутся.

13.8.1. Троянские программы, внедряемые при разработке

Если вы запустили на вашем компьютере чужую программу, это больше не ваш компьютер.

Народная мудрость

Несомненно, что современная вычислительная техника была бы невозможна без тех или иных форм разделения и повторного использования кода. Все со-

временные ОС, средства разработки и прикладные программные пакеты разрабатывались огромными коллективами людей в течение нескольких десятилетий. Многих программные пакеты имеют богатую событиями историю, в которой права на код и обязательства по его поддержке неоднократно переходили от одной организации к другой — как это происходило с Unix, Windows NT (ранее известной как OS/2 NT), Microsoft SQL Server, Lotus Notes/Domino.

Нередки также и случаи, когда код переходил без явного согласия предыдущего правообладателя. Так, реализация стека протоколов TCP/IP в Windows NT/2000/XP/2003 основана на реализации тех же протоколов в BSD Unix; лицензия BSD допускает неограниченное, в том числе и коммерческое, использование исходных текстов опубликованных под этой лицензией текстов, однако запрещает рекламу продуктов такого использования как "основанных на BSD" без разрешения правообладателя.

Повторное использование кода неразрывно сопряжено с вопросом доверия к автору этого кода. В условиях, когда авторов у кода много и между ними существуют или возможны конфликты интересов (как, например, между издателем программного продукта и работающими на него программистами), удовлетворительно решить этот вопрос невозможно. Технических способов гарантировать отсутствие "закладок" в разработанных другими людьми программных продуктах не существует. Часто в качестве такого способа упоминают доступ к исходным текстам продукта; к сожалению, есть две важные причины, по которым этот способ малоэффективен.

Во-первых, анализ исходных текстов может быть обойден с помощью так называемой "закладки Томсона". Такая возможность была предложена Дэвидом Томсоном (одним из разработчиков Unix) на конференции ACM при обсуждении доклада MIT по безопасности разработки ОС Multics [Karger/Shell 1974]; к счастью, документированных случаев практического применения атак такого типа неизвестно. Тем не менее теоретическая их возможность заслуживает обсуждения.

Идея "закладки Томсона" в основе своей проста, хотя сама она относительно сложна в реализации. Идея состоит в том, что закладка встраивается не в исходные тексты системы, а в компилятор, которым этот текст обрабатывается. Например, компилятор может обнаруживать, что обрабатывает исходный текст программы login, и встраивать в него код, дающий пользователю с именем `athomson` административные права. Компилятор представляет собой гораздо более сложную программу, чем login, и спрятать закладку в его коде пропорционально проще. Анализ исходных текстов компилятора также может быть обойден встраиванием закладки в код того, чем компилируется компилятор. Если следует принимать во внимание возможность анализа машинного кода, то необходимо также встроить закладку в код деассемблера.

Разумеется, наиболее опасны такие закладки в условиях, когда система и компилятор поставляются одной организацией.

Понятно, что стоимость реализации такой закладки очень высока — но настолько же высока и стоимость ее обнаружения. Поскольку все современные программы так или иначе генерируются другими программами, можно сделать вывод, что если разработчик задастся целью внедрить закладку в разрабатываемый для вас программный продукт, он это сделает.

Во-вторых, существует гораздо более простое и практическое соображение. Известно много примеров, когда в продуктах с открытыми исходными текстами (и даже в продуктах с открытыми исходными текстами, для которых было проведено или, точнее, считалось, что было проведено, доказательство корректности) обнаруживали опасные ошибки. В том числе известен ряд примеров, когда такие ошибки существовали в коде продуктов и не были обнаружены много лет. Некоторые из этих примеров рассматривались в разд. 13.7. Можно привести соображения, в силу которых обнаружить не преднамеренную ошибку в чужом коде значительно сложнее, чем преднамеренно внедренный код; с другой стороны, очевидно, что все, что происходит само собой (как, например, ошибки), ценой определенных усилий можно воспроизвести и искусственно.

Поэтому анализ исходников, хотя, безусловно, и является относительно эффективным средством защиты от приходящего вместе с продуктом вредоносного или потенциально вредоносного кода, но его ни в коем случае нельзя считать панацеей.

13.8.1.1 "Закладки"

Враг не ведал, дурачина, — тот, кому все поручил он,
Был чекист, майор разведки и прекрасный семьянин.

В. Высоцкий

Умышленно внедряемые при разработке "закладки" или логические бомбы (*logical bomb*) представляют большую опасность. Как уже отмечалось, надежных технических средств защиты от них не существует. Тем не менее у крупнотиражных продуктов вопросы защиты от закладок так или иначе разрешаются, главным образом за счет репутации автора или издателя программы. Опасность целенаправленно вносимых злонамеренных "закладок" в крупнотиражных программах следует признать скорее теоретической — производители таких программ уделяют большое внимание контролю процесса разработки и налаживанию нормальных взаимоотношений со своими сотрудниками. Однако "задние двери" и ошибки программирования встречаются в таких программах приблизительно так же часто, как и в программах, распро-

страняемых мелкими тиражами и разрабатываемых для внутреннего использования (так называемых *in-house* разработках).

Парадоксально, но наиболее опасны в плане закладок именно внутренние программные проекты. Даже проекты, разрабатываемые по разовым заказам (так называемые *outsourcing*), доставляют в этом отношении гораздо меньше проблем. Аутсорсинговая компания заинтересована в своей репутации настолько же, если не больше, чем компания, занимающаяся разработкой тиражируемых продуктов. Такая компания может позволить себе относительно большие вложения в построение процесса разработки и в обеспечение лояльности разработчиков.

Приглашенный со стороны разработчик, хотя и имеет техническую возможность внедрить закладку в код разрабатываемого продукта, обычно не имеет полного представления о бизнес-процессе компании-заказчика и, на практике, довольно сильно ограничен в возможностях построения и использования эффективной "закладки". Действительно, ведь аутсорсинговые контракты обычно не предусматривают доступа разработчиков к системе во время ее эксплуатации. Риск ущерба репутации в таких условиях намного превосходит возможный выигрыш от подобных действий. Приглашенный со стороны разработчик также не имеет никакого отношения к внутрикорпоративным интригам и обидам, так что не имеет стимулов встраивать "закладку" по субъективным причинам.

Все приведенные рассуждения необходимо полностью обратить, когда мы говорим о внутреннем проекте в строгом смысле этого слова, т. е. о программах, разрабатываемых сотрудниками компании, в которой эта программа будет эксплуатироваться. Такие проекты обычно реализуются в условиях жестких бюджетных ограничений; их разработчики низко оплачиваются и занимают невысокое место в корпоративной иерархии. Поскольку разработчик низкооплачиваемый, часто это свидетельствует о его низкой квалификации и/или о том, что ему практически нечего терять в смысле репутации.

С другой стороны, такой разработчик обычно штатно имеет административный доступ к системе, либо потому, что совмещает обязанности разработчика и администратора данных, либо в рамках своих обязанностей по поддержке этой системы; он хорошо знает весь бизнес-процесс компании, а значит, и узкие места в контроле за этим бизнес-процессом. Будучи сотрудником компании, он в курсе или является участником внутрикорпоративных интриг и может иметь сложные личные отношения с кем-то из менеджеров или собственников компаний.

Сотрудник компании имеет еще один своеобразный стимул для встраивания закладок во внутренний продукт — желание защититься от увольнения или иметь возможность отомстить руководству компании после увольнения.

Классический и часто реализуемый на практике пример такой закладки — это проверка в бухгалтерской программе, находится ли программист в ведомости на начисление зарплаты.

Впрочем, необходимо отметить, что есть ряд примеров "закладок" в коммерческих программах, когда можно подозревать злые намерения разработчика; в большинстве таких случаев злой умысел остался недоказанным. Так, в Microsoft Frontpage Extensions 3.0.2.1105 (продукт, который обеспечивает редактирование веб-страниц непосредственно на сервере) была обнаружена задняя дверь, позволяющая прочитать исходные тексты скриптов ASP, с помощью которых реализуются многие веб-сайты с динамическим контентом. Такие скрипты часто содержат имена и пароли для доступа к базам данных; кроме того, анализ этих текстов может быть полезен для поиска ошибок в скриптах и возможных точек атаки на веб-сайт, например путем внедрения SQL-кода. Доступ к этой двери контролировался паролем "Netscape engineers are weenies!" (инженеры Netscape — нытики) [bugtraq RFP2K02].

Механизм доступа и реальные полномочия, которые можно было получить таким образом, слишком своеобразны для того, чтобы считать это отладочной "задней дверью", подобной тем, которые рассматриваются в следующем разделе. В пользу гипотезы о том, что это умышленно внедренный код, свидетельствовал и тот факт, что версия DLL, в которой он был обнаружен, отличалась от версии остальных загрузочных модулей того же продукта. Результаты внутреннего расследования, проводившегося компанией Microsoft, опубликованы не были.

13.8.1.1. "Задние двери"

Оглянулся — все тихо, хвоста вроде нет.
Колодец двора, яма чёрного хода
Заколочена. Чёрт бы побрал этот свет,
Липнущий сверху чухонским уродом.
Выход — гнилая пожарная лестница
Хрупкая, сволочь, и окна вокруг.

В. Шевчук

При разработке и тестировании системы разработчики часто нуждаются в нештатных средствах доступа к ней. Особенно это справедливо для систем, реализующих сложные и теоретически "непробиваемые" средства защиты от несанкционированного доступа.

Чаще всего такие средства реализуются в виде жестко закодированных учетных записей, которые не содержатся в штатной БД этих записей, но позволяют разработчику аутентифицироваться в системе. Такие записи могут быть даже необходимы, например при отладке модулей, модифицирующих эту

БД, — ведь при ошибке в этих модулях возможно разрушение БД, после которого никто не сможет аутентифицироваться в системе.

Обычно такие средства должны удаляться или обходитьсь директивами условной компиляции при сборке распространяемой версии продукта, но этого нередко не происходит — часто без всякого злого умысла, просто по забывчивости.

Анализ исходных текстов относительно эффективен для обнаружения закладок такого типа, главным образом потому, что их обычно никак специально не прячут. Так, один из относительно недавних примеров такого рода — жестко закодированная административная учетная запись в Borland Interbase — был обнаружен только после того, как Borland опубликовал исходные тексты продукта [bdn.borland.com backdoor].

Впрочем, есть и довольно громкие примеры "задних дверей" в продуктах, распространяемых с открытыми исходниками. Так, до 1987 года реализация почтового сервера Sendmail имела команду DEBUG, которая позволяла исполнить на машине, на которой был запущен сервер, произвольную команду с произвольными параметрами. Червь Morris использовал эту "заднюю дверь" как один из каналов распространения.

13.8.1.2. Ошибки программирования

Епифан казался жадным, хитрым, умным, плотоядным,
Меры в женщинах и в пиве он не знал и не хотел.
В общем, так: подручный Джона был находкой для шпиона.
Так случиться может с каждым, если пьян и мягкотел.

В. Высоцкий

Ошибки программирования представляют собой одну из самых важных проблем для современной отрасли информационных технологий. До 90% стоимости разработки программ и 100% стоимости их поддержки приходится на поиск, исправление или обход ошибок. При всем этом рассуждения о полном искоренении ошибок из крупных программных комплексов следует признать утопическими и даже беспредметными.

Полный обзор и исчерпывающая классификация всех практически встречающихся типов ошибок, по-видимому, невыполнимы. Тем не менее мы рассматривали ряд наиболее важных (то есть часто встречающихся и в то же время опасных) ошибок в разд. 13.7.

В первом приближении, ошибки можно классифицировать по их опасности. Подавляющее большинство ошибок может использоваться как точка отказа сервиса (спровоцировав такую ошибку, можно "уронить" сервисный процесс или атакуемую систему целиком). Некоторые ошибки можно использовать

для повышения уровня привилегий, несанкционированного доступа к данным или внедрения троянского кода (как обсуждавшиеся в разд. 13.7.1 срыв стека и внедрение скрипта). Очевидно, что ошибки последнего типа наиболее опасны, но при этом они довольно-таки распространены. Появление ошибок такого типа в сетевых сервисах может иметь совершенно сокрушительный эффект.

Сопоставимую по уровню опасности проблему представляют ошибки в модулях, связанных с проверкой ACL, авторизацией и повышением уровня привилегий процессора (например, в диспетчере системных вызовов), а в системах семейства Unix — в setuid-программах.

Так, в системах Windows 2000 и Windows XP — как в ядрах, так и во входящих в стандартную поставку библиотеках и сетевых сервисах, таких как сервисы SMB (Microsoft Network) и IIS, — со временем их выхода было исправлено несколько сотен срывов стека; некоторые из этих ошибок использовались вирусами и сетевыми червями.

Можно сделать вывод, что разработчикам, в общем-то, не нужно целенаправленно внедрять в программные комплексы никакого вредоносного кода — такого кода в коммерческих программах и без того предостаточно. В продуктах с открытыми исходными текстами ситуация несколько лучше, чем в "закрытых" ОС, таких как Windows XP/2003, но разница здесь лишь количественная, а не принципиальная. Количество ошибок, обнаруживаемых в популярных дистрибутивах Linux за достаточно большой интервал времени, уступает количеству аналогичных ошибок в Windows в разы или на десятки процентов; в разных категориях опасности ошибок разница различна, но ни в одной из категорий не наблюдается разницы на порядок.

Единственный путь к решению этой проблемы на практике состоит в построении эшелонированных систем безопасности, в которых прорыв "периметра" в любой из возможных точек не может привести к полному захвату злоумышленником контроля над сетью предприятия, а приводит лишь к получению доступа к следующему барьеру. Основной прием, который для этого можно использовать, — это разделение системы на более или менее изолированные друг от друга компоненты с контролируемыми интерфейсами между ними. Такие компоненты должны работать с наименьшим из наборов привилегий, которые необходимы им для нормальной работы.

Машины, предоставляющие сервисы для публичных сетей (веб-серверы, почтовые шлюзы и т. д.), следует изолировать от приватной сети компании и, во всяком случае, не следует совмещать публичные сервисы с хранением чувствительных приватных данных на одной машине. Следует рассмотреть возможность установки фильтрующих маршрутизаторов между сегментами приватной сети компании так, чтобы прорыв безопасности в одном из сегментов сети не мог быть распространен на всю остальную сеть.

13.8.2. Троянские программы, внедряемые при распространении

Наш чай, каким бы сортом он ни поступал в продажу, ни на какой стадии производства ничем не подкрашивается и решительно никаких посторонних примесей не содержит.

Книга о вкусной и здоровой пище

Впервые вопрос об опасности внедрения враждебного кода в каналы распространения программных продуктов был поставлен в знаменитом докладе [Karger/Shell 1974], в котором было продемонстрировано, что существующая схема поставки ОС Multics заказчикам недостаточно защищена.

Из общих соображений представляется очевидным, что программы, распространяемые через публичные файловые архивы с доступом по FTP или HTTP, более уязвимы для внедрения троянов, чем программы, распространяемые на дискетах или CD. Поэтому, в частности, немаловажной статьей доходов дистрибуторов Linux является поставка системы на "гарантированно чистых" компакт-дисках.

Действительно, наиболее громкие из историй о троянском коде в популярных программах связаны именно с раздачей дистрибутивов по сети.

Закладка в *Sendmail 8.12.6*

В 2002 году неизвестными лицами был взломан FTP-сервер, с которого распространялись исходные тексты популярного почтового сервера *Sendmail*. В исходный текст и в скрипты, управляющие сборкой продукта, была вставлена довольно-таки грубая закладка, которая во время компиляции устанавливала соединение TCP с сервером злоумышленника и позволяла ему запустить на зараженной системе командный интерпретатор.

Обнаружение закладки было несколько затруднено тем фактом, что из четырех "зеркальных" серверов, с которых раздавались тексты, был взломан только один, так что с вероятностью 75% пользователь получал "чистую" версию продукта. Это привело к тому, что закладка была обнаружена лишь более чем через неделю после появления на сайте [www.cert.org CA-2002-28].

Справедливо ради, необходимо отметить, что большинство современных программных продуктов, распространяемых на компакт-дисках, в том числе MS Windows и MS Office, практически непригодны для эксплуатации без регулярно скачиваемых через Интернет "заплат" и обновлений. В этом смысле сайт windowsupdate.microsoft.com представляет собой гораздо более привлекательную мишень для злоумышленников, чем сайт ftp.sendmail.org. К счастью, ко времени выхода второго издания сам сайт Windows Update еще никому взломать не удалось (а если и удавалось, то опубликованных сведений об этом нет). Однако многие компании с большими парками машин под управлением Windows для экономии внешнего трафика и ускорения работы

устанавливают локальные зеркала этого сайта. Microsoft даже предоставляет для этого штатные средства. Очевидно, что такие зеркала гораздо более уязвимы, чем центральный сервер обновлений Microsoft.

Одно из относительно эффективных решений этой проблемы — электронная подпись дистрибутивов программных продуктов. В частности, она используется и для защиты обновлений, распространяемых через службу Windows Update.

Однако необходимо отметить, что любая электронная подпись заслуживает доверия не более, чем ключ, которым подписываются сообщения и, с другой стороны, чем код, которым проверяется подпись. Моего воображения не хватает, чтобы полностью описать катастрофу, к которой приведет утечка или подбор ключа, используемого для подписи обновлений Windows Update или (что более реально) обнаружение ошибки (например, срыва буфера) в клиентском коде, который проверяет валидность этих обновлений.

13.8.3. Троянские программы, внедряемые в уже установленную систему

— Это мой зуб!

— Слыкали, какой глупый! Он не твой, и даже не мой, понял! Он ихний!

Ю. Дунский, В. Фрид, к. ф. "Не бойся, я с тобой"

В начале раздела мы видели основную классификацию способов внедрения враждебного кода в уже установленную систему. Внедрение путем физического доступа к компьютеру мы, как уже договаривались, не будем подробно обсуждать. Впрочем, важно понимать, что в современных условиях всего нескольких секунд доступа к оставленному без присмотра терминалу может оказаться достаточно, чтобы скачать из сети или скопировать с удаленного носителя (дискеты, CD или флеш-памяти) троянскую программу довольно большого объема.

В современных условиях гораздо более распространены и, соответственно, гораздо более опасны ситуации, когда троянские программы внедряются в систему без физического доступа злоумышленника к ней. Два основных метода такого внедрения:

- запуск зараженной программы самим пользователем;
- удаленный доступ, например через срываы буфера или ошибки, допускающие внедрение скрипта, в сетевых сервисах или в программах, которые работают с внешними данными.

При этом, разумеется, первый метод обычно требует использования тех или иных социально-инженерных средств. Впрочем, у большинства пользовате-

лей не очень хорошо срабатывает интуиция, и они недооценивают опасность, сопряженную с запуском посторонних программ на своем компьютере. Поэтому зачастую не требуется такой уж сложной инженерии, чтобы уговорить пользователя открыть исполняемый файл, пришедший по почте с комментарием "прикольная картинка" или согласиться на установку контрола ActiveX, "необходимого" для просмотра порносайта.

Другой источник возможностей запуска троянских программ рассматривался в начале разд. 13.8: ошибки в настройке порядка поиска запускаемых программ, из-за которых система при определенных обстоятельствах пытается искать программы в каталогах данных. Если в системах семейства Unix это именно ошибка администрирования (включение текущего каталога в переменную среды PATH, особенно в ее начало), то в системах семейства CP/M это ошибка программирования или даже проектирования ОС — в этих системах жестко закодировано, что исполняемые модули и DLL в первую очередь ищутся в текущем каталоге.

13.8.3.1. Вирусы и аналогичные троянские программы

— Поправился товарищ Комманд Ком, — произнес Диггер, рассматривая проходящего мимо товарища Кома.
— На 2.7 кило, — определил командир Нортон.
— Так какую песню мы сегодня будем петь, Нортон? — спросил Диггер.
— Да, — отвечал командир Нортон, — Янки Дудль.

Л. Голубев

Наиболее важным примером троянов, внедряющихся в установленную систему посредством запуска пользователем, являются компьютерные вирусы.

Классические вирусы были очень распространены в первой половине 90-х годов, в эпоху MS/DR DOS. Они распространялись, главным образом, с играми и другими программами — как нелицензионными, так и бесплатными и условно-бесплатными. Когда зараженная вирусом программа запускалась на компьютере, код вируса искал другие исполняемые модули в системе и внедрялся в них.

Приемы внедрения не отличались большим разнообразием. Типичный вирус дописывал себя в конец EXE-файла и перенастраивал на себя стартовый адрес. Затем, исполнив свой код, вирус передавал управление на настоящий стартовый адрес программы. Более сложная схема, пригодная для заражения файлов с фиксированным стартовым адресом (COM-файлов), состояла в том, чтобы подменить первые несколько команд программы на команду перехода на тело вируса. Закончив свою работу, вирус копировал сохраненные оригинальные команды на место и передавал на них управление, аналогично тому, как делают отладчики при проходе точки останова (см. разд. 2.10.1).

Отдельной категорией вирусов следует признать упоминавшиеся в разд. 3.11 загрузочные вирусы, которые поражали только загрузочные секторы жестких дисков и дискет; они поражали новые системы при ошибочных загрузках пользователей с зараженных дискет.

Гораздо большим разнообразием отличалось поведение запущенного вируса. Некоторые вирусы только размножались. Другие вирусы при определенных обстоятельствах — например, при наступлении определенной даты или времени — запускали различные вредоносные действия. Диапазон этих действий варьировал от относительно безобидных (например, исполнение мелодии Yankee Doodle на спикере) до крайне деструктивных (например, один из малораспространенных вирусов менял местами указатели на функции чтения из файла и записи в файл, так что при попытке чтения в файл записывался мусор из буфера, в который предполагалось чтение).

Самые простые вирусы активизировались только в момент запуска зараженной программы и прекращали свою работу, передав ей управление. Более сложные — так называемые резидентные (resident) — вирусы внедряли себя в ядро ДОС и могли активизироваться при каждом системном вызове. Например, они могли заражать каждый исполняемый модуль, с которым так или иначе работали на зараженной системе. Если пользователь копировал на дискету ранее не зараженную игру, резидентный вирус мог заразить как копию игры на диске, так и вновь создаваемую копию на дискете.

Резидентные вирусы, в частности, могут перехватывать обращения к зараженным файлам и, если обращение происходит с целью чтения файла, а не его загрузки, удалять из создаваемого в памяти образа файла все следы своего присутствия. В частности, благодаря этому, при антивирусном сканировании зараженной машины вирус не может быть обнаружен. Такие вирусы называются stealth-вирусами.

Stealth-вирусы впервые продемонстрировали широким массам компьютерных специалистов один печальный факт, который должен осознавать любой борец с троянскими программами: если троян написан грамотно и исполняется с достаточно высокими привилегиями, то средствами зараженной системы его обнаружить невозможно. Обычно этот факт формулируют иначе: **невозможно достоверно гарантировать "чистоту" системы средствами самой этой системы.**

Приемы сокрытия своего присутствия в системе, аналогичные stealth-вирусам, используют неспособные к самостоятельному размножению троянские программы, такие как SpyWare и AdWare. Большинство этих программ практически невозможно удалить штатными средствами ОС.

Для надежных гарантий чистоты при любых подозрениях на внедрение вируса или другой троянской программы необходимо загружаться с гарантиров-

ванно чистого дистрибутивного носителя и переустанавливать систему или, во всяком случае, тщательно проверять ее на совпадение с "чистым" образом.

Если мы будем строить регламент работы с системой в строгом соответствии с принципами, которые нам диктует предыдущее рассуждение, мы получим нечто, похожее на принципы организации асептики в современной хирургии, когда все предметы в операционной комнате делятся на категории стерильности. Соприкосновение одного предмета (например, инструмента или одетой в перчатку руки) с другим предметом более низкой категории (например, со стеной операционной комнаты) автоматически обозначает перевод в более низкую категорию и, как правило, требует повторной стерилизации. Студентов хирургических специальностей медицинских вузов буквально дрессируют на соблюдение такого регламента.

Организация работы в соответствии с такими принципами чрезвычайно дорога, как из-за трудозатрат администратора, так и из-за частых и длительных незапланированных остановок системы на обслуживание, поэтому реальное поведение администраторов большинства вычислительных систем больше похоже на работу не хирурга, а сельского ветеринара. Уронив скальпель на пол хлева, ветеринар вытирает его о фартук и продолжает операцию; затем он засыпает операционное поле пенициллином ("антивирусом") и надеется на естественный ход событий [Хэрриотт].

Разумеется, такие методы работы терпимы лишь постольку, поскольку защищаемые данные имеют относительно низкую ценность и не могут оправдать дорогостоящих мер безопасности. Так, лечение в соответствии с описанной ранее схемой терпимо лишь постольку, поскольку мы готовы принять неизбежно следующую за ним рекомендацию по послеоперационному уходу: "если до утра температура не спадет, прирежьте, чтобы не мучился".

Распространение Windows 95 сделало многие резидентные вирусы для DOS нежизнеспособными; кроме того, распространение нового формата исполняемых модулей (Win32 PE) сделало нежизнеспособными и многие более мирные формы вирусов, которые полагались на определенную структуру загрузочных модулей, поэтому в 1995 — начале 1996 года у многих пользователей и даже системных администраторов возникла иллюзия, что новая система безопасна. Эта иллюзия быстро развеялась, когда весной 1996 года произошла пандемия поливалентного вируса Chernobyl, который жил и в загрузочных модулях Win32, и в загрузочных секторах.

При переходе на Windows NT многие также рассчитывали, что система безопасности этой ОС позволит защитить код системы и установленных приложений от вирусов. Действительно, если пользователь работает без привилегий локального администратора, он не сможет модифицировать код системы и приложений. Поэтому, даже если пользователь по тем или иным причинам

запустит зараженную программу, вирус, возможно, сможет заразить частные программы пользователя, но не сможет заразить всю систему. Тем более такой вирус не смог бы внедриться в ядро системы и скрыть свое присутствие.

Эти иллюзии также быстро развеялись — теоретически вполне удовлетворительная модель безопасности Windows NT оказалась крайне неудобна при реальной работе, так что большинство пользователей оказались вынуждены постоянно работать с правами локального администратора. В значительной мере это было обусловлено тем, что многие программы для Win32 разрабатывались для незащищенной Windows 95. Такие программы сохраняют свои настройки в защищенных ветвях реестра или в файлах в подкаталогах C:\Program Files, размещают разделяемый код в C:\Windows, пытаются модифицировать другие защищенные ресурсы и оказываются неработоспособны при низких уровнях привилегий в Windows NT.

До настоящего времени эта проблема не решена, а в некоторых отношениях даже усугубилась. Так, в Windows XP все локальные учетные записи по умолчанию создаются с правами локального администратора.

Поэтому системы линии Windows NT оказались столь же, а в некоторых отношениях даже более уязвимы для вирусов, как и Windows 95. Тем не менее очевидно, что защита кода от модификации является важным барьером на пути распространения вирусов и внедрения других троянов.

Столь же важным барьером на пути распространения и внедрения троянских программ может быть защита файлов и каталогов от исполнения. К сожалению, в файловых ACL Windows XP/2003 до сих пор отсутствует флаг защиты от исполнения; если пользователь может прочитать файл, он может исполнить его. В ACL систем семейства Unix, как мы видели из разд. 13.4.1, соответствующее право есть — это относится как к традиционным упрощенным ACL, так и к произвольным ACL, реализованным во многих современных системах этого семейства.

Аналогично, в системах семейства Unix при монтировании сетевых файловых систем и запоминающих устройств со сменными носителями можно использовать ключи `poexec` (запретить исполнение), запрещающий исполнение любых программ с этих файловых систем, и `nosetuid` (запретить setuid), запрещающий интерпретацию бита setuid. Если необходимо, такие ключи можно использовать и при монтировании томов на стационарных жестких дисках. Они могут защитить не только от ошибочного запуска троянской программы самим пользователем, но и от запуска программы злоумышленником, получившим кратковременный доступ к оставленному без присмотра компьютеру.

Напротив, в системах семейства CP/M ничего подобного нет; Windows XP SP2 при запуске программ с удалаемых, а при некоторых обстоятельствах —

с сетевых устройств спрашивает пользователя, действительно ли он хочет исполнить программу; понятно, что это может остановить пользователя при ошибочном запуске, но никоим образом не может служить защитой от злоумышленника. Более того, все Windows, начиная с Windows 95 и NT 4.0, по умолчанию запускают файл autorun.inf из корневого каталога каждого компакт-диска, вставляемого в привод! В штатном пользовательском интерфейсе до сих пор нет даже возможности отключить это поведение; для того, чтобы его выключить, необходимо изменить ключ реестра `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\CDRom\Autorun` [support.microsoft.com 155217].

Sony rootkit

Наиболее скандальный пример злоупотребления autorun.inf — это так называемый Sony rootkit. В 2004—2005 годах звукозаписывающая компания Sony BMG опубликовала ряд аудио-CD с защитой от копирования, известной как Sony XCP или First 4 Internet DRM (по сведениям из ряда источников, XCP разрабатывалась компанией First 4 Internet в качестве субконтрактора). Список дисков в конце концов был опубликован на сайте [[cp.sonybmg.com xcp](http://cp.sonybmg.com/xcp)].

Схема защиты от копирования основывалась на использовании autorun, на том, что большинство пользователей Windows работают из-под локального администратора, а также на том, что на CD, содержащих одновременно аудиотреки и треки данных ISO 9660 Windows по умолчанию показывает только треки данных. Диски содержали аудиотреки для проигрывания на традиционных CD-проигрывателях, а трек данных содержал файл autorun.inf, который при запуске внедрял в систему пользователя руткит и собственно сервис защиты от копирования.

Сервис защиты от копирования блокировал прямой доступ к аудиотрекам диска и позволял считывать их в виде MP3-файлов с низким битрейтом. Любопытно, что программа, кодирующая треки, была основана на популярном MP3-кодере LAME, который распространяется на условиях LGPL. Программа была статически собрана с кодером и поставлялась без исходных текстов и даже без ссылки на LAME, что представляет собой прямое нарушение LGPL.

М. Руссинович и другие исследователи кода этого сервиса весьма неподобранно отзываются о качестве программы и о способах, которыми она перехватывает доступ к аудио-трекам. На сайте [[www.boingboing.net prehistory_of_the_so](http://www.boingboing.net/prehistory_of_the_so)] приводится интересная подборка ссылок на письма сотрудников компании First 4 Internet в технические списки рассылки и группы Usenet, демонстрирующие их низкую компетентность в сочетании с интересом к технологиям DRM и нештатному использованию интерфейсов ядра Windows.

Наиболее неприятной частью программы, однако, следует признать руткит, который скрывал наличие сервиса защиты от копирования в системе и исключал возможность его выключения и удаления штатными средствами. Этот руткит внедрялся в ядро Windows (это было возможно потому, что чаще всего autorun.inf запускался от имени локального администратора, т. е. пользователя, имеющего право устанавливать модули ядра), менял код диспетчера системных вызовов и перехватывал все обращения к файловой системе, ключам реестра и списку процессов. При этом он модифицировал списки ключей реестра,

список файлов в каталоге Windows и список процессов, отфильтровывая все имена, начинавшиеся с "\$sys\$". Легко понять, что такое поведение позволяет скрыть не только сервис XCP, но и другой вредоносный код, т. е. машина, зараженная руткитом XCP, становится уязвима для любых троянских программ, авторы которых знают про XCP.

В октябре 2005 года в блоге Марка Руссиновича была опубликована информация о рутките [blog.technet.com markrussinovich 2005/10/31] и инструментарии, с помощью которого он может быть обнаружен. После скандала в прессе и угроз коллективного иска, Sony BMG объявила об отзыве дисков, защищенных посредством XCP, и обязалась опубликовать эффективные средства для удаления троянского кода с зараженных компьютеров. Вторая волна скандала поднялась после того, как выяснилось, что первая версия утилиты для удаления троянского кода, опубликованная на сайте Sony в первых числах ноября 2005 года, не удаляла маскирующую часть руткита, так что после ее работы система оставалась в столь же уязвимом состоянии, как и с полностью установленным трояном.

13.8.3.2. Активный контент

Но могут действовать они не прямиком:
Шашь в купе, и притворится мужиком,
А сама наложит толу под корсет.
Ты проверяй, какого полу твой сосед.

В. Высоцкий

Особенно большую опасность в качестве средства внедрения троянских программ представляет так называемый "активный контент", т. е. совмещение данных и кода, или, точнее, внедрение в данные различных "активных элементов", так или иначе реагирующих на действия пользователя при просмотре и/или модификации документа. Характерным примером такого совмещения являются макрокоманды, содержащиеся в стилевых файлах и в документах Microsoft Office. В конце 90-х годов распространявшиеся с документами MS Office макровирусы представляли большую проблему. Новые версии пакета содержат средства, позволяющие в определенных пределах контролировать исполнение этих макропрограмм, но крайняя непродуманность данных средств вынуждает многих пользователей отключать их.

Теоретически, ряд других форматов представления форматированного текста, такие как TeX и PostScript, также представляют собой достаточно мощные языки программирования, допускающие чтение, модификацию и создание файлов на компьютере, на котором происходит компиляция (а в случае PostScript — просмотр) документа. Демонстрировались даже примеры PostScript-вирусов и LaTeX-вирусов, однако жизнеспособных диких форм таких вирусов на сегодня не известно. По-видимому, дело в том, что названные языки программирования все-таки предоставляют недостаточно средств для определения среды, в которой просматривается документ, поэтому автор

вируса не может разместить внедряемый код в места, в которых он был бы действительно опасен и смог бы размножаться.

Главная опасность от совмещения данных и кода состоит в том, что даже если научить пользователей не запускать пришедшие из неизвестных источников программы, в большинстве случаев они не смогут избежать просмотра пришедших из неизвестных или не заслуживающих доверия источников данных. Если такие данные будут представлять собой "активный контент", это может привести к заражению компьютера пользователя троянской программой.

Почтовые вирусы

В конце 90-х — начале 2000-х годов большую опасность представляли почтовые вирусы, распространяющиеся в виде присоединенных файлов по почтовым системам SMTP и Microsoft Exchange. Некоторые из этих вирусов предполагали запуск присоединенного файла пользователем (это обеспечивалось "сobelазнительными" темами писем, например "прикольная картинка" или "вот информация, которую вы запрашивали"), другие полагались на то, что старые версии почтовой программы Microsoft Outlook по умолчанию запускали все присоединенные к письму исполняемые модули.

Задача уговорить пользователя запустить присоединенный файл упрощалась тем фактом, что в Outlook и многих других популярных почтовых клиентах, присоединенный файл открывается в соответствии с настройками Windows Explorer, в зависимости от его расширения: для файлов с расширением doc запускается Word, с расширением xls — Excel, с расширением exe — сам файл. Тем или иным способом замаскировав расширение и/или подделав "иконку" файла, разработчик вируса мог убедить пользователя, что он открывает документ Word, а не запускает троянскую программу.

Большинство почтовых вирусов, будучи запущены, сканируют адресную книгу Outlook и рассылают себя по всем адресам из этой книги. Более изощренные современные вирусы при такой рассылке подделывают исходящие адреса писем, подставляя в их качестве случайные записи из той же адресной книги. Из-за этого невозможно остановить эпидемию вируса в корпоративной сети, просто разослав объявление "не открывайте никакую почту от Иванова до особого объявления".

Судя по тому, что пандемии почтовых вирусов продолжают возникать каждые несколько месяцев, в сети еще немало пользователей со старыми версиями Outlook и/или пользователей, которые еще не научились внимательно смотреть на то, что присоединено к письмам. Впрочем, часть этих эпидемий обусловлена поливалентными вирусами, которые используют почту лишь как один из каналов распространения.

Вторая сложность при борьбе с вредоносным "активным контентом" состоит в том, что во многих случаях можно защитить код системы от пользователя, во всяком случае от той учетной записи, из-под которой пользователь чаще всего работает, от модификации. Но защитить от этой учетной записи данные невозможно, не лишив пользователя возможности работать с ними. Поэтому,

если от обычных вирусов в довольно широких пределах можно (и следует) защищаться с помощью средств авторизации операционной системы (как защиты данных от исполнения, так и защиты кода от модификации), от "активного контента" таким способом защититься невозможно.

Тем не менее интерес к "активному контенту" существует и время от времени даже возрастаєт, потому что активные элементы данных зачастую действительно могут расширить функциональность приложений, работающих с такими данными. Основной подход, позволяющий защитить систему, на которой исполняется такой код, от последствий его исполнения — это исполнение кода в специальной изолированной среде, так называемой "песочнице" (sandbox). При этом уровень изоляции может задаваться для каждого отдельного приложения или источника таких активных элементов — например, элементам, пришедшем из одних источников (например, с сервера корпоративной сети) может позволяться доступ к локальным файлам, а активным элементам, скачанным из Интернета, такого права давать не следует. В простейшем случае, происхождение кода может контролироваться по тому, откуда были получены соответствующие данные, но, разумеется, гораздо более привлекательна электронная подпись.

Действительно, если песочница будет определять источник кода только по адресу узла сети, с которого он был скачан, то это сделает систему уязвимой для множества злоупотреблений. Сетевые адреса легко подделать (так называемый спуфинг); злоумышленник может перенаправить пользователя на свой сайт, используя подделку маршрутизационной информации или ответов службы разрешения сетевых имен. Кроме того, если злоумышленник сможет изменить содержимое доверяемого сайта, он сможет распространять с него троянский код. Так, если пользователи корпоративной сети доверяют активному контенту, полученному от корпоративного сервера, то, взломав этот сервер, злоумышленник сможет поразить своим трояном всю сеть. Поэтому схемы, основанные на одном только адресе, с которого пришел элемент (как, например, "зоны безопасности" Internet Explorer), следует признать крайне малоэффективными.

Важно понимать также, что подпись кода, при всех ее достоинствах, не может защитить от всех социально-инженерных приемов, которые могут использоваться для того, чтобы уговорить пользователя расширить полномочия для данного конкретного приложения или запустить его с повышенным уровнем привилегий. Многие эффективные приемы борьбы с этими приемами, например централизованное управление настройками песочниц, применимы только в корпоративных сетях, но не на домашних компьютерах.

Кроме того, подпись может обеспечить хорошую защиту только в условиях корпоративной сети, когда все источники кода и сертификатов известны и авторизованы администратором сети. Управление ключами и сертификатами

в условиях Интернета, особенно отзыв сертификатов — проблема гораздо более сложная. Ее техническое решение обсуждается в [Иртегов 2004, Танненбаум 2002, Шнайер 2003] и других работах, описывающих инфраструктуру сертификации X.509, но это решение не устраняет всех организационных проблем. Примеры социально-инженерного злоупотребления публичными сертификатами уже известны (рис. 13.18).

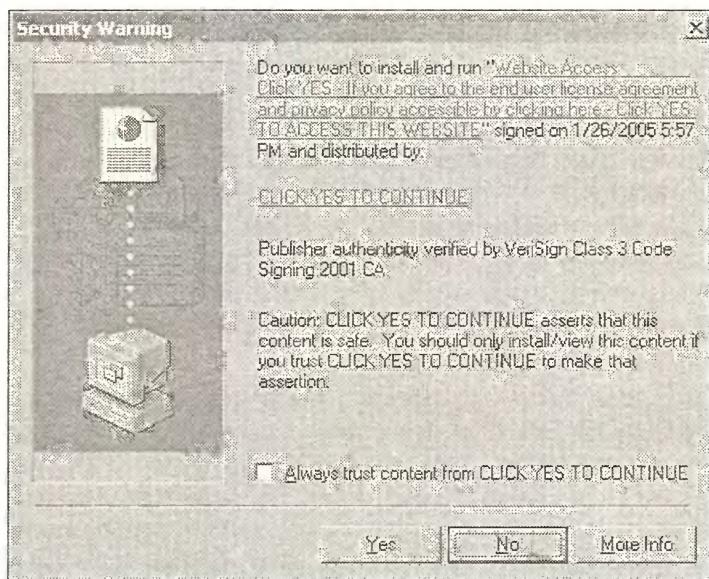


Рис. 13.18. Предупреждение о возможных проблемах с безопасностью Internet Explorer, по материалам сайта [www.benedelman.org 020305-1]

Рис. 13.18 нуждается в дополнительном комментарии. Это модальное окно выдавалось при открытии веб-страниц, содержащих контрол ActiveX, который, в свою очередь, устанавливал на компьютер троянскую программу типа SpyWare. При установке скачиваемых из Интернета контроллов ActiveX Windows проверяет наличие у них подписи сертификатом X.509. Пользователь может проверить подпись и решить, доверяет ли он источнику кода. К сожалению, выбирая имя организации и другие параметры сертификата, злоумышленник может дезориентировать пользователя — как это и происходит в обсуждаемом случае. Большую помочь злоумышленнику в данном случае оказывает структура диалога Windows, в котором нигде явно не сказано, что "CLICK YES TO CONTINUE" (нажмите [кнопку] YES, чтобы продолжить) — это на самом деле имя организации!

Очевидно, работники компании Verisign, обрабатывая заявку на регистрацию этого сертификата, были недостаточно внимательны. Для исправле-

ния ошибки, разумеется, необходим отзыв сертификата (по сообщению [www.benedelman.org 020305-1], конкретно этот сертификат был отозван вскоре после обнаружения); к сожалению, инфраструктура отзыва сертификатов X.509 далека от совершенства, а большинство пользователей не обновляют своевременно списки отозванных сертификатов.

Среди коммерчески успешных примеров относительно защищенных "песочниц" следует назвать, в порядке появления на рынке:

- ECL Lotus Notes/Domino;
- виртуальные машины Java или, точнее, среду, в которой JVM исполняет апплеты (applet), в отличие от среды, в которой исполняются полноправные приложения;
- скрипты Macromedia Flash, которые всегда исполняются в полностью изолированной песочнице;
- "управляемый код" (managed code) Microsoft .NET.

ECL Lotus Notes/Domino

В Lotus Notes/Domino пользовательские интерфейсы баз данных содержат многочисленные активные элементы: агенты, кнопки в представлениях и формах, проверочные формулы полей форм и т. д. Эти активные элементы могут реализовываться на различных интерпретируемых языках программирования. Lotus Notes R6 поддерживает три таких языка:

- @-формулы (своеобразный язык, название которого происходит от того факта, что имена всех встроенных функций этого языка начинаются с символа '@');
- Lotus Script (диалект Basic);
- Java.

Все перечисленные языки имеют достаточно мощные средства доступа к среде, в которой они выполняются, в том числе доступа к другим базам данных (при этом могут модифицироваться не только данные, но и элементы дизайна этих баз), чтения и модификации файлов на рабочей станции, изменения настроек клиента, а программы на Lotus Script и Java — даже вызывать любые функции в произвольных DLL. Для управления этим кодом каждая рабочая станция имеет список, называемый ECL (Execution Control List — список управления исполнением). В этом списке для каждого источника кода определяются наборы действий, которые этот код может выполнять. Источник кода определяется по имени ID-файла, ключом которого этот код подписан, и/или сертификатором, которым был заверен этот ключ.

В качестве подписей поддерживаются как нестандартные сертификаты Notes/Domino, так и сертификаты X.509.

По умолчанию, код, происходящий из неизвестных (не перечисленных в ECL) источников, вообще не исполняется.

Администратор домена Notes может создать централизованный ECL, с которым ECL рабочих станций будут синхронизоваться при каждой регистрации в домене. Можно даже запретить пользователям самостоятельно изменять ECL своих станций.

При реализации "песочницы" и интерфейса управления ею разработчики должны найти компромисс между противоречивыми требованиями: удобством пользователя, простотой управления полномочиями, уровнем детализацией этих полномочий, функциональностью приложения, в данные которого внедряется "активный контент", производительностью и т. д. Существующие решения далеки от совершенства и универсальности, поэтому в данной книге реализация "песочниц" не обсуждается — хотя, разумеется, это достаточно важная и актуальная тема.

Необходимо также принимать во внимание тот факт, что интерпретатор исполняющегося в "песочнице" кода может содержать ошибки, например срывы буфера или ошибки подъема по каталогам. Поэтому исполняющийся в "песочнице" код может в действительности иметь гораздо больше полномочий, чем указано в настройках. Во время внедрения языка Java обнаруживались многочисленные ошибки такого рода в различных версиях JVM.

Xbox Linux challenge

В 2003 году именно благодаря срыву буфера была решена задача, известная как XBox Linux Challenge — запуск Linux на немодифицированной игровой приставке XBox.

Игровая приставка XBox, разработанная и производящаяся по заказу компании Microsoft, представляет собой компьютер на основе процессора x86 с шиной расширения PCI и жестким диском, но нестандартным (не совместимым с IBM PC) BIOS. Установленная на диске ОС, основанная на ядре Windows 2000, разрешает запуск только подписанных приложений, а нестандартный BIOS не допускает загрузки на приставке неподписанных ОС.

Подпись в данном случае используется не столько для защиты от троянского кода, сколько для решения задач DRM (Digital Rights Management — цифровое управление правами или, как это предлагает расшифровывать Ричард Столлмэн, Digital Restrictions Management — цифровое управление ограничениями). Microsoft контролирует единственный ключ, подпись которым обеспечивает запуск приложений на приставке, поэтому все производители ПО (главным образом, игр) для этой приставки могут распространять свои программы, только заплатив Microsoft за подпись. Это обеспечивает некоторые преимущества производителям игр, в частности потому, что не допускает распространения "пиратских" копий игр со взломанными механизмами защиты от несанкционированного копирования, но, самое главное, благодаря такой схеме Microsoft выгодно субсидировать производство самих приставок и продавать их гораздо дешевле, чем стандартные PC-совместимые компьютеры сопоставимой комплектации.

В 2003 году основатель компании Lindows Майкл Робертсон объявил приз в 100 000 долларов тому, кто запустит Linux на немодифицированной XBox. Первым претендентом на этот приз оказался хакер под псевдонимом Habibi-XBox, который внедрил загрузчик Linux в образ процесса игры "007 — Agent Under Fire" через срыв буфера в коде, который занимался чтением файлов сохранения [news.com.com Xbox challenge].

13.8.3.3. Черви и боты

Там на неведомых дорожках
Следы невиданных зверей.

A. Пушкин

Но работать без подручных — может, грустно, может — скучно.
Враг подумал, враг был дока, — написал фиктивный чек.
И где-то в дебрях ресторана гражданина Енифана
Сбил с пути и с панталыку несоветский человек.

B. Высоцкий

Черви (worm) представляют собой троянский код, способный размножаться без участия пользователей системы. Заражению червями подвержены лишь многопоточные ОС с более или менее развитыми сетевыми сервисами. Заразив систему, червь запускает себя в качестве фонового процесса и начинает атаку других систем, используя фиксированный набор известных проблем в их системах безопасности, чаще всего срывы буфера в сетевых сервисах.

Нередко червь в обязательном порядке заражает системы, которые доверяют зараженной или просто используют общую с нею базу учетных записей — например, заразив одну из систем домена Windows NT от имени администратора системы, червь может штатными средствами установить себя на все остальные системы того же домена.

Первый из известных червей — неоднократно упоминавшийся в этой главе червь Морриса, созданный в 1987 году, — использовал несколько каналов распространения, в том числе:

- срыв буфера в сетевом сервисе finger (в настоящее время этот сервис практически не используется, потому что спаммеры применяли его для получения списков почтовых адресов пользователей);
- отладочную заднюю дверь в единственном распространенном в то время почтовом сервере Sendmail;
- подбор пароля методом словарной атаки;
- межмашинное доверие rlogin/rsh.

Червь поражал миникомпьютеры VAX и рабочие станции Sun, основанные на микропроцессоре Motorola 68000, работавшие под управлением BSD Unix. Получив доступ к системе, червь запускал командный интерпретатор, закачивал на заражаемую систему свои исходные тексты на языке C, компилировал их и запускал.

Буквально за считанные минуты червь поразил практически весь тогдашний Интернет. Быстрому обнаружению червя способствовал тот факт, что в коде, который определял, не является ли система уже зараженной, была ошибка,

поэтому червь многократно заражал одни и те же узлы сети и сильно снижал их производительность.

Черви размножаются очень быстро. Действительно, для их размножения пользователям не надо предпринимать никаких действий, поэтому скорость распространения червя оказывается ограничена только скоростью поиска годных для заражения узлов (размер самого червя обычно относительно невелик, поэтому он может быстро передаваться даже по относительно медленным каналам связи).

В условиях, когда все или почти все узлы сети уязвимы, простые способы поиска, такие как случайное сканирование адресов, обеспечивают экспоненциальный рост количества копий червя с каждым поколением. Впрочем, по мере того как все большее количество узлов оказывается заражено, скорость размножения падает; равновесие наступает при уровне заражения, который несколько ниже 100%.

Как показывает имитационное моделирование, некоторые более сложные приемы, такие как разделение адресного пространства между потомками, могут обеспечивать рост, который даже быстрее экспоненциального, и заражение всей сети. Впрочем, для большинства практических целей это оказывается излишеством. Реальные черви, как правило, довольствуются экспоненциальным размножением — ведь оно позволяет заразить большую часть современного Интернета за несколько поколений червя.

Время между поколениями червя определяется выбором его разработчика. Агрессивно размножающиеся черви, такие как *Sapphire* (*SQL Slammer*), могут генерировать по потомку каждые несколько миллисекунд.

Столь высокая скорость размножения обусловлена тем, что код червя полностью размещается в единственном пакете UDP, и заражение не требует диалога с заражающим узлом. Червь использует срыв стека в службе монитора MS SQL Server 2000; код, запускаемый при срыве стека, представлял собой тело червя, а не первичный загрузчик, как у большинства других червей. Получив управление, червь находит адреса системных DLL, через которые он исполняет системные вызовы, и немедленно начинает размножаться, рассыпая свои копии по случайно выбранным адресам.

Несмотря на относительно небольшое количество уязвимых узлов, *Sapphire* в 2003 году поразил почти все уязвимые серверы в течение 15 минут. Червь не содержал никакого целенаправленно вредоносного кода, но сами его запросы на размножение оказались столь многочисленными, что парализовали многие магистральные каналы Сети и корпоративные сети [www.wired.com/slammer_pr].

Другие черви размножаются медленнее. Это позволяет им потреблять небольшую долю ресурсов зараженного узла, так что инфекция относительно

долгое время остается незамеченной. При этом пандемия развивается в течение нескольких часов или даже суток.

В современных условиях пандемии относительно быстро подавляются (в том числе за счет своевременно выпущенных "заплаток" к ошибкам, которые червь использует для распространения), но после них всегда остается жизнеспособная популяция червя, который поражает вновь подключаемые к Сети уязвимые компьютеры.

Средний срок жизни Windows XP SP1, подключенной к Интернету с настройками по умолчанию, определяемый как время от установления соединения до заражения первым червем или ботом, в 2004—2005 годах измерялся несколькими секундами — это подтверждается как официально опубликованными данными, например опытами проекта Honeynet [www.honeynet.org], так и моим личным опытом. Этого времени заведомо недостаточно, чтобы скачать обновления с Windows Update даже по самому быстрому сетевому каналу.

Для Windows XP SP2 ситуация на время подготовки второго издания книги была несколько лучше; впрочем, насколько я понимаю, это обеспечивается не столько за счет того, что все опасные ошибки были исправлены, сколько за счет того, что в SP2 при подключении к Интернету по умолчанию фильтруются все входящие сетевые соединения. По данным того же проекта Honeynet, срок жизни Windows XP SP2 составляет около двух часов. Этого уже может быть достаточно для скачивания последних патчей с Windows Update. Впрочем, "срок жизни" в данном случае обозначает среднее, а вовсе не гарантированное время жизни, так что все-таки лучше скачивать патчи для новой системы с защищенного компьютера. К сожалению, структура сайта Windows Update такова, что скачивание патчей для версии системы, отличающейся от вашей, представляет собой нетривиальную исследовательскую задачу, которая не по силам подавляющему большинству сертифицированных администраторов Windows, не говоря уж о рядовых пользователей.

Те же самые данные имитационного моделирования показывают, что подобные явления не могли бы развиваться, если бы уровень разнообразия вычислительных систем в Сети был выше, чем сейчас. Если большинство систем в сети неуязвимы для каждой конкретной атаки (а в гетерогенных сетях так оно обычно и есть), пандемия может вообще не развиться, особенно если червь не прибегает к агрессивному размножению или к целенаправленному поиску уязвимых узлов (как это делал червь phpBB, рассматривавшийся в разд. 13.7.2). При этом деятельность червя может быть легко обнаружена по необычно большому количеству обращений к несуществующим сетевым адресам и недоступным сетевым сервисам.

Боты аналогичны червям по способам размножения: они поражают уязвимые системы через срывы буфера и другие аналогичные ошибки. В отличие от

червя, боты не автономны: заразив систему, они регистрируются на некотором центральном сервере и начинают принимать команды от этого сервера. В качестве протокола общения с сервером, распространенные боты чаще всего используют IRC (Internet Relay Chat — сетевой релейный чат, т. е. средство онлайн общения, использующее релейные серверы; стандарт протокола IRC определяется документом RFC 1459) или его нестандартные варианты, специально модифицированные для управления большими количествами ботов. Владелец ботнета может затем управлять своими ботами с помощью относительно простого командного языка. Многие ботнеты допускают обновление программного обеспечения бота, в том числе добавление новых модулей, как функциональных, так и служащих для размножения бота.

Размер ботнета обычно определяется не способностями бота к размножению, а способностями владельца ботнета его контролировать. По данным проекта Honeynet, в рамках которого была организована "ловля" ботов на контролируемые компьютеры и дальнейшее прослушивание трафика в управляющем канале, основная масса ботнетов невелика по размеру, от нескольких десятков до нескольких сотен компьютеров, и управляет малоквалифицированными лоботрясами подросткового и позднеподросткового возраста. Специалисты Honeynet наблюдали, как владелец, введя ошибочную команду, потерял контроль над относительно крупным ботнетом. Впрочем, удавалось заметить и деятельность действительно крупных ботнетов, состоящих из десятков и сотен тысяч компьютеров, и, по-видимому, управляемых квалифицированными специалистами. Доступные в Сети исходные тексты модифицированных серверов IRC рассчитаны на контроль над ботнетами из нескольких миллионов машин.

Функциональные модули ботов отличаются большим разнообразием. Их функции включают в себя запуск червей и вирусов, перехват паролей (наибольшей популярностью пользуются пароли для платных порносайтов), поиск регистрационной информации для коммерческих и условно-бесплатных программ, phishing (кражи номеров кредитных карт и другой банковской информации), рассылка почтового спама. По данным из сетевых форумов, на черном рынке ботнет из тысячи компьютеров, пригодный для рассылки спама, можно продать за несколько сотен долларов или евро.

Потребность в ботнетах для рассылки спама определяется тем фактом, что основным средством по борьбе со спамом в современном Интернете являются централизованные черные списки. Узел или сеть, замеченные в рассылке спама, быстро (обычно в течение нескольких часов) включаются в черный список, и другие почтовые релеи перестают принимать у этого узла какую бы то ни было почту. Поэтому спаммеры не могут использовать одни и те же почтовые серверы в течение сколько-нибудь длительного времени. Ботнет представляет для спаммера идеальное средство — довольно большую сеть компьютеров, каждый из которых по отдельности мало ценен. Тысячи ком-

пьютеров, находящихся в разных сетях, может хватить на несколько недель работы.

Черви, вирусы, боты и неразмножающиеся крупнотиражные троянские программы, такие как SpyWare и AdWare, представляют собой наиболее важную проблему для современного Интернета. Основным средством борьбы с ними являются антивирусные пакеты и их специализированные версии для борьбы со AdWare/SpyWare; впрочем, очевидно, что если ваша сеть подвержена вирусным атакам, а особенно если единственной защитой вашей сети от них служит антивирусный пакет, способный только находить известные вирусы, она будет столь же уязвима и для троянских программ, специально разработанных с целью атаки на данные вашей организации. При этих условиях вашу систему безопасности следует признать абсолютно непригодной.

Ранее приводилась одна важная причина, по которой антивирус не может быть эффективен даже против крупнотиражных троянских программ: он не может защитить от сложных программ, внедряющихся в ядро системы и использующих stealth-технологии.

Кроме того, важно понимать, что производители антивирусных пакетов обычно на шаг, а иногда и больше, отстают от разработчиков вирусов, так что обновление баз антивируса часто появляется лишь после пика пандемии.

Поэтому, хотя установкой антивирусных фильтров, как и любым другим возможным эшелоном защиты, не следует пренебрегать, рассчитывать на них как на основное средство защиты ни в коем случае не следует. Основным средством защиты должна быть комплексная система безопасности, включающая в себя:

- продуманную структуру сети с расставленными в ключевых местах фильтрующими маршрутизаторами и другими фильтрами;
- доступ к разделяемым ресурсам, построенный по принципу минимально необходимых привилегий;
- адекватные средства мониторинга сети и узлов;
- продуманные и надежные схемы восстановления узлов сети после аварий;
- своевременную установку "заплат";
- и ряд других мероприятий.

13.9. Практические рекомендации

Ясность мысли. Реалистичный подход. Что очень важно для работы, подобной нашей.

T. Пратчетт

Некоторые из приводимых рекомендаций уже упоминались в тексте, но здесь мы постараемся собрать их воедино.

Пользователь не должен иметь доступа к данным более того, что требуется для исполнения им служебных обязанностей. Это позволяет минимизировать вред от исполняемых пользователями троянских программ и прямых проникновений в систему от имени этого пользователя, а также вред, который пользователь может причинить сам, как сознательно, так и по ошибке. Кроме того, работа с ограниченным подмножеством данных удобнее для пользователя и иногда приводит к повышению производительности.

Когда это возможно, доступ к разделяемым данным следует устанавливать не индивидуальным пользователям, а группам. Список групп и структура их вложенности должны соответствовать иерархической структуре организации и существующим в ней функциональным группам должностей. Многие современные ОС поддерживают иерархические структуры учетных записей. Этим нужно пользоваться. Как правило, иерархия БД учетных записей должна соответствовать структуре организации.

Умопостигаемая, хорошо инятно документированная организационная структура оказывает администратору значительную помощь в проектировании структуры групп и ACL, а хорошая документация бизнес-процесса необходима разработчику приложений для поддержки бизнеса, в том числе и для проектирования модели безопасности.

Везде, где это не приводит к чрезмерным накладным расходам, следует применять шифрованные протоколы передачи данных. Данные, хранящиеся на компьютерах за пределами здания компании, а особенно на домашних и переносных компьютерах, следует шифровать в обязательном порядке.

Защита данных практически не имеет смысла без защиты самой системы и прикладного программного обеспечения: если злоумышленник имеет возможность модифицировать код системы или прикладной программы, он может встроить в него троянские подпрограммы, осуществляющие несанкционированный доступ к данным.

Доступ к коду приложений и системы для модификации должен предоставляться только техническому персоналу, занимающемуся поддержкой и установкой обновлений этих приложений. Это позволяет защититься не только от запускаемых пользователями троянских программ (особенно вирусных), но и от ошибочных действий пользователей.

Доступ к конфигурации ОС и прикладных программ также должен требовать высоких привилегий. В идеале, пользователю следует иметь доступ только к настройкам пользовательского интерфейса ОС и приложений.

Когда это возможно, следует защищать каталоги данных от исполнения — это также может помочь защитить пользователей от некоторых способов внедрения троянских программ.

Все ресурсы системы, которые могут резервироваться пользователями, должны квотироваться. Каждый неквотируемый общедоступный ресурс является потенциальной точкой DoS.

Каждый активный серверный или сервисный процесс, исполняемый в системе, может содержать ошибки и, таким образом, является потенциальной точкой атаки. Следовательно, все сервисы, которые не нужны непосредственно для работы системы или реально используемых прикладных программ, необходимо остановить. Это может также привести к некоторому повышению производительности.

Для каждого сетевого сервиса необходимо решить, следует ли предоставлять доступ к нему всем (всей сети компании или даже всему Интернету). Доступ к большинству сервисов, как правило, можно и следует ограничить. Самые сетевые сервисы необходимо запускать от имени учетных записей с минимально необходимыми привилегиями.

Во всех случаях, когда это возможно, следует делать операционную среду гетерогенной — различные ОС и приложения имеют различные (и, как правило, непересекающиеся) наборы проблем с безопасностью, поэтому сложность взлома гетерогенной среды резко возрастает. Особенно это справедливо для автоматических и полуавтоматических взломщиков, таких как черви и боты.

Необходимо продумать схему мониторинга сети и узлов сети. Особенное внимание следует уделить схемам хранения лог-файлов ОС и приложений. Если это возможно, следует размещать логи не на той машине, которая их ведет, а на другой, что чрезвычайно затруднит подчистку логов после взлома. Необходимо также продумать схему ротации логов, т. е. уничтожения устаревших данных.

Следует продумать схему полного или частичного восстановления всех серверов после аварий — это полезно не только при вирусных атаках, но и при отказах аппаратуры и во многих других случаях. Мне неудобно напоминать о необходимости регулярного и надежного резервного копирования, но многие практикующие администраторы им пренебрегают. Необходимо также понимать, что схема восстановления включает в себя не только резервное копирование, но и восстановление данных. Мне доводилось слышать о корпоративных стандартах, в соответствии с которыми каждый новый сервер после недели работы должен быть стерт и восстановлен с резервной копии — без этого сервер не считается принятым в регулярную эксплуатацию.

Поставщики ОС и приложений нисколько не заинтересованы в том, чтобы в их продуктах существовали известные ошибки, особенно способные привести к проблемам с безопасностью. Поэтому все сколько-нибудь приличные поставщики ПО, как свободно распространяемого, так и коммерческого, публикуют обновления к своим продуктам — "заплаты" или, как говорят практи-

кующие русскоязычные администраторы, "патчи" (patch — заплата). Эти обновления содержат исправления обнаруженных ошибок и рекомендации по обходу ошибок известных, но еще не исправленных. Системный администратор должен следить за этими публикациями — что, впрочем, не следует понимать как рекомендацию немедленно устанавливать на промышленно эксплуатируемые серверы самые последние "заплаты".

Тем не менее необходимо помнить, что многие из пандемий последних лет не произошли бы, если бы системные администраторы были более расторопны: черви phpBB, Sapphire и некоторые другие распространялись через дыры, которые были известны за несколько месяцев до пандемии, и для которых были доступны "заплаты" от производителей уязвимых программ.

Вообще, необходимо также отметить, что интервал времени между выпуском публичного патча и его установкой большинством пользователей представляет собой самое "злачное" время для разработчиков malware. Поиск ошибок во всем объеме кода системы представляет собой сложную и не очень благодарную задачу — в почти равной мере это относится как к системам с открытым, так и системам с закрытым исходным кодом. В то же время, проанализировать выпущенный производителем патч, найти отличия между патченным и оригинальным кодом и обнаружить ошибку в отличающемся коде может любой ПТУшник. Самые громкие вирусные атаки последних лет происходили через ошибки, обнаруженные именно таким способом.

При принятии решения об эксплуатации той или иной программной системы — операционной, прикладной или среды разработки — необходимо ознакомиться с политикой поддержки, которую предоставляет ее поставщик.

Важным источником информации об известных проблемах с безопасностью (как ошибках в коде ОС и приложений, так и распространенных ошибках при установлении прав администратором) являются списки рассылки и Websites [www.cert.org], [www.ntbugtraq.com] и ряд других, а также специальные списки рассылки, поддерживаемые поставщиками ПО.

Хотя антивирусные пакеты и не являются адекватным средством защиты вашей сети от троянских программ, их применением не следует пренебрегать, особенно в средах, где активно используются приложения и ОС фирмы Microsoft. Лучше иметь хоть какую-то защиту, чем вообще никакой. В качестве дополнительного эшелона защиты антивирусный пакет также может оказаться полезен.

Важной частью работы администратора и/или специалиста по безопасности является взаимодействие с пользователями. Такое взаимодействие должно быть двусторонним — пользователи должны понимать не только сами требования системы безопасности, но и то, чем они обусловлены (обычно это повышает уровень исполнения правил), а администратор должен понимать, как

именно эти правила исполняются. Если некоторые из правил систематически нарушаются, необходимо понимать, с чем это связано, и искать решения проблем, которыми эти нарушения обусловлены.

Вопросы для самопроверки

1. Согласны ли вы с утверждением, что обеспечение безопасности — это не действие, а процесс? Как вы можете обосновать это утверждение?
2. Почему так важен принцип минимально необходимых привилегий?
3. Какие схемы аутентификации пользователей вы знаете?
4. Каковы преимущества и недостатки парольной аутентификации?
5. Что такое словарная атака и как от нее защищаться?
6. Почему в большинстве современных систем аутентификации (как уровня ОС, так и прикладных) пароли пользователей не доступны системному администратору? Какими средствами это обеспечивается?
7. Какие известны альтернативы парольной аутентификации? Что ограничивает применение этих альтернатив?
8. Приведите примеры полномочий в модели обеспечения безопасности систем семейства Unix.
9. Почему утверждается, что все практически применяемые системы безопасности прямо или косвенно основаны на полномочиях?
10. Почему необходимо сокращать списки управления доступом? Приведите основные применяемые на практике приемы такого сокращения.
11. Почему схема наследования списков управления доступом в Novell Netware допускает фильтрацию всех прав, кроме права супервизора?
12. Какие преимущества, кроме сокращения ACL, дает использование групп?
13. Что такое кольца доступа? Почему сокращенные ACL систем семейства Unix, несмотря на явное сходство с кольцами доступа, все-таки не являются ими?
14. Если вы постоянно работаете на вашем компьютере с привилегиями локального администратора, то почему? Что мешает вам создать учетную запись с ограниченными правами для обычной работы, а из-под локального администратора выполнять только те операции, для которых это действительно необходимо?
15. Почему в этой главе утверждается, что любой ресурс, на который нет квот, является потенциальной точкой отказа сервиса?

16. Приведите примеры способов, которыми злоумышленник мог бы с выгодой для себя использовать атаку отказа сервиса.
17. Что такое срыв буфера? Почему срывы буфера в стеке считаются наиболее опасными?
18. Верно ли, что срывы буферов, размещенных не в стеке, не могут приводить к исполнению кода? Верно ли (как это, например, утверждается в докладе [Karger/Shell 1974]), что срыв стека допускает исполнение кода, только если стек растет вниз, как у PDP-11, VAX, x86? Как использовать срыв стека, растущего вверх, для исполнения кода?
19. Верно ли, что защита сегмента данных от исполнения обеспечивает надежную защиту от исполнения кода посредством срывов буфера?
20. Что такое ошибка внедрения SQL? Как от нее следует защищаться?
21. Чем опасна ошибка подъема по каталогам?
22. Почему среди всех практически реализуемых видов атак наиболее опасным считается внедрение троянского кода?
23. Почему в этой главе утверждается, что приложения для внутреннего использования гораздо более опасны с точки зрения внедрения в них злонамеренного кода разработчиками, чем крупнотиражные программы и даже чем программы, разработанные на заказ? Согласны ли вы с приводимой в этой главе аргументацией в защиту этого утверждения?
24. Являются ли открытые исходные тексты надежной защитой от "закладок" в коде программы? Если нет, то почему?
25. Рассмотрите ваш домашний компьютер. Хорошо ли он защищен от внедрения троянских программ? Что можно сделать для улучшения его защиты? Знали ли вы о перечисленных мерах до прочтения этой главы, и если да, то почему вы их не предпринимали?
26. Какие из файловых систем вашего домашнего компьютера следовало бы защитить от исполнения? На каких файловых системах такую защиту следовало бы включать по умолчанию, но с возможностью при необходимости ее выключить?

ПРИЛОЖЕНИЕ 1

Обзор современных ОС

Не помнящие прошлого обречены повторять его.
Дж. Сантаяна

В данном приложении приводится краткое изложение истории семейств современных ОС и обзор архитектур наиболее важных представителей каждого из семейств. В отличие от остальной книги, при выборе тем для обсуждения я руководствовался не интересностью или поучительностью конкретных архитектурных концепций, а распространенностю и практической важностью входящих в то или иное семейство программных продуктов.

П1.1. MVS, OS/390, z/OS

Первые две ОС этого семейства вышли в 1966 году, вскоре после анонса аппаратной архитектуры System/360. Это были PCP (Primary Control program — первичная управляющая программа) и DOS/360 (Disk Operating System). Архитектура обеих систем была типична для вычислительных систем второго поколения — это были пакетные мониторы, рассчитанные на работу одной прикладной программы без защиты памяти (первые компьютеры серии System/360 не имели диспетчеров памяти).

В 1967 году были выпущены версии PCP: MVT (Multiprogramming with a Variable number of Tasks — многопрограммная [система] с переменным числом задач) и MFT (Multiprogramming with a Fixed number of Tasks — то же, но с фиксированным числом задач). Обе системы реализовывали вытесняющую многозадачность в едином адресном пространстве — MFT использовала разделы памяти, а MVT — относительную загрузку с базовым регистром. Позднее к MVT была добавлена подсистема работы с несколькими терминалами в режиме разделения времени TSO (TimeSharing Option — возможность разделения времени), ASP (Asymmetric Multiprocessing System — асимметричная многопроцессорность) и ряд других прикладных подсистем.

В 1972 году, после появления машин System/370 с диспетчером памяти, была выпущена переходная система OS/SVS (Single Virtual Storage — единая виртуальная память), которая позволяла использовать страничную подкачку, но не защиту заданий друг от друга. Наконец, в 1974 году была выпущена MVS (Multiple Virtual Storage — множественная виртуальная память), которая предоставляла каждой задаче собственное виртуальное адресное пространство объемом до 2 Гбайт (в System/360 и первых моделях System/370 адрес был 24-разрядным). Большая часть дополнительных подсистем MVT была включена в стандартную поставку MVS.

MVS предоставляла:

- раздельные адресные пространства;
- пакетный и интерактивный (разделение времени) запуск задач;
- вытесняющую многозадачность;
- выполнение нитей в общем адресном пространстве (нити в этой системе называются *единицей обслуживания (service unit)*);
- многопоточное ядро;
- примитивы взаимоисключения — замки (lock);
- симметричную многопроцессорность;
- динамическое подключение и отключение процессоров (как центральных, так и канальных) и их горячую замену;
- библиотеки — аналог вложенных каталогов;
- последовательные, блочные и индексно-последовательные файлы;
- отображение файлов, в том числе и индексно-последовательных, в адресное пространство (VSAM);
- систему безопасности на основе ACL;
- развитые средства системного мониторинга;
- сетевые средства — поддержку стека протоколов SNA (Standard Network Architecture).

MVS воплотила все наиболее прогрессивные функции и архитектурные концепции своего времени. MVS была основным продуктом IBM вплоть до середины 90-х, когда вышла новая версия системы — OS/390.

В OS/390 основные архитектурные принципы MVS не подверглись пересмотру [redbooks.ibm.com sg245597]. Новшества заключались в добавлении следующих возможностей:

- поддержка многомашинных кластеров (Parallel Sysplex);
- развитие сетевых средств равноправного (peer-to-peer) взаимодействия;

- поддержка сетевых протоколов семейства TCP/IP (ранее взаимодействие с сетями TCP/IP и NETBIOS осуществлялось посредством включения в состав вычислительного комплекса модуля с процессором x86 под управлением OS/2);
- совместимость с системами семейства Unix [redbooks.ibm.com sg245992] (в 1998 году OS/390 прошла набор тестов Unix 95 консорциума X/Open и получила право называться UNIX™ [www.opengroup.org xu007]).

В 1999 году, в связи с началом выпуска 64-разрядного семейства компьютеров z900, вышла 64-разрядная версия системы z/OS [www.ibm.com zOS].

Системы под управлением OS/390 и z/OS применяются главным образом в качестве серверов транзакций и СУБД масштаба предприятия и составляют становой хребет вычислительных систем большинства крупных компаний.

П.1.2. ОС для компьютеров DEC

[на мотив "Johnny B. Goode"]

Way up in the north there was a factory,
Run by Ken Olson making PDPs,
Made them by the zillions,
Sold them for a low price,
Said they weren't computers,
Just called them PDPs.

Go, go DEC go, go,
Go DEC go, go,
keep making PDPs.

R.D. Davis

Там на севере была фабрика,
Кен Olson делал на ней PDP.
Делал их зиллионами,
Продавал по низким ценам,
Говорил, что они не компьютеры,
Называл их просто PDP.

...

P. Д. Дэвис

А помнишь, Мишка, семьдесят второй,
И психодром, и сейшн в Лужниках?
Как двери вышибали головой,
Как фаны нас носили на руках?

A. Макаревич

Операционные системы для компьютеров компании DEC — TOPS-10, TENEX/TOPS-20, RT-11, RSX-11, VMS — известны многим лишь по легендам. Однако влияние этих ОС легко прослеживается в более современных системах семейств Unix и CP/M, которые рассматриваются далее в этом приложении. Влияние семейства во многом обусловлено тем, что дешевые мини-

компьютеры DEC охотно закупались университетами и использовались для обучения студентов. Впрочем, эту причину нельзя признать единственной: компьютеры DEC и разработанные для них ОС обладали значительными собственными достоинствами. Поэтому описание истории и архитектуры современных ОС, наверное, было бы неполным без рассказа об этом семействе.

Все ОС данного семейства (за исключением OpenVMS) разрабатывались в расчете на конкретную аппаратную архитектуру. TOPS-10 была предназначена для PDP-6 и PDP-10, TENEX — для модифицированной PDP-10 со специальным диспетчером памяти BBN, TOPS-20 — для DEC-20, RT-11 и RSX-11 — для PDP-11. VAX/VMS была разработана для компьютеров DEC VAX и позднее перенесена на DEC Alpha и Intel Itanium. Эта привязка к оборудованию, в конечном итоге, и привела к тому, что события на рынке аппаратного обеспечения, в конце концов, обусловили потерю рынка и практически полное забвение во многих отношениях замечательным системам.

П1.2.1 PDP-6 MONITOR и TOPS-10

Достоверной информации о происхождении названия TOPS история до нас не донесла. Буква Т в начале названия может означать как Tape (лента), потому что первые версии TOPS-10 не поддерживали диски, так и Timesharing (разделение времени). TOPS-10 известна как первая массовая ОС, поддерживающая разделение времени.

ОС первоначально разрабатывалась для 36-разрядной машины DEC PDP-6, анонсированной в 1964 году. Процессор PDP-6 имел ряд архитектурных особенностей, наиболее известная из которых — это отсутствие доступных программисту регистров. Команды имели два адресных поля, одно с разрядностью 18 бит, второе — 4 бита. Однако 4-битовое адресное поле адресовало не регистры, а первые 16 слов ОЗУ. Лишь в довольно поздних моделях PDP-10 эти 16 слов были отображены на быстрое статическое ОЗУ.

PDP-6 имела диспетчер памяти на базовых регистрах, который позволял перемещать пользовательские программы по физической памяти и защищать их друг от друга. Диспетчер памяти состоял из двух регистров, **PR** (ограничитель) и **RLR** (ReLocation Register, регистр перемещения, т. е. базовый регистр). Процессор мог работать в двух режимах, пользовательском и "исполнителем" (executive). В пользовательском режиме были запрещены остановы процессора, операции ввода/вывода и манипуляции с регистрами диспетчера памяти.

Переключение в режим "исполнителя" происходило при передаче управления на команды с недопустимым кодом операции. Коды команд 040—0100 соот-

ветствовали системным вызовам; таким образом, каждому системному вызову соответствовал свой код команды.

Первая версия ОС требовала, чтобы все исполняющиеся задачи находились в ОЗУ, но позднее, когда была добавлена поддержка магнитных дисков и барабанов, появилась возможность подкачки задач. Машина допускала подключение нескольких телетайпов или, позднее, видеотерминалов с интерфейсами "токовая петля 20 μ A" и RS232.

Операционная система для PDP-6, известная как Monitor, поставлялась вместе с исходными текстами на ассемблере. Никакой динамической загрузки модулей ядра не предполагалось, при изменении конфигурации требовалась пересборка системы из исходных текстов.

Одним из важнейших компонентов системы была консоль (CONSOLE) — интерактивный интерпретатор командного языка, который обеспечивал запуск внешних программ (как стандартных, так и созданных пользователем). Стандартные программы запускались посредством собственных команд — так, ассемблер запускался командой MACRO, а утилиты для работы с файлами — командой PIP (Peripheral Interchange Program). Пользовательские программы запускались командой RUN. В отличие от более современных ОС, интерпретатор командного языка был частью ядра системы, а не обычной пользовательской программой.

Система поставлялась с ассемблером MACRO, компилятором Fortran II, утилитой для работы с файлами PIP и строчным редактором EDITOR.

Поскольку пользователями системы были студенты и преподаватели университетов, доступность исходных текстов быстро привела к возникновению нестандартных версий системы, количество которых было сравнимо с общим количеством изготовленных компьютеров. Наиболее известная из этих ОС — это разработанная в Массачусетском технологическом институте ITS (Incompatible Timesharing System).

Собственно, именно этот эпизод в истории DEC PDP-10/20 часто приводят в качестве аргумента в дискуссиях о том, следует ли распространять программное обеспечение с исходными текстами. Действительно, если поставщик ПО не предлагает удовлетворительной поддержки, то пользователи, некоординированно осуществляющие поддержку собственными силами, могут создать множество несовместимых ветвей системы. Однако если ПО поставляется вместе с контрактом на поддержку, разработчик может попросту отказаться поддерживать модифицированные версии системы и, таким образом, предотвратить неконтролируемое ветвление. Кроме того, современные средства коммуникации, прежде всего Интернет, дают пользователям возможность координировать децентрализованную поддержку и модификацию програм-

мы. Опыт разработки ПО с открытыми исходными текстами в современных условиях показывает, что ветвления успешных проектов происходят довольно редко и, как правило, приводят к возникновению жизнеспособных продуктов. Далее в этом приложении мы увидим исторические примеры, которые демонстрируют, что закрытые исходные тексты вовсе не исключают опасности ветвления программных проектов.

PDP-6 была первой для DEC попыткой выхода на рынок больших компьютеров. Машина получилась дорогостоящей и наделенной рядом существенных эксплуатационных недостатков. Было изготовлено всего 23 компьютера, все они были проданы университетам. С коммерческой точки зрения проект оказался неудачным и DEC объявил о своем уходе с рынка 36-битных компьютеров.

Однако под давлением пользователей компания была вынуждена пересмотреть это решение, и в 1968 году был анонсирован новый процессор KA10, ставший первым из линии PDP-10. Система команд KA10 практически не отличалась от PDP-6, однако схема адресации подверглась значительным изменениям. Адрес был расширен с 18 до 22 бит, а схема трансляции усложнилась. Вместо единственного сегмента объемом до 265К слов программа могла использовать один приватный "нижний" сегмент (lower segment), соответствующий адресному пространству PDP-6, и несколько разделяемых "верхних" сегментов (high segments). В "верхних" сегментах могли размещаться разделяемые сегменты данных или разделяемые библиотеки.

Кроме усложнения адресации, был введен новый уровень привилегий, известный как User I/O (пользовательский ввод/вывод). Программы, исполняющиеся на этом уровне, не могли останавливать процессор и манипулировать регистрами трансляции адресов, но могли выполнять операции ввода/вывода.

Для поддержки пользовательских программ ввода/вывода в операционной системе был реализован класс планирования реального времени.

Производительность процессора мало изменилась по сравнению с PDP-6, но новое устройство было дешевле в производстве и удобнее в эксплуатации. Кроме того, введение разделяемых библиотек значительно повысило эффективность использования памяти. Внешние интерфейсы процессора не изменились, поэтому оказалось возможным собирать многопроцессорные комплексы, в которых совместно использовались процессоры PDP-6, KA10 и более поздние устройства. Так, последний работавший процессор PDP-6 в Университете Беркли, демонтированный в 1980 году, всю вторую половину 70-х функционировал в составе трехпроцессорного комплекса, включавшего в себя также KA10 и KL10.

Усовершенствованная версия Monitor получила название TOPS-10.

Одной из наиболее примечательных особенностей этой ОС, унаследованной от Monitor PDP-6, была организация ввода/вывода. Ввод/вывод осуществлялся асинхронно. Пользовательская программа, желающая прочитать данные, должна была запросить у системы создание кольцевой цепочки буферов. Цепочка размещается в пользовательском адресном пространстве и состоит из буферов фиксированного размера, каждый из которых имеет заголовок. В заголовке хранится счетчик элементов буфера (в зависимости от режима работы драйвера устройства или типа файла, элемент может быть как словом, так и семибитным байтом), признак занятости буфера и указатель на байт, используемый прикладной программой.

Сразу после того, как цепочка буферов связывалась с устройством или файлом, ОС начинала читать данные с устройства и заполнять буфер. После того как буфер заполнялся, система переходила к следующему, пока не кончался файл или пока не упиралась в занятый буфер.

Пользовательская программа считывает данные из буфера, увеличивая указатель на первый занятый байт. Когда буфер кончается, программа должна исполнить команду `INPUT`. Эта команда представляет собой системный вызов, который помечает текущий буфер как свободный и блокирует программу, если следующий буфер в цепочке еще не заполнен.

Как и MONITOR PDP-6, TOPS-10 не допускала динамической загрузки драйверов и других модулей ядра. При изменении конфигурации систему необходимо было "перегенерировать", т. е. пересобрать из объектных модулей.

П1.2. BBN TENEX и TOPS-20

Компания BBN Technologies и лаборатория искусственного интеллекта Массачусетского Технологического Института в начале 60-х годов XX века занимались разработками в области искусственного интеллекта, в том числе по заказам DARPA. Эти работы не завершились успехом, но привели к изобретению того, что ныне известно как страничная трансляция и страничная подкачка.

В начале 60-х годов в лаборатории искусственного интеллекта использовались 18-разрядные мини-машины DEC PDP-1. Эти устройства имели 18-разрядный адрес, теоретически допускавший до 256К слов ОЗУ, но реальные компьютеры использовали не более 64К слов памяти. Поэтому один из разработчиков Lisp-машины Дэн Мерфи предложил использовать старшие (неиспользуемые) биты адреса для индексации расширенной памяти, размещаемой на магнитном барабане. При обращении к элементу списка Lisp-интерпре-

татор просматривал старшие биты и, если это было необходимо, подкачивал элемент с барабана. Позднее Мерфи перешел на работу в компанию BBN.

После анонса PDP-6 компания BBN заинтересовалась этим компьютером и предложила расширить его аппаратным устройством страничной трансляции. DEC не проявил к этому предложению интереса, и BBN не стала покупать PDP-6. Однако когда на рынке появились более дешевые PDP-10, BBN приобрела несколько машин и оснастила их самостоятельно разработанным устройством страничной трансляции, которое занимало полную 19" стойку. Для работы с этим устройством была разработана собственная ОС, которая, впрочем, обеспечивала эмуляцию системных вызовов TOPS-10 и совместимость с прикладным ПО. Эта ОС получила название TENEX [Bobrow et al 1972].

Кроме виртуальной памяти, система известна своим командным языком, который, в частности, реализовал многие манипуляции с файлами, в том числе просмотр каталогов, копирование и переименование файлов непосредственно, а не с помощью внешней программы PIP (как это делалось в TOPS-10). В отличие от TOPS-10, командный процессор был реализован как пользовательский процесс.

Командный язык TENEX был очень многословным. Например, полная форма команды просмотра каталога выглядела так: DIRECTORY OF FILES. Пользователям, разумеется, не нравилось набирать такие длинные команды, поэтому интерпретатор реализовал автоматическое дополнение команд. Так, чтобы набрать DIRECTORY OF FILES, пользователю достаточно было набрать уникальный префикс, отличавший команду от всех остальных, и ввести символ ASCII ESC. Для обсуждаемой команды таким уникальным префиксом были символы DIR. Автоматическое дополнение было реализовано также для имен файлов.

Замечание

Автоматическое дополнение имен команд и файлов реализовано в таких современных командных интерпретаторах, как 4DOS/4OS2/4NT и bash.

Другим важным новшеством TENEX по сравнению с TOPS-10 было значительное усложнение структуры файловой системы (в частности, файлы теперь могли занимать несмежные участки диска), значительное увеличение длины имени файла и введение понятия версии файла. При обращениях к файлам номер версии указывается после расширения и отделяется от него точкой с запятой. Если номер версии не указать, то для чтения будет открыт файл с наибольшим номером версии, а для записи будет создан новый файл со следующим по порядку номером.

Преимущества страничного диспетчера памяти были быстро осознаны пользователями и BBN — по-видимому, неожиданно для владельцев компании — оказалась занята весьма прибыльным бизнесом по изготовлению и поставке

страничных диспетчеров для PDP-10. TENEX быстро стал одной из самых популярных ОС для PDP-10, но DEC так и не проявил интереса к включению страничного диспетчера BBN в стандартную поставку своих компьютеров.

Впрочем, после неудачи следующего процессора семейства, KI-10, и после того, как IBM анонсировал компьютеры линии System/370, оснащенные диспетчером памяти, здравый смысл возобладал и DEC заключил соглашение с BBN о передаче документации на диспетчеры памяти и исходных текстов TENEX. Дэн Мерфи перешел на работу в DEC и возглавил работу над новой версией ОС [Murphy 1989].

Новый процессор был реализован на микросхемах низкой степени интеграции, значительно превосходил по производительности более ранние машины линии PDP-6/10 и содержал страничный диспетчер памяти. Новая архитектура получила название DECSYSTEM-20, а ОС — TOPS-20. Окончательная версия системы была значительно переработана по сравнению с TENEX, но сохранила совместимость по системным вызовам, командному языку и формату файловой системы. Обеспечивалась также совместимость по системным вызовам с TOPS-10. Был реализован интерпретатор командного языка TOPS-10. В поставку системы был включен также ряд программ независимых разработчиков, в том числе реализованный в МТИ экранный редактор TECO.

Первые машины DEC-20 были поставлены в 1976 году. Модель оказалась относительно успешной в коммерческом отношении, но в начале 80-х руководство DEC осознало, что эти системы конкурируют с другим продуктом компании — "супермини"-компьютерами VAX. В 1983 году было принято решение о прекращении развития линии DEC-10/20, но некоторые машины продолжали эксплуатироваться до начала 90-х. Несколько компаний также пытались делать клоны DECSYSTEM-10/20. Две машины были сделаны разработчиками Xerox PARC для собственного использования. Компании Foonly и System Concepts пытались делать машины на продажу, но эти попытки закончились полным провалом. Компания XKL даже наладила серийное производство клонов DECSYSTEM-20 — машин TD-1/Toad-1, но коммерческого успеха этот проект также не имел.

Прототип F-1, разработанный Foonly, известен тем, что на нем производились расчеты для "Tron" — первого полнометражного фильма, использовавшего компьютерную графику. По-видимому, название фильма связано с одноименной мнемоникой команды PDP-10/20 (Test Right-halfword Ones and skip if Not masked).

Несколько работающих машин под управлением TOPS-20 сохранилось до сегодняшнего дня. Так, XKL Toad-1, принадлежащая одному из основателей Microsoft Полу Аллену, даже доступна по протоколу Telnet (<telnet://xkleten.paulallen.com>).

П1.2.3. RT-11

Во второй половине 60-х годов успех IBM System/360 продемонстрировал преимущества архитектур с байтовой адресацией и разрядностью слова, кратной восьми битам. В компании DEC было предпринято две попытки разработать такую машину — отвергнутый проект PDP-X и PDP-11.

16-разрядная машина PDP-11 имела двухадресную систему команд, восемь регистров общего назначения (в том числе указатель стека и счетчик команд), адресацию с точностью до байта, ортогональный набор режимов адресации (любой из восьми доступных режимов мог использоваться во всех командах и с любым из регистров) и отображенный на память ввод/вывод. Младшие модели серии не имели диспетчера памяти и физическое адресное пространство совпадало с логическим. Старшие модели были оснащены сегментным диспетчером памяти с раздельными адресными пространствами для пользовательского и системного режимов, и допускали адресацию до двух (а позднее и до четырех) мегабайт ОЗУ.

Архитектура оказалась весьма успешной. Линия PDP-11 включала в себя широкий спектр компьютеров разной производительности, от мощных "супермини"-компьютеров до настольных микрокомпьютеров и встраиваемых вычислительных модулей. Клоны PDP-11 (как "цельнодраные", так и оригинальные разработки) производились в странах советского блока под названиями СМ (Серия Малых ЭВМ) и "Электроника" (впрочем, последние компьютеры этих серий, СМ-1700 и Электроника-85, были попытками воспроизвести VAX).

Первые компьютеры серии делались на основе микросхем малой степени интеграции и оснащались ОЗУ на основе ферритовых сердечников. Постепенно происходил переход к микросхемам высокой интеграции и динамическому ОЗУ, а в начале 80-х появились микропроцессорные реализации LSI-11. Последние модели серии — 11/93 и 11/94 — были анонсированы в 1990 году и имели процессоры с тактовой частотой 18 МГц. Производство этих машин было прекращено в 1997 году, лицензии на производство микросхем J-11 и операционные системы были переданы ирландской компании Mentec, которая до сих пор производит платы на основе PDP-11, главным образом для встраиваемых приложений.

Советские клоны последних моделей LSI-11, микросхемы 1806ВМ2, производятся на заводе ОАО "Ангстрем" до сих пор [www.angstrom.ru mk-mp].

Первая ОС, с которой поставлялись компьютеры PDP-11, в 1970—1973 годах называлась DOS/BATCH-11. История сохранила о ней мало достоверных сведений. После 1973 года она была быстро вытеснена RT-11.

RT-11 представляла собой компактную однопользовательскую ДОС с незащищенной памятью, простой файловой системой и асинхронным вводом/вы-

водом. Существовали версии системы SJ (Single-Job, однозадачная) и FB (Foreground-Background, [задачи] первого и второго плана), которая поддерживала кооперативную многозадачность для двух задач.

Для реализации системных вызовов использовалась команда EMT (EMulated Trap); в PDP-11 эта команда имеет операнд, который в RT-11 обозначал номер системного вызова. В поставку системы входил ассемблерный файл, в котором для каждого системного вызова было соответствующее макроопределение. Это макроопределение позволяло одной директивой ассемблера сформировать блок параметров, произвести вызов и сделать постобработку результатов, если она была необходима. Многие документы описывают системные вызовы RT-11 через формат этих макроопределений, а не через EMT-коды.

Механизм асинхронного ввода/вывода был значительно усовершенствован по сравнению с TOPS-10. Для чтения система предоставляет три формы системного вызова: .READ, .READC и .READW. .READW возвращает управление после физического завершения операции чтения. .READ и .READC возвращают управление немедленно. При использовании .READ система явно не оповещает задачу о завершении операции, но задача может подождать готовности данных, выполнив системный вызов .WAIT. При использовании .READC задача должна указать функцию-обработчик, которая будет вызвана в момент завершения операции.

Из необычных архитектурных решений следует упомянуть оверлейную реализацию ядра системы: ядро состояло из двух модулей, резидентного монитора и оверлейного модуля USR. Программа могла запросить дополнительную память, при этом USR выгружался из ОЗУ, а занятая им ранее память передавалась программе. Однако USR использовался для реализации ряда системных вызовов. Например, системные поиска в каталоге и создания файла требовали наличия USR в ОЗУ, а вызовы чтения из файла и записи в уже открытый файл — нет. При исполнении вызова, требовавшего USR, система сбрасывала расширенную область памяти программы на диск, загружала USR, исполняла вызов и загружала образ расширенной памяти обратно.

Если задаче требовалось последовательно сделать несколько системных вызовов, требовавших USR, задача могла выполнить системный вызов .LOCK, который на некоторое время (до вызова .UNLOCK) блокировал USR в памяти. Разумеется, в течение этого времени задача не должна была обращаться к своей расширенной памяти.

Еще одной интересной особенностью была возможность делать произвольные системные вызовы из обработчиков прерываний. В действительности, ассемблерный макрос .SYNCH, посредством которого это делалось, содержал внутри себя возврат из прерывания. Но перед возвратом код макроса устанав-

ливал в системную очередь запрос на вызов оставшейся части кода обработчика после того, как ядро выйдет из нереентерабельной части. Но для программиста это было практически прозрачно и выглядело именно как обращения к системе из обработчика прерывания.

Первые версии системы поставлялись с командным интерпретатором, похожим на TOPS-10. В частности, операции над файлами требовали запуска внешней программы PIP, как и в TOPS-10. В начале 80-х для RT-11 был реализован новый командный интерпретатор, предоставлявший подмножество DCL (DEC Command Language, командный язык DEC, разработанный на основе командного языка TENEX и использовавшийся в RSX-11 и VMS; подробнее этот язык описывается в следующем разделе).

Операционная система поставлялась с исходными текстами на ассемблере. Первоначально добавление новых драйверов в конфигурацию системы требовало пересборки ядра системы. Позднее, насколько я знаю — уже в версии 4, — была добавлена возможность динамической загрузки драйверов.

Линия RT-11 была весьма популярна, главным образом из-за своей неприхотливости и малых требований к оперативной памяти. На основе RT-11 компанией S&H Computer Systems была разработана система разделенного времени TSX-11, которая обеспечивала запуск прикладных программ для RT-11 в многозадачной многотерминальной среде. Для TSX-11 существует даже реализация протокола TCP/IP и Telnet.

Тем не менее эту линию ОС следует признать тупиковой — наиболее значительно влияние другой ОС для PDP-11, RSX-11, которую мы рассмотрим в следующем разделе. Пожалуй, наиболее заметный след влияния этой ОС на современные системы — это тот факт, что командный язык COMMAND.COM DOS/Windows 9x и CMD.EXE OS/2 и Windows NT основан именно на урезанной версии DCL RT-11, а не на "настоящем" DCL.

П1.2.4. RSX-11

Непосредственным предшественником RSX-11 следует считать ОС RSX-15 для DEC PDP-15. Первая машина из этого семейства была разработана в 1969 году и была первой из 18-разрядных мини-машин DEC, реализованных на интегральных микросхемах. PDP-15 использовалась во многих промышленных и военных приложениях, в которых требовалось жесткое реальное время. ОС RSX-15 представляла собой многозадачную систему с незащищенной памятью, развитой системой приоритетов процессов и асинхронным вводом/выводом. Для одновременной загрузки нескольких программ в общую память использовались разделы памяти.

ОС поставлялась в двух основных конфигурациях — Advanced Software Monitor и Real-Time Monitor. Advanced Software Monitor использовался для разработки, отладки и сборки программного обеспечения, и представлял собой многотерминальную интерактивную систему разделенного времени. Real-Time Monitor представлял собой минимальную конфигурацию системы, предназначенную для запуска приложений на машинах с небольшим объемом памяти.

Дэн Бревик утверждает [Brevik 2003], что эта ОС основывалась на более старых малоизвестных разработках для промышленных компьютеров RW-300, сделанных еще в конце 50-х годов XX века.

Разработка аналога RSX-15 для PDP-11 началась с версий RSX-11/A, B и C. Эти системы представляли собой простые многозадачные ядра, загружавшиеся из-под DOS/BATCH-11. Версии A и C не поддерживали работу с диском, версия B использовала для работы с диском системные вызовы DOS/BATCH.

RSX-11/D представляла собой полноценную ОС с собственным загрузчиком, собственной файловой системой и рядом весьма продвинутых возможностей. Однако ее потребности в оперативной памяти были слишком большими. В 1973 году перешедший на работу в DEC Дэвид Катлер переписал ядро RSX-11 практически с нуля, в целом сохранив архитектуру и совместимость по системным вызовам, но значительно уменьшив размер. Собственно, именно эта версия (RSX-11/M) и ее усовершенствованный вариант RSX-11/M-Plus и известны большинству пользователей.

RSX-11/M представляет собой многозадачную ОС разделенного времени, способную работать как на процессорах с диспетчером памяти (при этом используются все преимущества защищенной памяти, в том числе сегментная подкачка и разделяемые библиотеки), так и на машинах с незащищенной памятью. Система является многопользовательской не только в том смысле, что она поддерживает много терминалов, но и в том смысле, что все терминальные сессии и все задачи имеют идентификатор пользователя, от имени которого они исполняются. При этом же сохранялся класс планирования реального времени и остальные возможности, необходимые для систем жесткого реального времени (миллисекундные таймеры, возможность блокировать сегменты программы в ОЗУ).

Как и в RT-11, для реализации системных вызовов использовалась команда EMT, но ее операнд должен был быть всегда равен 0377. Собственно номер системного вызова передавался в младшем байте первого слова блока параметров. Как в RT-11, для всех системных вызовов существовали ассемблерные макроопределения, обеспечивающие формирование блока параметров.

RSX-11/M имеет сложную файловую систему с поддержкой вложенных каталогов, версий файлов и структурированных файлов. Файлы подразделяются

на двоичные (неструктурированные) данные, файлы с записями переменной длины (как правило, текстовые файлы) и индексно-последовательные файлы. Метаданные файловой системы размещаются в динамической структуре, которая сама присутствует в корневом каталоге тома в виде файла INDEX.SYS.

Наиболее известной особенностью RSX-11/M является организация асинхронного ввода/вывода на основе флагов событий и AST (Asynchronous System Trap), которая вкратце обсуждалась в *разд. 10.7*.

Знаменита также система оверлейной загрузки. Машины PDP-11 с диспетчером памяти допускали не более 64 Кбайт виртуальной памяти на процесс, но физическое адресное пространство могло достигать 4 Мбайт. Чтобы воспользоваться расширенной памятью, программист должен был собрать программу как оверлейную. Нулевой сегмент программы всегда был резидентным, но другие сегменты можно было указать в качестве оверлейных. В разные моменты времени в такой сегмент отображались различные модули программы. При этом, если в системе хватало физического ОЗУ, перекрытие сегментов не обязательно выгружались из памяти, поэтому производительность оверлейной программы могла быть вполне удовлетворительной.

Стандартный редактор связей RSX-11 имел средства для сборки оверлейных программ и даже средства для оптимизации распределения объектных модулей по оверлеям.

RSX-11/M имела командный язык, аналогичный TOPS-10, т. е. использовавший для работы с файлами внешнюю программу PIP. В документации этот язык известен под названием MCR (Monitor Control Routine — мониторная управляющая программа). В RSX-11/M-Plus в качестве командного языка был представлен DCL.

DCL (DIGITAL Command Language — командный язык [компании] DIGITAL) — язык, разработанный в районе 1976—77 годов комитетом, который был собран компанией DEC. Перед этим комитетом была поставлена задача согласовать командные языки TENEX/TOPS-20 и вариантов MCR, чтобы облегчить пользователям переход между различными версиями ОС. В итоге наибольшее влияние на результаты работы комитета оказала команда TOPS-20, так что результирующий командный язык был гораздо более похож на TENEX/TOPS-20, чем на MCR.

Команды DCL состоят из "глагола" (verb), т. е. собственно имени команды, и дополнительных аргументов — опций и параметров.

"Глаголы" представляли собой полные слова английского языка. Так, список файлов в текущем каталоге выводится командой DIRECTORY. Однако вместо автоматического дополнения команд было решено разрешить использовать

сокращения. Так, вместо DIRECTORY можно написать любой префикс, уникально идентифицирующий эту команду, например DIR или даже DT.

Опции обычно представляли собой полные слова английского языка и также допускали сокращение. Чтобы правильно обработать возможные опции, интерпретатор командного языка должен был знать, какие опции вообще бывают. Поэтому определение каждой команды задавалось в специальном файле CLD (Command Language Definition — определение командного языка), см. пример П1.1.

Пример П1.1 Определение команды DCL

```
DEFINE VERB FOO
  IMAGE "SYS$SYSEXEC:FOO.EXE"
  QUALIFIER BAR
```

При всех этих возможностях минимальные требования RSX-11/M к объему ОЗУ равны 32 Кбайт. Доступно также минимальное ядро RSX-11/S, аналогичное Real-Time Monitor RSX-15, и ряд специализированных версий системы.

П1.2.5. VAX/VMS

Уже к середине 70-х годов XX века стало очевидно, что 64 Кбайт ОЗУ для многих приложений недостаточно (для сравнения, 18-разрядные мини-машины позволяли адресовать 256К слов ОЗУ). Разработка 32-разрядного наследника линии PDP-11 началась в 1975 году. Первоначально предполагалось простое расширение регистров и адресного пространства PDP-11, но окончательный дизайн, представленный в 1977 году под названием VAX (Virtual Address Extended — расширенный виртуальный адрес), сильно отличался от предковой архитектуры. Первоначально линия называлась VAX-11, чтобы подчеркнуть преемственность с популярной линией PDP-11, но позднее цифры 11 были отброшены, и реализации архитектуры стали называться просто VAX.

Пожалуй, наиболее важной из унаследованных архитектурных особенностей была ортогональность системы команд — все команды допускали использование во всех операндах всех регистров и всех режимов адресации.

Количество регистров было увеличено вдвое по сравнению с PDP-11 — с 8 до 16.

Количество режимов адресации было увеличено почти вдвое. PDP-11 поддерживала восемь режимов адресации: регистровый, косвенно-регистровый, с автоинкрементом, косвенный с автоинкрементом, с автодекрементом, косвен-

ный с автодекрементом, индексный (регистровый со смещением) и косвенно-индексный. В VAX были добавлены два варианта регистровой адресации со смещением — с использованием 8-битного и 32-битного смещений, индексный режим (допускавший использование нескольких регистров при формировании одного адреса) и режим короткого литерала. Короткий литерал занимал четыре из возможных 16 кодов режимов адресации.

В PDP-11 команды были фиксированной длины (2 байта) и всегда выравнивались на границу слова. В VAX команды переменной длины выровнены на байт.

В PDP-11 основные арифметические команды двухадресные, т. е. имеют два операнда. В VAX они трехадресные, и, кроме того, система команд содержит несколько четырех- и шестиадресных команд. Максимальная длина команды VAX может составлять 50 байт.

PDP-11 реализует плавающую точку посредством отдельной подсистемы процессора (так называемого сопроцессора) с собственным набором регистров. VAX использует для операций с плавающей точкой регистры общего назначения. Список различий можно продолжать и дальше. Несмотря на все эти различия, VAX обеспечивал совместимость с PDP-11 по языку ассемблера (т. е. все команды PDP-11 со всеми комбинациями режимов адресации имели точный функциональный эквивалент в системе команд VAX), т. е. допускал автоматическую конверсию программ. Кроме того, был предусмотрен запуск отдельных задач в режиме бинарной эмуляции PDP-11.

Первый серийный процессор линии (VAX-11/780) был реализован на ПЛМ средней степени интеграции, занимал две 19" стойки и содержал "консольный" компьютер PDP-11/20, который использовался для загрузки микрокода и инициализации основного процессора.

Несмотря на сложность и, пожалуй, даже экстравагантность системы команд, архитектура была неплохо принята рынком. Линия VAX содержит без малого сотню моделей и включает в себя широкий спектр различных компьютеров, начиная от мэйнфреймов, занимающих пять-шесть стоек (таких, как VAX-11/780, VAX 9000), и заканчивая настольными компьютерами.

Основной ОС для VAX была VAX/VMS (Virtual Memory System), по архитектуре в целом аналогичная RSX-11. В частности, архитектура подсистемы ввода/вывода была оставлена вовсе без изменений. Разработка VMS велась параллельно с разработкой процессора, и ряд особенностей VAX — в частности, знаменитая атомарная (т. е. исполнявшаяся в режиме монопольного захвата шины) команда включения элемента в связанный список — были сделаны по просьбе разработчиков ОС.

Все системные вызовы RSX имеют функциональные аналоги в VMS, похожа и структура файловой системы. Из главных новшеств следует назвать значи-

тельное усовершенствование подсистемы виртуальной памяти. VAX/VMS была первой массовой операционной системой, поддерживавшей страничную виртуальную память. Вместо своеобразного диспетчера памяти PDP-11 с его размещенной в регистрах таблицей трансляции и раздельными адресными пространствами для ОС и задачи, VAX имеет страничный диспетчер с таблицей трансляции, размещенной в ОЗУ, и отображением ОС в адресное пространство задачи. Архитектура диспетчера памяти VAX вкратце описывалась в *главе 6*.

Изменения коснулись и файловой системы. Были расширены допустимые имена файлов; RSX-11 использует имена файлов в кодировке RADIX-50 и в формате 6.3 (6 символов — имя, 3 — расширение). В VMS имена файлов могут содержать большинство символов ASCII (с единственным ограничением, что буквы должны быть в верхнем регистре) и имеют формат 32.32.

Файловую систему VMS обычно не описывают как журналируемую, но фактически она является таковой. Изменения в метаданных (файл INDEX.SYS) записываются на диск два раза, и после аварии восстановление тома происходит за гарантированное время.

Поскольку в VAX ядро отображается в адресное пространство пользовательской задачи, вместо команд системных вызовов в VAX/VMS используется прямой вызов процедур ядра. Точки входа всех процедур ядра размещены в первых страницах системного сегмента; те процедуры, которые могут работать без повышения уровня привилегий, исполняются непосредственно в контексте пользовательского процесса; те же процедуры, которые требуют системных привилегий, содержат команды `СНМ(х)`, обеспечивающие контролируемое повышение уровня доступа.

Как и в более старых системах DEC, ОС поставлялась с файлами ассемблерных макроопределений, позволявших сформировать большинство системных вызовов одной ассемблерной директивой. Для сравнения, для вызова системной процедуры из программы на языке С необходимо было описать структурную переменную, соответствующую блоку параметров, проинициализировать все поля этой структуры и только потом делать собственно вызов. В результате программа на С, делающая несколько системных вызовов, часто оказывалась во много раз длиннее, чем эквивалентная программа на ассемблере.

Исходные тексты VMS, строго говоря, нельзя назвать открытыми — они не опубликованы в Интернете и других открытых источниках. Однако все легитимные пользователи VMS могут приобрести исходные тексты системы на компакт-диске. Во многих форумах и даже в документации есть ссылки на эти исходные тексты как на ценный источник информации, в частности для разработки драйверов устройств и других модулей ядра.

В первой половине 80-х годов XX века для VMS и RSX-11 был реализован стек сетевых протоколов DECNet. Основные сервисы, предоставляемые этим протоколом, включали в себя обмен почтой, монтирование файловых систем и удаленный терминальный доступ. Для VAX/VMS предоставлялась также кластеризация, обеспечивающая общую базу данных пользовательских учетных записей между несколькими компьютерами.

К концу 80-х годов началось движение в сторону конвергенции VMS с Unix-системами. Для VMS был реализован стек протоколов TCP/IP. Представители DEC участвовали в деятельности консорциума X-Open; в частности, при разработке протокола X 11 были учтены особые требования VAX/VMS. Первые реализации X Window были доступны как для Unix, так и для VAX/VMS.

П1.2.6. OpenVMS

Компьютеры линии VAX имели огромный коммерческий успех, так что в начале 80-х годов XX века компания без особых сожалений отказалась от поддержки 36-битных машин линии DEC-10/20. Будущее компании представлялось безоблачным: компьютеры и ОС линии VAX/VMS по всем параметрам могли конкурировать как с мэйнфреймами IBM, так и с мини- и суперминикомпьютерами других поставщиков. Компания имела большую и лояльную пользовательскую базу и чрезвычайно лояльную субкультуру разработчиков ПО.

Тучи начали сгущаться во второй половине 80-х. К этому времени выяснилось, что микропроцессоры и микрокомпьютеры, ранее считавшиеся просто игрушками, представляют серьезную угрозу для рынка миникомпьютеров, особенно в нижнем ценовом сегменте. Архитектура PDP-11 была довольно простой и однокристальные реализации процессоров LSI-11 появились еще в начале 80-х годов. Однако PDP-11 из-за ограниченного адресного пространства была неадекватна требованиям многих сегментов рынка и с трудом могла конкурировать даже с IBM PC. Архитектура VAX, требовавшая огромных объемов микрокода, в свою очередь, была слишком сложна для однокристальной реализации.

В 1984 году появился первый компьютер MicroVAX, в котором часть наиболее сложных команд и комбинаций режимов адресации была реализована с помощью программной эмуляции. Это позволило резко уменьшить объем микрокода и сделать процессор сначала на одной плате, а позднее и на одном кристалле — CVAX, NVAX, Rigel и др.

Выпущенные на рынок в 1987 году микросхемы CVAX знамениты из-за надписи кириллицей, напыленной на незанятую область между контактными площадками, — "СВАКС... Когда вы забатите довольно воровать настоящий лучший" (рис. П1.1). Надпись была адресована советским "разработчикам"

микросхем. Впрочем, воспроизведение CVAX все равно оказалось далеко за пределами возможностей советской микроэлектронной промышленности.

Переход к архитектуре MicroVAX, впрочем, в долгосрочной перспективе не помог. Микросхемы все равно были очень сложными, а программная эмуляция приводила к потерям производительности, так что 32-разрядные процессоры Motorola 680x0 и RISC-процессоры различных изготовителей, а потом и Intel x86 быстро начали догонять MicroVAX сначала по критерию производительность/цена, а потом и по абсолютной производительности. Немаловажным фактором, повышавшим цену микросхем DEC, был тот факт, что к концу 80-х годов XX века большинство изготовителей, в том числе Intel, Motorola и Texas Instruments (производившие микросхемы SPARC), перешли к практически полностью автоматизированной разработке интегральных схем, в то время как инженеры DEC проектировали устройства высокой степени интеграции вручную.



Рис. П1.1. Участок микросхемы CVAX, фото с сайта [micro.magnet.fsu.edu russians]

DEC делал несколько попыток перейти к RISC-архитектуре. Наиболее известны четыре из них:

- завершившийся неудачей проект PRISM Дэвида Катлера;
- лицензирование процессоров MIPS, использовавшихся в настольных компьютерах DECStation под управлением Unix-системы Ultrix;
- проект Alpha;

- совместный с компанией ARM (Advanced RISC Machines) проект StrongARM.

В техническом отношении удачным можно признать лишь последние два из проектов. Процессоры StrongARM нашли применение на рынке встраиваемых устройств и позднее, при разделе имущества DEC, были приобретены компанией Intel. Сейчас эти процессоры производятся под названием Intel XScale и применяются в карманных компьютерах и встраиваемых приложениях.

Первый процессор линии Alpha — EV4 (21064) — был выпущен в 1992 году. Alpha представляет собой RISC-процессор второго поколения, в архитектуре которого предусмотрены явные средства управления конвейеризацией. Одной из целей разработки было облегчение программной эмуляции системы команд VAX. Предполагалось исполнять программы для VAX в режиме бинарной JIT-компиляции, т. е. преобразовывая машинный код VAX в машинный код Alpha в момент загрузки. Такая же поддержка планировалась и для x86.

Это был первый серийный 64-битный микропроцессор. Некоторое время процессоры Alpha удерживали абсолютное лидерство как по тактовой частоте (сравнимой с реализованными на эмиттерно-связанной логике процессорами тогдашних суперкомпьютеров), так и по производительности. Архитектура Alpha оказала значительное влияние на проект Intel Itanium и микроархитекттуру суперскалярных реализаций x86 — Intel P6, AMD Athlon и последующие микросхемы. Влияние было настолько значительным, что DEC даже пытался судиться с Intel по этому поводу.

Для процессоров Alpha была разработана новая версия ОС, известная как OpenVMS, обеспечивающая полную совместимость с VAX/VMS на уровне исходных текстов. Саму систему, однако, пришлось переписывать на языке высокого уровня.

Реализация JIT-компиляторов для Alpha оставляла желать лучшего, поэтому плавной миграции пользователей VAX на новую аппаратную архитектуру не получилось. Не получилось организовать и прозрачную совместимость с машинами x96. Версии Windows NT для Alpha поставлялись, но бинарной совместимости с Win32/x86 не обеспечивали; при этом NT использовала "альфу" как 32-битный процессор.

Дела компании пошли под гору. В районе 1997 года началась, фактически, распродажа имущества. Основную часть производственных мощностей и интеллектуальной собственности, в том числе и права на OpenVMS, приобрел Compaq в 1998 году. Впрочем, переварить приобретение Compaq не смог; производство и поддержка машин на основе процессоров Alpha сокращались. В 2001 году Compaq был поглощен компанией Hewlett-Packard. HP продолжает производство процессоров Alpha EV7 и поддержку OpenVMS для Alpha

и Intel Itanium, но планы прекращения продаж и поддержки уже опубликованы.

П1.3. Семейство Unix

В 80-е годы мейнстримные пользователи предпочтали юникс VAX/VMS.

Сейчас они предпочитают юникс Windows NT.

Какая часть этого сообщения вам непонятна?

*Реплика из конференции USENET
alt.comp.os.windows.advocacy*

Обширное и бурно развивающееся семейство Unix оказало огромное идеиное влияние на развитие операционных систем в 80-е и 90-е годы XX века. Генеалогия систем семейства опубликована на сайте [\[perso.wanadoo.fr\]](http://perso.wanadoo.fr) и слишком обширна для того, чтобы ее можно было полностью привести в книге.

Применения систем семейства крайне разнообразны, начиная от встраиваемых приложений реального времени, включая графические рабочие станции для САПР и геоинформационных систем, и заканчивая серверами класса предприятия и массивно параллельными суперкомпьютерами. Некоторые важные рыночные ниши, например, web-хостинг, передачу почты и другие структурные сервисы Интернета, системы семейства занимают практически монопольно.

Родоначальником семейства следует, по-видимому, считать не первую версию Unix, а Multics, совместно разрабатывавшуюся в 1965—1969 годах General Electric и Bell Laboratories. За это время General Electric выделило подразделение, занимавшееся работами над Multics и аппаратной платформой для нее (GE-645), в отдельную компанию Honeywell.

Multics была первой из промышленных систем, предоставлявших:

- создание процессов системным вызовом `fork`;
- страничную виртуальную память;
- отображение файлов в адресное пространство ОЗУ;
- вложенные каталоги;
- неструктурированные последовательные файлы;
- многопользовательский доступ в режиме разделения времени;
- управление доступом на основе ограниченных ACL (кольцо доступа).

Multics оказала огромное влияние не только на разработчиков Unix — значительные следы идейного влияния этой системы прослеживаются также в

RSX-11 и VAX/VMS фирмы DEC. Последние Multics-системы были доступны в Интернете в 1997 году.

В 1969 году Bell Laboratories отказалась от работ над Multics и начала разработку собственной ОС для внутренних нужд. По-видимому, основной причиной этого шага было осознание несоответствия между амбициозными целями проекта Multics (ОС была весьма требовательна к ресурсам и могла работать только на больших компьютерах Honeywell), в то время как материнской компании Bell Labs — American Telephone & Telegraph — требовалась единая операционная среда, способная работать на различных миникомпьютерах, используемых в подразделениях телефонной сети США.

В Bell Laboratories был объявлен внутренний конкурс на разработку переносимой ОС, способной работать на миникомпьютерах различных поставщиков. К проекту были предъявлены следующие основные требования:

- многоплатформенность;
- вытесняющая многозадачность;
- многопользовательский доступ в режиме разделения времени;
- развитые телекоммуникационные средства.

Один из участников работ над Multics, К. Томпсон, разработал крайне упрощенное ядро ОС, названное UNIX, для миникомпьютера PDP-7. К 1972 году К. Томпсон и Д. Ритчи переписали ядро системы в основном на языке С и продемонстрировали возможность переноса ОС на миникомпьютеры PDP-11. Это обеспечило выполнение всех требований конкурса, и UNIX была признана основной платформой для вычислительных систем, эксплуатируемых в AT&T.

Легенды доносят до нас более колоритные детали ранних этапов истории новой системы. Так, одна из очень популярных легенд, излагаемая в той или иной форме несколькими источниками, утверждает, что история UNIX началась с разработанной для Multics игровой программы под названием Star Wars (звездные войны — сама эта программа и даже правила игры до нас не дошли). Уволенный из группы разработчиков Multics за разгильдяйство (я привожу наиболее радикальный вариант легенды, не заботясь о его согласовании с историческими фактами), Томпсон занялся "оживлением" неиспользуемой PDP-7, стоявшей в углу машинного зала. Оживление заключалось в написании ядра ОС, реализующего подмножество функциональности Multics, достаточное для того, чтобы перенести и запустить его любимые Star Wars [Бах 1986].

Первые версии UNIX были рассчитаны на машины без диспетчера памяти. Процессы загружались в единое адресное пространство. Ядро системы разме-

щалось в нижних адресах ОЗУ, начиная с адреса 0, и называлось *сегментом реenterабельных процедур*. Реenterабельность обеспечивалась переустановкой стека в момент системного вызова и запретом переключения задач на все время исполнения модулей ядра. На машинах с базовой адресацией выполнялось перемещение образов процессов по памяти и сброс образа процесса на диск (задачный своппинг).

Работа — а особенно разработка программного обеспечения — в режиме разделенного времени на машине с незащищенной памятью — это весьма своеобразное занятие. По воспоминаниям одного из участников команды разработчиков Unix, перед запуском неотлаженной версии программы необходимо было кричать "a out" и делать паузу, чтобы остальные пользователи могли сохранить редактируемые файлы.

В Multics и современных системах Unix `fork` реализуется посредством копирования страниц при модификации. Ранние версии UNIX физически копировали образ процесса. Большая часть (по некоторым оценкам, до 90%) `fork` немедленно продолжается исполнением системного вызова `exec`, поэтому одной из первых оптимизаций, придуманных в университетских версиях системы, было введение системного вызова `vfork`. Порожденный этим вызовом процесс исполнялся на самом образе родителя, а не на его копии. Создание нового образа процесса происходило только при исполнении `exec`. Для того чтобы избежать возможных проблем взаимоисключения (например, при вызове нереenterабельных функций стандартной библиотеки), исполнение родителя приостанавливалось до тех пор, пока потомок не выполнит `exec` или не завершится.

Выделяют [Баурн 1986] следующие отличительные особенности системы:

- порождение процессов системным вызовом `fork`, который создает копию адресного пространства и пользовательской области (user area, так в Unix называются структуры данных ядра, связанные с процессом) процесса;
- результат завершения процесса хранится в его дескрипторе и может быть считан только родителем. В списке процессов такой дескриптор выглядит как процесс в специальном состоянии, называемом зомби (zombie);
- процессы-сироты (продолжающие исполнение после завершения родителя) усыновляются процессом с идентификатором, равным 1;
- процесс с идентификатором 1 запускается при загрузке системы (по умолчанию это `/bin/init`) и запускает все остальные обязательные задачи в системе. Наличие такого процесса иногда объявляют необходимым и достаточным критерием для причисления той или иной системы к семейству Unix;

- древовидная структура пространства имен файловой системы: дополнительные ФС монтируются в те или иные точки корневой ФС и идентифицируются затем точкой монтирования, а не именем устройства, на котором расположены;
- файлы рассматриваются ОС как неструктурированные потоки байтов и не типизированы на уровне ОС (в частности, на уровне ОС нет деления файлов на записи);
- файловая система поддерживает множественные имена файлов в виде жестких и, у более поздних версий, символьических связей;
- отложенное удаление файлов: если процесс открыл файл, а другой процесс его удалил, то первый процесс может продолжать работу с файлом, и физическое удаление происходит только после того, как первый процесс его закроет;
- лозунг "все — файл" (который, впрочем, последовательно реализован только в экспериментальной системе Plan 9) — в реальных Unix системах практически всегда присутствуют объекты, к которым не применимы файловые операции: переменные среды, структуры данных ядра в ранних версиях (позднее они были оформлены в виде объектов псевдофайловой системы /proc), объекты SysV IPC и примитивы взаимоисключения POSIX Thread Library в современных системах;
- своеобразный командный язык, основанный на широком применении переназначения ввода/вывода и конвейеров (последовательностей задач, соединенных трубами).

Некоторые из перечисленных особенностей были позаимствованы другими ОС. Так, переназначение ввода/вывода и конвейеры реализуются командными процессорами ОС семейства CP/M, начиная с MS DOS 3.30. cmd.exe — командный процессор OS/2 и Windows NT/2000/XP — даже правильно понимает конструкцию `cmdline | more >&1`. Стандартные командные процессоры UNIX портированы практически на все современные ОС. Функциональный аналог Korn Shell включен в стандартную поставку Windows 2000.

Жесткие связи файлов (но без отложенного удаления) поддерживаются FCS2 (файловой системой VAX/VMS), NTFS и jfs при использовании под OS/2. Точки монтирования (подключение дополнительных ФС в дерево каталогов существующей ФС) реализованы в Windows 2000 и Toronto Virtual File System для OS/2.

Системный вызов `CreateProcess` Windows NT/2000/XP может имитировать семантику `fork`: для этого достаточно передать в качестве имени программы для нового процесса пустой указатель.

Библиотека EMX для OS/2 реализует `fork` с помощью собственного обработчика страничного отказа.

П1.3.1. Распространение UNIX

Глупый пингвин робко прячет,
умный — гордо достает.

Неизвестный автор

AT&T в 70-е годы XX века была "естественной" монополией в области телекоммуникаций. Этот статус гарантировался законодательным запретом деятельности других телекоммуникационных компаний на территории США. В обмен на этот статус AT&T вынуждена была подчиняться ряду регуляторных мер, в частности — ей было запрещено выходить на другие, кроме телекоммуникационного, рынки, в том числе на рынок программного обеспечения. Однако разработчики UNIX чувствовали, что их системе суждено гораздо более привлекательное будущее, чем внутренний стандарт крупной компании.

С 1973 года одна из дочерних компаний AT&T, Western Electric, дала разрешение на использование UNIX в некоммерческих целях. Началось распространение системы в университетах США. Наибольший вклад в распространение и развитие университетской версии системы внес университет Беркли, в котором было создано специальное подразделение — BSD (Berkeley Software Distribution).

В BSD Unix было включено множество ценных нововведений, таких как:

- сегментная (на старших моделях PDP-11) и страничная (на VAX-11/780) виртуальная память;
- раздельные адресные пространства процессов и выделенное адресное пространство ядра;
- абсолютные загрузочные модули формата a.out;
- примитивная форма разделяемых библиотек;
- усовершенствования механизма обработки сигналов;
- управление сессиями и заданиями в пределах сессии.

Самое важное нововведение было сделано в начале 80-х годов, когда в рамках работ по проекту DARPA сетевое программное обеспечение ARPANet было перенесено с TOPS/20 на BSD Unix. Вскоре сетевой стек BSD стал референтной реализацией (реализация, на совместимость с которой тестируют все остальные) того, что ныне известно как семейство протоколов TCP/IP.

В 1980 году было решено начать коммерческое распространение системы на несколько необычных принципах: AT&T предоставляла сторонним коммерческим фирмам (естественно, за плату) лицензии на использование исходных текстов ядра и основных системных утилит текущей версии UNIX, а уже эта сторонняя коммерческая фирма (дистрибутор) строила на основе полученных и самостоятельно разработанных компонентов законченную систему — с инсталляционной программой, системой управления пакетами и т. д. — и занималась ее продажей конечным пользователям и сопровождением. Таким образом была создана специфическая бизнес-модель распространения ОС семейства Unix, хорошо знакомая пользователям Linux.

Первым из коммерческих распространителей стала фирма Microsoft, продающая ядро UNIX v7 в составе ОС Microsoft Xenix. Xenix поставлялся почти для всех популярных в то время 16-разрядных миникомпьютеров и микропроцессорных систем [Дейтел 1987]. Как и BSD Unix, Xenix использовал виртуальную память и имел отдельное адресное пространство для ядра. В 1983 году торговая марка Xenix и весь дистрибуторский бизнес были переданы фирме SCO в обмен на долю акций последней.

К середине 80-х воспитанное на университетских версиях UNIX поколение студентов пришло в промышленность. Началось бурное развитие *рабочих станций (workstation)* — мощных 32-разрядных персональных компьютеров, как правило, оснащенных страничными или сегментными диспетчерами памяти. Лицензия BSD допускала построение на основе кода BSD коммерческих систем без каких-либо ограничений, в том числе и без денежных выплат разработчикам ядра. Благодаря этому, а также благодаря техническому совершенству ядра BSD Unix, последнее оказалось гораздо более привлекательным, чем ядро AT&T, поэтому основная масса поставщиков рабочих станций строили свои ОС на основе BSD Unix. Это привело к быстрому и неконтролируемому размножению систем, называвших себя Unix, и при этом имевших значительное количество несовместимостей — дополнительных или, наоборот, нереализованных системных вызовов, ошибок, "документированных особенностей" и т. д.

В 1984 году AT&T заключила с федеральным антимонопольным комитетом США соглашение, в соответствии с которым компания должна была выделить локальные телефонные сети в отдельные компании, и согласовала планы создания конкурентной среды на рынке междугородней связи и выделения в отдельные компании подразделений, не имеющих отношения к телекоммуникациям. Долгосрочные результаты этого соглашения до сих пор являются предметом горячих дебатов среди юристов и экономистов, но важным с нашей точки зрения является то, что AT&T смогла напрямую заняться продажами и поддержкой программного обеспечения. На рынок вышло ядро Unix System V — первая поддерживаемая версия ядра AT&T UNIX.

В 1987 году вышла версия UNIX System V Release 3, включавшая в себя асинхронные драйверы последовательных устройств (STREAMS), универсальный API для доступа к сетевым протоколам (TLI), средства межпроцессного взаимодействия (семафоры, очереди сообщений и сегменты разделяемой памяти), ныне известные как *SysV IPC*, BSD-совместимые сокеты и ряд других "BSDизмов" [Робачевский 1999]. SVR3 в то время воспринималась как этапная ОС, однако дальнейшее развитие системы вынуждает нас отнести ее, скорее, к переходным версиям.

В этом же году AT&T и Sun Microsystems заключили стратегическое соглашение о разработке перспективного ядра UNIX System VI, которое должно было обеспечить совместимость с System V, BSD Unix и Xenix и, тем самым, консолидировать возникший зоопарк Unix-систем.

Не имея финансовой поддержки со стороны локальных телефонных сетей, AT&T оказалась вынуждена заняться поисками средств для поддержки деятельности по развитию UNIX. Во второй половине 80-х было сделано несколько попыток взыскать лицензионные отчисления с поставщиков коммерческих систем на основе BSD Unix. Нельзя сказать, чтобы эти попытки были особенно последовательными и успешными, но они породили ряд инициатив по разработке "лицензионно чистой Unix-системы".

Среди этих инициатив необходимо назвать следующие:

- микроядро BSD Mach;
- minix A. Танненбаума;
- проект Р. Столлмана GNU (GNU Not Unix — рекурсивная аббревиатура) [www.gnu.org];
- консорциум OSF (Open Software Foundation — фонд открытого программного обеспечения).

П1.3.2. Микроядро

Концепция микроядра с технической точки зрения подробно рассматривается в разд. 8.3. С коммерческой (если уместно говорить о коммерческих целях разработки свободно распространяемого ПО) точки зрения BSD Mach был попыткой убить одновременно двух зайцев — совместить переписывание ядра BSD Unix для достижения лицензионной чистоты с изменением архитектуры этого ядра.

Микроядерная архитектура позволила бы избежать самой одиозной черты традиционных систем Unix — однопоточного (или, точнее, кооперативно многозадачного) ядра, и сделала бы систему пригодной для использования в задачах реального времени. Проект Mach не имел полного успеха — основные ветви BSD до сих пор используют традиционное монолитное ядро.

Наиболее успешная ОС, основанная на микроядре Mach, — это Darwin (Unix-подсистема MacOS X).

Однако идея микроядра и сам термин получили широкое распространение. Микроядерную архитектуру имеет UNIX System V Release 4. Кроме того, на самостоятельно разработанном микроядре основана своеобразная ОС реального времени, часто относимая к семейству Unix — QNX.

Основные работы над ядром BSD UNIX пошли в другом направлении: подсистемы, которые AT&T считал основанием для требования лицензионных выплат, переписывались с нуля, но архитектура системы в целом пересмотрена не подвергалась. Этот процесс был в основном завершен к 1994 году, и современные ветви BSD по-прежнему имеют монолитную архитектуру.

П1.3.3. Minix

Minix был разработан А. Танненбаумом, преподавателем университета Врийе (Vrije University) в Амстердаме [[www.cs.vu.nl minix](http://www.cs.vu.nl/minix)]. Эта компактная система, созданная для учебных целей, способна работать на 16- и 32-разрядных микропроцессорах, причем не только самостоятельно, но и будучи скомпилирована и запущена в качестве задачи под "нормальной" ОС Unix. Первая версия системы имела очень консервативную (чтобы не сказать — архаичную) архитектуру, очень близкую к архитектуре ранних версий UNIX. Minix 2.0, выпущенный в 1996 году, основан на микроядре и поддерживает страничную виртуальную память на процессорах x86.

Основной целью разработки было создание системы, которая, с одной стороны, была бы работоспособна и могла бы продемонстрировать основные архитектурные концепции современных многозадачных ОС, а с другой — достаточно проста, чтобы студенты могли полностью в ней разобраться. Второе требование фактически исключало возможность доработки ОС до состояния, в котором она могла бы стать коммерчески применима.

Наиболее известен прямой потомок Minix, Linux. Первая версия Linux разрабатывалась путем переписывания ядра Minix модуль за модулем, что значительно упростило Л. Торвальдсу отладку системы. Воспоминаниями об этих временах в современном Linux является поддержка файловой системы minix, название основной ФС — ext2fs (Second Extended File System — расширенная [по сравнению с minix] файловая система, вторая версия) и реликты кода Minix в некоторых модулях.

П1.3.4. GNU Not Unix

Проект GNU был начат преподавателем Массачусетского технологического института Р. Столлмэном и имел целью разработку полностью свободной

операционной системы. "Полная свобода" гарантировалась своеобразным лицензионным соглашением, так называемым *copyleft* — текст современной версии этого соглашения, GPL (General Public License — общая публичная лицензия), размещается в заголовке каждого файла исходного текста программных продуктов, распространяемых в соответствии с данной лицензией [www.fsf.org].

Вопросы о необходимости, целесообразности и допустимости этой схемы распространения ПО, а также о моральных, юридических, экономических, социальных и других последствиях ее применения до сих пор являются предметом жарких дебатов [www.tuxedo.org homesteading]. Тем не менее в рамках деятельности FSF (Free Software Foundation — фонд свободного программного обеспечения) было разработано немало высококачественного и полезного ПО, прежде всего — компилятор GNU C/C++, текстовый редактор (и по совместительству интегрированная среда разработки) GNU Emacs, функциональные эквиваленты стандартных утилит UNIX и ряд других программ и утилит. Основной целью проекта являлась разработка GNU HURD, весьма амбициозной микроядерной ОС.

В 1996 году публике была представлена крайне сырья альфа-версия системы. К тому времени Linux уже шествовал по планете победным шагом и отвлек на себя всех специалистов, способных участвовать в разработке ядра и согласных распространять результаты своей деятельности на условиях GPL. Наверное, из-за этого HURD не привлек внимания ни разработчиков, ни бета-тестеров. С тех пор до момента публикации этой книги не поступало ни новых версий, ни объявления о прекращении работ. По-видимому, следует признать, что проект HURD завершился провалом.

П1.3.5. Open Software Foundation

Консорциум OSF, в который вошли DEC, IBM, Hewlett-Packard и ряд менее известных поставщиков рабочих станций и серверов, был создан в 1988 году. Его деятельность началась с принятия и публикации стандарта POSIX.1 (Portable OS Interface based on uniX — переносимый интерфейс ОС, основанный на Unix).

В рамках OSF начались работы по разработке ядра Unix-совместимой ОС, по архитектуре в целом аналогичной UNIX SVR3 (монолитное ядро с поддержкой STREAMS и некоторыми особенностями BSD). К моменту завершения разработки уже был выпущен UNIX System V Release 4.2 (Destiny), достигший всех целей, заявлявшихся в проекте UNIX System VI, и консорциум фактически распался. DEC Unix (известный также как OSF/1) оказался чрезмерно тяжеловесным, не имел коммерческого успеха и, возможно, сыграл немалую

роль в судьбе компании DEC. Значительно более счастливой оказалась судьба IBM AIX, лишь частично основанной на коде OSF.

В 1996 году то, что осталось от OSF, слилось с консорциумом X/Open, деятельность которого по стандартизации Unix-систем имела гораздо больший успех.

П1.3.6. X/Open

Консорциум X/Open [www.opengroup.org] был основан в 1990 году и имел гораздо более широкий состав, чем OSF, включая в себя практически всех производителей и поставщиков Unix-систем и ряд образовательных учреждений. Вместо разработки новой версии системы консорциум занялся разработкой стандартов, которым система любого поставщика должна была удовлетворять, чтобы иметь право называться Unix.

Одним из главных достижений в деятельности этого консорциума следует считать разработку и публикацию спецификаций X Window — протокола распределенной графической оконной системы, который стал основой графического интерфейса практически всех Unix-систем.

В 1993 году фирма Novell, которая к тому времени приобрела авторские права и команду разработчиков AT&T, передала консорциуму торговую марку UNIX™. С тех пор консорциум выдавал право носить название UNIX™ системам, которые проходили тесты на совместимость с текущей версией спецификаций. Было также, наконец-то, решено, что торговой маркой является только UNIX (все буквы заглавные), но не Unix. Стандартизация оказала крайне благотворное влияние на рынок Unix-систем и приложений для них, практически устранив различия, существенные для разработки прикладного ПО, между наиболее распространенными системами семейства.

Поскольку сертификация была платной, некоммерческие версии системы, такие как ветви BSD и Linux, ее не проходили. Неожиданным результатом сертификационной политики консорциума стало право OS/390 называться UNIX™ после прохождения тестов в 1998 году [www.opengroup.org xu007].

П1.3.7. UNIX System V Release 4

А больше всего было стыдно Максима и Федора, которые мало того, что пьянь политурная, так еще и смотрят с сочувствием.

В. Шинкарев

Обещанная в 1987 году UNIX System VI вышла на рынок в 1989 году под названием UNIX SVR4. Микроядерная система обеспечивала полную бинарную совместимость с SVR3, бинарную же совместимость с 16- и 32-разрядными

Xenix на процессоре x86, и совместимость на уровне исходных текстов с BSD Unix v4.3 [Хевиленд/Грей/Салама 2000]. Заявленная цель консолидации всех основных ветвей Unix в единой системе была полностью достигнута. Sun Microsystems приступила к переводу своих пользователей на Sun OS 5.x (ныне известна как Solaris), основанную на ядре SVR4.

Бизнес-модель распространения UNIX SVR4 была похожа на традиционную для AT&T: Unix System Laboratories сохраняла авторские права на ядро системы, но лицензировала право на использование ядра и отдельных компонентов другим компаниям, которые, в свою очередь, строили на основе этих компонентов свои поддерживаемые системы — Sun Solaris, Silicon Graphics IRIX, Novell UnixWare и некоторые менее известные.

Версия SVR4 была этапной — она включала в себя следующие компоненты:

- многопоточное микроядро;
- класс планирования реального времени (процессы с этим классом планирования имеют приоритет выше, чем нити ядра);
- новый формат загрузочного модуля ELF (Executable and Linking Format), обеспечивавший удобную работу с разделяемыми и динамическими библиотеками;
- динамическое подключение и отключение областей свопинга;
- динамическую загрузку и выгрузку модулей ядра;
- многопоточность в пределах одного процесса (так называемые LWP (Light Weight Processes — легкие процессы));
- псевдофайловую систему /proc, обеспечивающую контролируемый доступ к адресным пространствам других процессов и структурам данных ядра;
- оптимизирующий компилятор ANSI C, по качеству кода не уступающий GNU C.

В 1991 году подразделение AT&T, занимающееся развитием и поддержкой UNIX, было выделено в отдельное предприятие, USL (UNIX System Laboratories). Дальнейшая история этой организации представляет неплохой сюжет для романа: в 1992 году USL была приобретена фирмой Novell — тогдашний CEO (Chief Executive Officer — главный администратор) компании Р. Нурда пытался сформировать линию продуктов, способную конкурировать со всеми предложениями Microsoft. В 1993 году права на торговую марку UNIX были переданы консорциуму X/Open. В 1995 году акционеры Novell, испуганные перспективой конфронтации с Microsoft, сняли Нурду с поста CEO и стали распродавать его приобретения. В частности, USL и лицензионные соглашения с распространителями UNIX SVR4 (Sun, Silicon Graphics, Microport и др.) были проданы фирме SCO. Нурда основал компанию Caldera, основным биз-

несом которой стало распространение и поддержка Linux. 7 мая 2000 года в тексте этой истории была поставлена... ну, скорее всего, не точка, но весьма важный знак препинания: Caldera приобрела компанию SCO вместе со всеми правами на SVR4 [www.sco.com].

Впрочем, завершение этой истории, видимо, уже не за горами. В начале XXI века дела Caldera/SCO пошли под гору. Отчасти это было связано с коммерческими неудачами SGI Irix и постепенным уходом Hewlett Packard с рынка RISC-серверов — лицензионные отчисления SGI за SVR4 и HP за код SVR3, использовавшийся в составе HP/UX, составляли значительную часть доходов SCO.

В 2005 году компания SCO решила повторить неудачную попытку Bell Laboratories и взыскать с пользователей Linux деньги за использование кода, якобы заимствованного из System V. Основные претензии при этом предъявлялись компании IBM, которая получала лицензии на разные версии Unix System V и различные его подсистемы несколькими разными путями. IBM использует довольно много кода Unix System V R3 в IBM AIX. Кроме того, в первой половине 90-х годов IBM приобрела компанию Sequent, которая делала массивно параллельные многопроцессорные системы на основе процессоров x86, работавшие под управлением специализированной версии SCO Unix (также основанной на System V Release 3). По утверждениям SCO, сотрудники IBM перенесли часть кода в модули ядра Linux; затем эти модули были включены в основное дерево исходных текстов ядер Linux 2.4 и 2.6. Впрочем, на время подготовки книги к печати так и не было опубликовано кода, о котором якобы идет речь. Вполне возможно, что, как и в случае с процессом Bell Laboratories против BSD, дело будет тихо замято.

Развитие основной ветви системы (System V Release 4.2) практически прекратилось, активным развитием системы продолжал заниматься только Sun.

Sun Solaris, построенный на ядре System V Release 4, поддерживает три основные аппаратные архитектуры — 32битные процессоры SPARC v8, 64-битные процессоры SPARC v9 и x86 в 32-разрядном режиме. С каждым релизом Solaris делалась попытка прекратить поддержку x86, но под давлением пользователей эта поддержка сохраняется. В Solaris 10 добавилась поддержка x64 (IA-32e, 64-разрядного режима процессоров x86).

Многие из подсистем Solaris были значительно усовершенствованы по сравнению с оригинальной SVR4.

- В Solaris, по сравнению с традиционными Unix-системами, значительно изменен процесс управления драйверами устройств. При загрузке система создает дерево каталогов, соответствующее иерархии периферийных шин и устройств, обнаруженных на этих шинах, а также отдельные каталоги для установленных в системе псевдоустройств. На машинах Sun этот ката-

лог создается на основе информации, предоставляемой загрузочным ПЗУ. На машинах x86 соответствующую информацию собирает вторичный загрузчик, имитирующий также и другие функции ПЗУ Sun. Корень этой иерархии находится в каталоге /devices. На каждую из обнаруженных периферийных шин создается каталог, в котором, в свою очередь, создаются каталоги и файлы, соответствующие устройствам. В каталогах для шин PCI, имена каталогов и файлов устройств соответствуют PCI ID этих устройств. Файлы в традиционном для Unix-систем каталоге /dev представляют собой символические ссылки на иерархию /devices.

- В Solaris 7 была добавлена поддержка сетевого протокола IPv6.
- В Solaris 8 была реализована журнальная версия файловой системы UFS, позволившая администраторам серверов перейти на журнальную ФС без переразметки дисков.
- Также в Solaris 8 были реализованы контрольные точки, сохранявшие состояние файловой системы на определенный момент (этот механизм относительно подробно описывается в *разд. 11.4.3*).
- В Solaris 9 была реализована система "проектов" (projects), которая позволяет выделять ресурсы (например, гарантированную долю процессорного времени или ОЗУ) группам процессов.
- Sun Solaris 8 при работе на серверах семейства Sun Fire поддерживает динамическое подключение, исключение и горячую замену процессорных модулей и модулей ОЗУ, а также одновременную работу нескольких копий Solaris на разных процессорах одного вычислительного комплекса с динамическим перераспределением ресурсов (процессоров, ОЗУ, дисков) между виртуальными системами.
- В дистрибутив Solaris 9 был включен набор из наиболее популярных утилит проекта GNU, таких как командный процессор bash. Лицензия GPL, на основе которой распространяются эти утилиты, не запрещает включать программы в коммерчески распространяемые дистрибутивные пакеты, она лишь не позволяет запрещать кому бы то ни было дальнейшее распространение этих программ.
- В Solaris 10 была реализована новая файловая система ZFS, основанная на принципе write anywhere (*см. разд. 11.5*).
- В Solaris 10 реализованы зоны — механизм виртуализации операционной системы, который позволяет запускать на одной физической машине несколько виртуальных копий Solaris. Каждая зона имеет собственную БД учетных записей, собственную учетную запись root, свой список устройств и (виртуальных) файловых систем, свой процесс init и собственную таблицу процессов. Зоны даже могут независимо перезагру-

жаться. Однако все зоны разделяют общее ядро, хотя доступ к этому ядру частично ограничен, а частично виртуализован. Зонам может выделяться гарантированная доля ОЗУ и процессорного времени. В ноябрьском патче 2006 года реализован также "горячий" перенос зон между компьютерами, с сохранением состояний всех процессов и всех сетевых соединений.

На всем протяжении 90-х годов архитектура ядра не подверглась существенным изменениям. Как и MVS полугода десятилетиями раньше, UNIX достиг совершенства в своем роде и нуждается не в новой архитектуре, а только в оптимизации существующего кода (ядро SVR4 несколько тяжеловато по сравнению с монолитными ядрами BSD и Linux) и развитии отдельных подсистем.

В 2005 году Sun Microsystems опубликовала значительную часть исходных текстов ядра Solaris в рамках программы Open Solaris [opensolaris.org] на условиях CDDL (Common Development and Distribution License — общая лицензия на разработку и распространение). Обсуждение отличий этой лицензии от других популярных "свободных" лицензий, таких как GPL, уело бы нас далеко от основной темы книги. По мере достижения соглашений с компаниями, которые владеют авторскими правами на еще неопубликованные части системы, список опубликованных модулей, по-видимому, будет расширяться.

Новая версия Solaris (Solaris 11) разрабатывается на основе OpenSolaris и включает в себя значительную часть кода, разработанного добровольцами в рамках программы OpenSolaris.

П1.3.8. Linux

В 1991 году Л. Торвальдс, в тот момент — студент университета Хельсинки, приступил к разработке того, что ныне известно как Linux, — полноценной операционной системы, основанной на исходных кодах Minix и распространяемой на условиях GPL [www.linux.org].

В 1992 году была выпущена первая публичная версия системы. К тому времени сообщество пользователей и разработчиков freeware уже успело устать от задержек выпуска GNU HURD и обещаний Столлмана и приняло новый проект с огромным энтузиазмом. Ряд компаний (RedHat, Caldera, SuSe и множество других) начал распространение коммерчески поддерживаемых дистрибутивов ОС на основе ядра Linux, воспроизводя таким образом бизнес-модель распространения AT&T UNIX начала 80-х.

Вышедшее в 1997 году ядро Linux 2.0 имело вполне приемлемую по стандартам коммерческих ОС надежность и почти все наиболее прогрессивные черты других Unix-систем:

- загрузочные модули и разделяемые библиотеки формата ELF;
- псевдофайловую систему /proc;
- динамическое подключение и отключение swap-файлов;
- длинные файлы (64-разрядные — длина файла и смещение в нем);
- многопоточность в пределах одного процесса (POSIX thread library);
- поддержку симметричной многопроцессорности;
- динамическую загрузку и выгрузку модулей ядра;
- стек TCP/IP, совместимый с BSD 4.4, с поддержкой IPSec, фильтрации пакетов и др.;
- SysV IPC;
- бинарную совместимость с UNIX System V на процессорах x86 (iBCS — Intel Binary Compatibility Standard) и, позднее, на SPARC и MIPS;
- поддержку задач реального времени (класс планирования реального времени в монолитном Linux невозможен; такие задачи загружаются как модули ядра).

Linux перенесен практически на все 32- и 64-разрядные машины, имеющие диспетчер памяти, начиная от Amiga и Atari и заканчивая IBM System/390 и IBM z/90. Бинарные эмуляторы Linux включены в состав Solaris/SPARC и FreeBSD.

Ядро Linux быстро развивается и еще не достигло той степени "зрелости" и стабильности, которая характерна для SVR4 и ветвей BSD. В частности, поэтому среднее количество опасных ошибок, обнаруживаемых в системе за фиксированный интервал времени, существенно выше, чем в двух указанных ОС; производительность отдельных подсистем также оставляет желать лучшего. Однако положение довольно быстро улучшается и, по-видимому, в обозримом будущем Linux может стать одним из технологических лидеров отрасли.

П1.3.9. MacOS X

Гамп: Лейтенант Дэн уговорил меня вложить деньги в какую-то фруктовую компанию. А потом он мне позвонил и сказал, что о деньгах можно больше не беспокоиться. А я сказал, что это хорошо! Одним [повородом для беспокойства] меньше.

Э. Рот, к. ф. "Форрест Гамп"

Недавнее и, пожалуй, неожиданное прибавление в семействе Unix — это MacOS X.

Первые версии MacOS — операционной системы для компьютеров Apple Macintosh — представляли собой кооперативно многозадачную ДОС с незащищенной памятью и событийно-ориентированным интерфейсом.

В 1991 году в MacOS 7 была реализована защита памяти, что существенно повысило устойчивость ОС, но система осталась кооперативно многозадачной. В начале 90-х это еще казалось приемлемым, но к середине 90-х, особенно после выхода на рынок Windows 95, всем, в том числе руководству Apple, стало очевидно, что так жить нельзя.

Тем не менее общие сложности перехода от кооперативной к вытесняющей многозадачности привели к тому, что MacOS версий 8 и 9 оставались кооперативными.

Наряду с MacOS, компания Apple поставляла также систему A/UX, основанную на ядре BSD Unix, но предоставлявшую графический пользовательский интерфейс, аналогичный MacOS, обеспечивающую ограниченную совместимость с бинарными приложениями для MacOS. Проблемы с совместимостью приложений, разумеется, сильно сужали сферу применения A/UX, который использовался практически только на серверах.

Историю того, что сейчас известно под названием MacOS X, следует отсчитывать не от основной линии MacOS, а от проекта Стива Джобса NextSTEP.

В 1985 году один из основателей компании Apple Стив Джобс, после конфликта с новым президентом компании Джоном Скалли, основал собственную компанию и назвал ее NeXT. Вместе с Джобсом из Apple ушли несколько ведущих разработчиков Macintosh и MacOS, большинство из которых участвовали также в работах над неудачным проектом Apple Lisa.

Компания NeXT разработала собственную аппаратную платформу на основе микропроцессора Motorola 68030. В качестве интерфейса для подключения периферийных устройств использовалась шина NuBus, та же, что и в Apple Macintosh. Первая версия платформы имела характерный корпус в виде черного куба и так и называлась NeXTcube.

Для этих компьютеров была разработана ОС NEXTSTEP. Ядро ОС было основано на BSD Mach. Наиболее важными особенностями новой ОС были графический пользовательский интерфейс, основанный на технологии Display PostScript, и объектно-ориентированный инструментарий для разработки приложений на основе языка Objective C (объектно-ориентированное расширение C, альтернативное C++).

PostScript — язык описания векторных изображений, первоначально разработанный компанией Adobe в качестве языка управления лазерными принтерами. В действительности, PostScript представляет собой полнофункциональ-

ный (т. е. эквивалентный машине Тьюринга с конечной памятью) язык программирования.

Использование PostScript для отрисовки изображений на дисплее, разумеется, должно было сильно упростить разработку приложений WYSIWYG. Впрочем, особенности работы с дисплеем в многооконной графической среде потребовали внесения специфических дополнений и расширений в язык.

Компьютеры NeXT вызвали большой интерес у специалистов в области компьютерной техники, но рынок принял их довольно прохладно. Машины были существенно дороже, чем Apple Macintosh сопоставимой конфигурации и при этом уступали им по производительности. Из-за низкой производительности они не могли конкурировать и с рабочими станциями на основе RISC-процессоров, так что президент Sun Скотт МакНили ехидно заметил в 1989 году, что это "компьютер с неправильным процессором по неправильной цене". Привлечь значительных разработчиков прикладного ПО на новую платформу не удалось.

В течение первой половины 90-х годов разработчики NeXT предпринимали несколько попыток сохранить жизнь платформе, главным образом путем конвергенции ее с существующими и уже утвердившимися на рынке ОС. Наиболее близкой к удаче попыткой следует признать проект OPENSTEP, разрабатывавшийся совместно с компанией Sun.

Наконец, в 1997 году акционеры Apple уговорили Джобса вернуться на пост CEO компании. При этом компания NeXT была поглощена Apple, и начались работы по конвергенции NEXTSTEP с MacOS. Первые попытки такой конвергенции — проекты Copland и Taligent (разрабатывавшийся совместно с IBM) — анонсировались с большой шумихой, но с технической точки зрения закончились неудачей.

Наконец, в 1998 году была показана публике первая демо-версия того, что двумя годами позже вышло на рынок под названием MacOS X.

Система представляет собой весьма любопытный сплав разнородных компонентов с совершенно различными условиями лицензирования. Важная часть пакета — это Unix-подсистема, известная как Darwin. Это система с открытыми исходными текстами, публикуемыми на условиях BSD License, основанная на BSD Mach 3.0. Ядро реализует внутреннюю вытесняющую многопоточность и обеспечивает интерфейс системных вызовов FreeBSD, однако использует собственную подсистему ввода/вывода (I/O Kit), не имеющую прямых аналогов в системах семейства BSD Unix. Для мультимедийных приложений предоставляется класс планирования реального времени.

Наиболее важное отличие I/O Kit от подсистем ввода/вывода других ОС состоит в том, что традиционные ОС предполагают разработку драйверов на ассемблере или, в более современных системах, на чистом C. Даже в тех ОС,

где допускается использование C++, сами точки входа функций драйвера должны использовать соглашения о вызовах C. В то же время понятно, что методика реализации драйверов имеет очевидные параллели с объектно-ориентированными технологиями. Драйвер представляет собой структуру данных (блок переменных состояния устройства), с которой связан набор процедур (точек входа драйвера). Это очень похоже на объект с методами в объектно-ориентированных языках программирования, поэтому идея реализовать "объектно-ориентированную ОС" приходит в голову многим, кто только что изучил ООП и операционные системы.

Darwin I/O Kit воплощает эту идею в жизнь. Он предполагает разработку драйверов на специальном диалекте C++ (так называемом Embedded C++, в котором не поддерживаются шаблоны, исключения и множественное наследование, а вместо RTTI предоставляется нестандартный API для получения информации о типах объектов).

I/O Kit предоставляет богатый набор базовых классов, обеспечивающий инициализацию драйвера, его регистрацию в реестре доступных устройств, обмен данными с пользовательскими программами и стандартные "продвинутые" возможности современных ОС, такие как автоматическое распознавание устройств, работа с периферийными шинами PCI, ATA, USB, FireWire, SCSI и др., динамическая подгрузка и выгрузка драйверов, горячее подключение, управление питанием устройства (перевод в режим сниженного энергопотребления и вывод из него) и т. д. Над ядром Darwin надстроена полноценная Unix-система с командным интерпретатором Korn Shell, стандартным набором утилит командной строки и некоторыми полезными усовершенствованиями. Так, функции демонов init и cron традиционных Unix-систем (первый отвечает за инициализацию системы, второй обеспечивает запуск задач по расписанию) объединены в демоне launchd.

Основой графической подсистемы MacOS X является Quartz — графический API, базирующийся на стандарте OpenGL для трехмерных изображений и PDF (Portable Document Format) для двумерных изображений. С технологической точки зрения, Quartz 2D является наследником Display PostScript. PDF представляет собой подмножество языка PostScript, разработанное компанией Adobe для электронной публикации документов.

Распространена легенда, что отказ от PostScript и переход к PDF обусловлен слишком жесткими условиями, на которых Adobe предоставляет лицензии на использование PostScript. Впрочем, есть и другие аргументы в пользу такого перехода. Действительно, PostScript представляет собой полнофункциональный язык программирования. Но для таких задач, как отрисовка двумерных изображений на дисплее, полная программируемость скорее вредна, чем бесполезна, например потому, что на полнофункциональном языке программирования можно написать бесконечный цикл или программу, иным способом

бом потребляющую слишком много ресурсов. Таким образом, полнофункциональный интерпретатор PostScript представляет собой потенциальную точку для атаки отказа сервиса. Некоторые реализации PostScript, например GhostScript, предоставляющие операции для чтения и записи файлов, могут даже использоваться для внедрения вирусов и других троянских программ.

PDF представляет собой подмножество PostScript, эквивалентное базовому языку по графическим возможностям, но лишенное полной программируемости. Таким образом, поведение интерпретатора PDF гораздо более предсказуемо, как по требованиям к ресурсам, так и по безопасности.

Для совместимости с приложениями MacOS 9 система реализует две подсистемы — Carbon и Cocoa. Carbon представляет собой, грубо говоря, бинарный эмулятор MacOS 9, который позволяет исполнять бинарные модули MacOS 9 под OS X без каких-либо изменений. Cocoa представляет собой API, обеспечивающий легкое портирование приложений для традиционной MacOS в новую среду. Приложения Cocoa могут пользоваться как традиционными API, так и новыми возможностями, предоставляемыми Mac OS и Unix-подсистемой.

В отличие от Darwin, графические подсистемы OS X распространяются с закрытыми исходными текстами. Для защиты от неавторизованного копирования (а также, в определенной мере, и для защиты от внедрения троянского кода) ряд ключевых утилит последних версий OS X распространяются в зашифрованном виде. Для загрузки этих утилит подсистема виртуальной памяти ядра включает специальные криптографические модули.

В поставку MacOS также включена подсистема цифрового управления ограничениями (DRM) iTunes, совместимая с портативным цифровым медиаплеером iPod и реализациями iTunes/QuickTime для других платформ.

Первоначально OS X разрабатывалась для компьютеров PowerMac на основе процессоров IBM/Motorola Power. Впрочем, по некоторым сведениям, ОС сразу разрабатывалась как кроссплатформенная, с параллельной поддержкой Power и x86. В 2005 году Apple объявила о планах оставить платформу Power и перейти к изготовлению компьютеров на основе процессоров Intel CoreDuo с архитектурой x86. Главной причиной этого решения, безусловно, являлся тот факт, что процессоры Intel довольно сильно опередили процессоры Power как по абсолютной производительности, так и по отношению производительность/цена.

Новые компьютеры, известные как Intel Mac, обеспечивают исполнение бинарных загрузочных модулей для PowerMac в режиме бинарной эмуляции. Новые версии компиляторов C/C++ позволяют генерировать дуальные загрузочные модули, содержащие два сегмента кода для каждой из поддерживаемых аппаратных архитектур.

Intel Mac не являются PC-совместимыми. Вместо традиционного BIOS они используют новый стандарт EFI (Extensible Firmware Interface). Из-за этого загрузка ОС для PC-совместимых компьютеров на Intel Mac требует специального Boot-менеджера. В настоящее время доступна бета-версия загрузчика BootCamp, который обеспечивает установку и загрузку Windows XP.

Аналогично, OS X в стандартной поставке не может быть загружена на PC-совместимом компьютере. Подсистема Darwin легко может быть модифицирована (и реально была модифицирована) для такой загрузки, но Darwin — это далеко не вся система и даже не самая интересная ее часть. Версии OS X с модифицированным загрузчиком доступны в файлообменных сетях, но Apple не предоставляет технической поддержки пользователям этих версий, в том числе не предоставляет драйверов для работы ОС с периферийными устройствами, для которых нет аналогов в стандартных конфигурациях Intel Mac, и даже по мере сил пытается бороться с их распространителями как с "пиратами".

П1.4. Семейство CP/M

Родоначальником семейства является дисковая операционная система CP/M (Control Program/Monitor) фирмы Digital Research. Первая версия системы была разработана в 1974 году для использования в инструментальных микропроцессорных системах на основе микропроцессоров i8080 и i8085.

Инструментальные микрокомпьютеры, популярные в 70-е годы, использовались как средство кросс-разработки и отладки программ для встраиваемых микропроцессорных систем. Типичная система такого типа состояла из микропроцессорной платы, устройства чтения/записи магнитных или перфолент, а позднее — накопителя гибких дисков и, наконец, видеотерминала. Можно считать их предками персональных компьютеров, но в описываемый период такие системы были слишком громоздки и дороги для домашнего и офисного использования.

CP/M была первой ОС для машин такого рода, обеспечившей возможность использования гибких дисков, поэтому она быстро приобрела огромную популярность и стала стандартом де-факто для микрокомпьютеров [Дейтел 1987]. Система была перенесена практически на все 8- и 16-разрядные и многие 32-разрядные микропроцессоры манчестерской архитектуры. Появившиеся в конце 70-х персональные компьютеры обычно также были ориентированы на использование CP/M. В начале 80-х были реализованы многозадачная и сетевая версии CP/M. Появилось также немало клонов системы, программно совместимых с ней и в целом аналогичных по архитектуре.

С архитектурной точки зрения, CP/M представляет собой довольно типичную однозадачную ДОС, предназначенную для работы на процессоре без диспетчера памяти и средств базовой адресации. К отличительным особенностям CP/M можно отнести следующие:

- своеобразный командный язык, представляющий собой подмножество DCL (DEC Command Language) — командного языка систем RT-11, RSX-11, VAX/VMS . Так, в DCL команды являются полными словами английского языка, но разрешено их сокращение: DIRECTORY, например, может быть сокращена до DIR или даже до DI — в CP/M же команда называется DIR;
- устройства последовательного ввода/вывода обозначаются трехбуквенными аббревиатурами, например TTY: означает телетайп, а LPT: — строчный принтер. Некоторые устройства, например CON: (консоль), LST: (устройство вывода листинга) могут динамически переназначаться;
- диски обозначаются буквами латинского алфавита.

CP/M имеет модульную архитектуру и состоит из трех основных подсистем: командного процессора (Console Command Processor, CCP), базовой дисковой операционной системы (Basic Disc Operating System, BDOS) и базовой системы ввода/вывода (Basic Input/Output System, BIOS). CCP и BDOS представляют собой неизменные компоненты системы, BIOS содержит драйверы физических устройств и подлежит перекомпоновке при каждой перегенерации системы для новой конфигурации аппаратуры. Память, не занятая компонентами системы и таблицей векторов прерываний, называется TPA (Transient Program Area — область пользовательских [дословно — преходящих] программ).

П.1.4.2 MS DOS

— Не помните ли вы Первой Версии Доса, товарищ Ком? — спрашивал Диггер т. Комманда Кома.

— Нет, — плакал тот, — от нее только песня осталась, — и пел: "ОС, ДОС, Пер. Вер. ДОС, а бабушка здорова..."

A. Голубев

В 1981 году фирма IBM анонсировала персональный компьютер IBM PC на основе 16-разрядного процессора i8088. Первоначально предполагалось, что в качестве ОС для этого компьютера будет использоваться CP/M, однако представителям IBM не удалось достичь приемлемых условий соглашения с Digital Research. Ни история, ни легенды не сообщают нам о том, что именно послужило причиной разногласий.

Легенды, однако, сохранили для нас немало подробностей (к сожалению, не очень достоверных) дальнейшего развития событий. Вместо того чтобы пойти на компромисс с Г. Кидалом, CEO Digital Research, представитель IBM обратился к сыну одной из своих старых знакомых, Биллу Гейтсу. Билл Гейтс в это время занимался продажей собственного интерпретатора языка BASIC для любительских 8-разрядных микрокомпьютеров. Билл не имел ни опыта разработки ОС или DOS, ни даже теоретической подготовки в этой области, поскольку был отчислен из колледжа. Однако IBM обещала щедрую предоплату и вообще довольно выгодные условия, поэтому Билл взялся за проект.

Примечание

Нужно отметить, что эта часть легенды несколько расходится с достоверными историческими сведениями. В описываемый период Microsoft занималася не только и даже не столько BASIC, сколько собственной ОС, основанной на Unix v.6 — Microsoft Xenix. Эта система была реализована для нескольких микропроцессорных систем на 32-разрядных микропроцессорах, таких как MC68000, NS32032 и даже i8086/8086.

По другой версии легенды, Билл Гейтс первоначально предложил Xenix, но представители IBM хотели что-нибудь похожее на CP/M.

Гейтс приобрел за 13 тысяч долларов лицензию на систему QDOS — клон CP/M, разработанный компанией Seattle Computer Products. По легенде, QDOS расшифровывается как Quick and Dirty Operating System — "Быстро [сделанная] и Грязная Операционная Система".

Фирма Microsoft переделала загрузчик и дисковую подсистему QDOS для работы с IBM PC и использования сервисов ПЗУ этого компьютера (эти сервисы также называются BIOS, хотя имеют довольно мало общего с BIOS CP/M), и предложила результат фирме IBM. Заказчики оказались довольны, и Билл быстро стал миллионером.

К версии 3.30 MS DOS (такое название получила новая система) уже накопила очень много отличий от оригинальной CP/M. В качестве файловой системы была использована изобретенная лично Б. Гейтсом для применения в интерпретаторе BASIC ФС FAT. Эта ФС была переделана так, чтобы в ней можно было создавать вложенные каталоги. Был добавлен новый формат загрузочного модуля — вдобавок к абсолютным загрузочным файлам формата COM (совместимых с CP/M) были реализованы относительные загрузочные файлы EXE (известные также как файлы MZ). Также реализованы загружаемые драйверы внешних устройств. Динамическая загрузка и выгрузка драйверов не поддерживалась, но, по крайней мере, изменение номенклатуры драйверов теперь не требовало перегенерации системы. Список загружаемых

драйверов задавался текстовым файлом C:\CONFIG.SYS. Позднее был даже реализован интерфейс для драйверов файловых систем.

Была разрешена загрузка нескольких программ в стековом порядке (впрочем, не допускалось их параллельного исполнения). Система приобрела многие черты, аналогичные примитивным версиям Unix, — так, каждая загруженная программа в MS DOS снабжается дополнительным сегментом памяти, так называемым PSP (Program Segment Prefix — заголовок программного сегмента), который аналогичен User area (пользовательской области, сегменту данных ядра ОС, относящихся к конкретному процессу) в Unix. В некоторых документах сегментный адрес PSP даже называют *pid* (Process Identifier, по аналогии с идентификатором процесса в Unix). Как и в Unix, исполнение системного вызова сопровождалось переустановкой стека. В состав системы были включены:

- файловый API, очень похожий на интерфейс файловых системных вызовов в Unix;
- переменные среды;
- переназначение ввода/вывода;
- и даже конвейеры (последовательности задач, в которых поток стандартного вывода предыдущей задачи является потоком стандартного ввода следующей), реализуемые через промежуточный файл.

По одной из легенд, нежелание Microsoft реализовать в DOS вытесняющую многозадачность обусловлено не столько технической некомпетентностью, сколько соглашением с фирмой SCO — в соответствии с ним, передавая права на торговую марку Xenix, Microsoft обязалась не разрабатывать и не продавать функциональных эквивалентов UNIX. Такое соглашение существовало в действительности: через много лет, в 1997 году, SCO отсудила у Microsoft принадлежавшую последней долю своих акций и ряд других обязательств (упоминание Microsoft в списке держателей авторских прав, поддержку бинарной совместимости с Xenix/286) на том основании, что Microsoft рекламировала Windows NT как замену и даже "убийцу" UNIX и, таким образом, нарушила соглашение.

Со времен DOS 3.30 архитектура системы не подверглась сколько-нибудь заметным изменениям [Панкратов 2001]. Так, DOS 7, входящая в состав Windows 98/ME в качестве вторичного загрузчика, отличается от 3.30 практически только поддержкой файловой системы FAT32.

Digital Research не смотрела на это развитие безучастно. К концу 80-х система DR DOS, основанная на исходных текстах оригинальной CP/M, обеспечивала полную программную совместимость с MS DOS и включала в себя все новшества не только версии 3.30, но и более поздних версий.

Ряд полезных идей, впервые реализованных в DR DOS, такие как загрузка в верхнюю память, условные операторы в CONFIG.SYS и упакованная файловая система, появились в MS DOS лишь на одну или две версии позже, а некоторые идеи — например, экранный редактор командной строки (возможность, знакомая пользователям командных процессоров DCL, bash и 4DOS/4OS2/4NT) и загрузка из расширенного раздела — вообще не были реализованы в MS DOS и Windows 95/98/ME.

П1.4.2. Расширители DOS

В 1987 году компания Intel выпустила на рынок 32-разрядный процессор 80386, который предоставлял линейную 32-разрядную адресацию и два режима совместимости с 8086: "реальный режим", в котором процессор полностью имитировал работу 8086 и структуру его адресного пространства, и "виртуальный 8086", в котором структура адресного пространства 8086 воссоздавалась в пределах одной задачи.

Выход новой архитектуры на рынок оказался длительным и болезненным. Новыми возможностями процессора могли воспользоваться лишь новые ОС, но для того чтобы оправдать приобретение новой ОС, необходимы были новые приложения и новые системы программирования. Но мало кто рисковал разрабатывать новые приложения для ОС, которые занимали лишь небольшую и медленно растущую долю рынка. По этой же причине производители оборудования неохотно писали драйверы для новых ОС, что, разумеется, не способствовало принятию этих ОС пользователями.

Большинство потребителей использовали машины на основе 80386 под управлением DOS, т. е. как быстрые 8086.

Уже в конце 80-х на рынке появились среды программирования, позволявшие запускать 32-разрядные программы из-под DOS, так называемые расширители DOS (DOS extender). В действительности, среда исполнения таких программ представляла собой род операционной системы, которая использовала DOS в качестве вторичного загрузчика. Загрузившись, расширитель переводил процессор в защищенный режим и создавал для DOS задачу в режиме "виртуального 8086", так что DOS могла продолжать работу. Благодаря этому расширитель мог использовать системные вызовы DOS для работы с файловыми системами и устройствами, поддерживаемыми DOS и BIOS.

Расширители быстро завоевали популярность среди производителей ресурсоемких программ — например, пакетов автоматизированного проектирования и игрушек. Кроме того, реализации C/C++ и Fortran для расширителей были очень популярны среди ученых, которые использовали эти среды программирования для разработки вычислительных программ.

Среди разработчиков вычислительных программ известен расширителем Phar Lap, первоначально поставлявшийся одноименной компанией. Затем эта компания была приобретена компанией Ardent. Среди разработчиков и пользователей игровых программ более популярны расширители компаний Rational — DOS4G и DOS4GW (последний поставлялся с компиляторами Watcom). Доступен также бесплатный расширителем djgr.

Расширители первых поколений были очень капризными и не умели мирно сосуществовать друг с другом. Так, широко известен нестандартный расширителем Voodoo Memory Manager компании Origin, на основе которого были реализованы такие популярные игры, как Ultima VI, Ultima Underworld, Privateer. Эти игры могут работать практически только под чистым DOS.

В 1991 году был принят стандарт DPMI (DOS Protected Mode Interface), который определял требования к расширителю DOS и обеспечивал сосуществование нескольких расширителей в одной оперативной памяти. Одним из наиболее важных требований этого стандарта было то, что расширитель при загрузке обязан был проверять наличие других работающих расширителей и менеджеров памяти. Обнаружив такие программы, расширитель должен пользоваться их сервисами, а не пытаться управлять процессором и памятью самостоятельно.

В действительности, вопросы взаимодействия между расширителями регулируются несколькими стандартами разного происхождения (DPMI, VCPI и XMS). Соответствующий корпус оккультных знаний слишком ужасен, чтобы подробно излагать его в этой книге, и представляет мало практического интереса для современного читателя.

Несколько расширителей, поддерживающих стандарт DPMI или определенные его подмножества, могут одновременно работать на одной машине. Более важным в исторической перспективе следствием оказалось то, что большинство программ, написанных с использованием DPMI, оказались работоспособны в DOS-эмulateрах OS/2, Windows NT и Linux и DOS-окнах Windows 95.

В начале 90-х годов XX века компания Quarterdeck реализовала менеджер расширенной памяти в виде драйвера DOS qemm386.sys. Этот драйвер обеспечивал управление оперативной памятью за пределами первого мегабайта и эмуляцию LIM EMS; кроме того, он мог отображать оперативную память на адреса в диапазоне от 640 Кбайт до 1 Мбайт. В обычных IBM PC-совместимых компьютерах этот диапазон адресов используется для BIOS и отображенных на ОЗУ устройств, но обычно большая часть этих адресов не используется. Отображенное на эти адреса ОЗУ называется верхняя память (Upper memory). Qemm386 позволял загружать в эту память драйверы DOS и небольшие резидентные программы, освобождая таким образом нижнюю (lower) или, что то же самое, обычную (conventional) память, т. е. ОЗУ с адрес-

сами ниже 640К. Большинство программ для DOS могло использовать только нижнюю память.

Qemm386 поставлялся на коммерческой основе в виде самостоятельного продукта и в составе пакета DesqView (многозадачной надстройки над DOS). В 1993 году этот драйвер был также включен в поставку DR DOS.

Несколько позже Microsoft реализовала функциональный аналог qemm386 — драйвер emm386.sys. Этот драйвер был включен в поставку MS DOS 6.0 и Windows 3.1.

Ближе к середине 90-х годов стало очевидно, что дни DOS как платформы сочтены (впрочем, мало кто ожидал в то время, что в той или иной форме эта платформа сможет прожить до самого конца столетия). В 1993 году Digital Research вместе с авторскими правами на DR DOS была приобретена фирмой Novell. В 1995 году, вскоре после того как собрание акционеров выгнало Р. Нурду с поста CEO, авторские права на этот продукт были переданы созданной им компании Caldera. Несколько позднее Caldera опубликовала исходные тексты системы на условиях GPL под названием Caldera OpenDOS [www.caldera.com]. С тех пор эта система широко используется в составе DOS-эмуляторов различных дистрибутивов Linux, а также в качестве первичного загрузчика при загрузке некоторых конфигураций Linux с дискет (затем используется специализированная версия вторичного загрузчика, LOADLIN.EXE)

П1.4.3. Win16

Сидят как-то Диггер, командир Нортон и т. Комманд Ком, а мимо идет отец Виндоуз. "Сидите, — говорит, — Ну, ну. Скоро все подо мной сидеть будете". Никто ему так ничего и не ответил. Один т. Комманд Ком сплюнул и грязно выругался. "Проходи, проходи, — сказал, — оболочка дешевая".

А. Голубев

Вскоре после анонса Apple Macintosh в 1984 году, Microsoft выпустила электронную таблицу Excel и текстовый процессор Word для этой системы. Автор не может подтвердить это официальными данными, но трудно избавиться от впечатления, что основной задачей при разработке Win16 было максимальное облегчение переноса приложений Mac на IBM PC.

Версии Windows 2.x—3.x воспроизводят почти все характерные черты Mac OS:

- событийно-ориентированную кооперативно многозадачную архитектуру;
- единое адресное пространство;
- сборку программ в момент загрузки с использованием DLL;

- "ручечное" управление памятью;
- даже соглашение о вызовах у процедур системного API: параметры помещаются в стек, начиная с первого, стек очищается вызываемой процедурой.

Ядро системы было собрано в виде загрузочного модуля DOS (Win.exe). После загрузки этот модуль брал на себя управление памятью и осуществлял загрузку собственных и пользовательских модулей формата NE (так называемые сегментированные модули). DOS, однако, сохранялась в оперативной памяти и использовалась в качестве дисковой подсистемы.

Первые версии системы были совершенно неудовлетворительными не только с точки зрения надежности, но и по производительности. Довольно большие требования к ресурсам не позволяли запустить сколько-нибудь ресурсоемкое приложение в 640К ОЗУ, системы же с большим объемом памяти были в то время редкостью. Большой объем памяти был доступен только на машинах IBM PC AT с процессором 80286. На таких компьютерах обращения к DOS требовали переключения в реальный режим процессора и поэтому происходили очень медленно.

Значительный прорыв в эксплуатационных характеристиках Windows 3.x обеспечил процессор 80386, на котором можно было создать для DOS *виртуальный 8086* (см. П2.2.2.1). Это позволило избежать сброса процессора на каждом системном вызове и резко повысило производительность. Еще большее повышение производительности было достигнуто в Windows 3.11 с появлением так называемого *32-разрядного доступа к диску* — собственной дисковой подсистемы, которая работала целиком в защищенном режиме. Тем не менее надежность даже этих версий системы оставляла желать много лучшего.

В Win16 впервые была реализована технология, без упоминания которой описание этой системы было бы не полным — не только потому, что это одна из немногих оригинальных концепций, впервые реализованных в системах семейства СР/М, но и потому, что эта технология оказала значительное влияние на современные методики разработки прикладного программного обеспечения. Речь идет о технологии COM (Common Object Model — общая объектная модель).

Идея, лежащая в основе СОМ, довольно проста и решает весьма насущную проблему: точки входа DLL не хранят сведений не только о семантике соответствующих процедур, но даже о количестве и типах параметров, передаваемых этим процедурам. Различные системы программирования используют разные соглашения о способе передачи параметров — заголовок DLL не хранит информации и об этом. Отсутствие перечисленных сведений затрудняет

взаимодействие между подсистемами, реализованными на разных языках, и делает невозможным синтаксическую проверку допустимости вызова внешних процедур из интерпретируемых языков.

СОМ предполагает снабжение DLL внешним описателем, который перечисляет все процедуры, реализуемые данной DLL, и типы данных, используемые этими процедурами. Описание формируется на специальном языке IDL (Interface Definition Language — язык описания интерфейса), который затем компилируется в двоичное представление, используемое объектными средами и интерпретирующими системами программирования. Современные системы программирования выполняют автоматическую генерацию IDL и "болванок" (заготовок) кода, реализующего данный интерфейс, на конкретном языке программирования.

IDL является довольно простым языком, на котором можно описывать объекты — структуры данных, с которыми ассоциированы наборы процедур-методов. Поля структур (атрибуты) могут принадлежать одному из нескольких скалярных типов (целое число, число с плавающей точкой, дата и время, строка — последний тип в большинстве компилируемых ЯВУ не является скалярным). Допустимы также атрибуты, являющиеся объектами других классов. Методы объекта в качестве параметров могут получать как значения скалярных типов, так и объекты, и используют стандартное соглашение о вызовах — тем самым облегчается взаимодействие подсистем, реализованных на разных языках программирования.

Объектно-ориентированный стиль описания интерфейсов является популярной методологией описания и разработки сложных программных систем, поэтому, несмотря на многочисленные недостатки технологии СОМ (например, не поддерживается контроль версии интерфейса и наследование), она была хорошо принята сообществом разработчиков. Впрочем, заявленная в начале работ над этой технологией цель — переход от монолитных приложений к компонентным средам, составляемым из взаимозаменяемых объектов СОМ — достигнута не была. Достижению этой цели не способствовала также техническая политика Microsoft, состоявшая в низкокачественной и неполной документации и хаотических сменах флагманской объектной среды (версии OLE, ActiveX, OCX и т. д.).

П1.4.4. OS/2 1.x

Параллельно с развитием Win16, во второй половине 80-х годов Microsoft занималась разработкой еще одной операционной системы, в данном случае совместно с фирмой IBM. OS/2 создавалась как ОС для новой серии машин IBM Personal System/2, основанных на процессоре 80286. Архитектура системы представляет собой самое полное из известных автору воплощение идей, которые имел в виду Intel, разрабатывая этот процессор. Весьма ограничен-

ный успех этой системы обусловлен, по-видимому, несостоятельностью идей Intel, а не качеством их воплощения.

Система использует сегментированную виртуальную память и сборку в момент загрузки. Формат загрузочных модулей и DLL тот же самый, что в Win16 — NE. Однако система имеет раздельные адресные пространства — задачи не имеют доступа к сегментам данных и приватным сегментам DLL других задач. Сегменты кода — разделяемые и защищены от записи. К сожалению, 80286 не обрабатывал сегментных отказов, поэтому виртуальная память использовалась лишь для защиты задач друг от друга, но не для сегментной подкачки [Коган/Роусон 1989, Лафо/Нортон 1991].

OS/2 реализует вытесняющую многозадачность, многопоточность в пределах одной задачи и богатый набор примитивов взаимоисключения (семафоры как двоичные, так и счетчики, очереди сообщений). Ядро — кооперативно многозадачное с управляемыми сообщениями асинхронными драйверами. Одной из отличительных особенностей системы является мощный механизм обработки исключений, аналогичный используемым в MVS-OS/390-z/OS и VMS.

Одной из главных задач при разработке системы было максимальное облегчение переноса программного обеспечения (как прикладного, так и системного, включая и драйверы устройств) из MS DOS. Эта цель была в основном достигнута: все системные вызовы DOS имели полные функциональные эквиваленты в OS/2, и достаточно аккуратно написанные программы для DOS могли быть перенесены в OS/2 1.x простой перекомпиляцией.

Впрочем, оказалась неразрешимой другой, более важной задача — обеспечение бинарной совместимости. Процессор 80286 в защищенном режиме не имел возможности исполнять программы для реального режима 8086. Для исполнения бинарных модулей DOS была нужна полноценная копия DOS и переключение режима процессора. Таким образом, в системе могла использоваться только одна сессия DOS, а во время ее работы вся активность приложений OS/2 полностью прекращалась.

Из-за этого недостатка OS/2 1.x имела успех лишь в качестве серверов файлов и печати в сетях NETBIOS (LAN Manager) и серверов приложений: Lotus Notes, Sybase и др.

Бинарная несовместимость с DOS могла быть преодолена только с использованием возможностей процессора 80386. Существовали и другие показания к переходу на этот процессор: например, возможность страничной подкачки. Кроме того, используемая в x86 плоская модель памяти упрощает программирование, снимает ограничение в 64 Кбайт на переменную и дает много других преимуществ.

В этот момент между партнерами возникли серьезные разногласия в вопросе о том, как следует переходить на новый процессор. Предложенная фирмой Microsoft архитектура новой 32-разрядной версии системы (OS/2 New Technology) оказалась абсолютно неприемлемой для IBM.

Камнем преткновения стал вопрос о том, как следует организовывать взаимодействие между 16-разрядным кодом, использующим сегментированную память, и 32-разрядным, использующим линейное пространство.

На самом деле, адреса в обеих моделях памяти имеют длину 32 бита, но в 16-разрядной модели адрес разбит на селектор сегмента и смещение в нем. Это разбиение накладывает серьезные ограничения на указательную арифметику. Задача преобразования 16-разрядного указателя 80286 в 32-разрядный достаточно проста; задача же обратного преобразования требует нетривиальных вычислений и в общем случае во время исполнения неразрешима.

Предложение Microsoft состояло в том, чтобы сохранить бинарную совместимость с программами для OS/2 1.x и дать им возможность обращаться к новым 32-разрядным DLL и системным модулям, но не предоставлять возможности для 32-разрядных приложений обращаться к старым 16-разрядным DLL. Это решение требовало полной переделки всех сервисных подсистем (включая графическую подсистему Presentation Manager), ядра ОС и подсистемы ввода/вывода (т. е. всех драйверов) в 32-разрядную модель памяти. Переделка драйверов требовала отказа от совместимости с существующими драйверами устройств, файловых систем и сетевых протоколов для OS/2 1.x.

IBM предложила более элегантное решение, требовавшее, однако, переделки компилятора. Операционная система предоставляла так называемую плиточную память (tiled memory), отображая 512 Мбайт ОЗУ одновременно как 8196 сегментов 80286 и как соответствующий объем плоского адресного пространства x86. Таким образом, и 32-разрядные программы x86, и 16-разрядные программы 80286 могли разделять общую память, передавать указатели на объекты в этой памяти и вызывать друг друга. Для нормальной работы с такой памятью предлагалось научить компилятор при вызове из 32-разрядного кода 16-битной процедуры генерировать специальный код, осуществляющий преобразование "плоского" указателя в сегментированный для всех параметров-указателей (пример П1.2). Компилятор должен был принимать решение о необходимости такого преобразования на основе прототипа вызываемой функции. Указатели 80286 в формате `сегмент:смещение` обозначаются в языках C/C++ ключевым словом `far` (например, `char far * ptr;`); указатели, при описании которых не использовалось никаких ключевых слов, считаются плоскими 32-разрядными.

Пример П1.2. Код, порождаемый компилятором IBM Visual Age C++ при вызове 16-разрядной функции

```
; 104    /*
; 105    * открыть обработчик
; 106    *для мыши
; 107    */
; 108    MouOpen (NULL, &hmou) ;

sub    esp, 038h
push   0h
mov    eax, offset FLAT:hmou
call   _DosFlatToSel
push   eax
push   08h
push   eax
mov    eax, offset FLAT: MOU16OPEN
call   _DosFlatToSel
xchg  eax, dword ptr[esp]
push   0h
call   __EDC3216
add   esp, 04ch
```

Сведений о ходе переговоров история не сохранила, однако по косвенным признакам они были весьма бурными. По причинам, изложенным в главе 10, решение об отказе от поддержки существующих драйверов и DLL было абсолютно неприемлемо для IBM. Почему переделка компилятора не устраивала Microsoft, менее понятно. Я не владею достоверными сведениями на этот счет, но есть ряд косвенных оснований предполагать, что взаимодействие между подразделениями Microsoft оставляет желать много лучшего, так что создатели ОС попросту не имели возможности (или даже права) выдвигать столь сложное требование к разработчикам компилятора. Возможно, на результат переговоров повлияли и какие-то другие, например, сугубо политические или даже психологические факторы.

Важно отметить, впрочем, что никаких реальных проблем реализация данного требования не представляла: практически все 32-разрядные компиляторы для OS/2 2.x — Zortech C++, Watcom C++, IBM C/Set (позднее IBM Visual Age for C++) — с успехом выполняют преобразование указателей.

Так или иначе, переговоры не только проходили бурно, но и закончились разводом. Дальнейшая судьба OS/2 — это совсем другая история или, точнее сказать, две разные истории.

П1.4.5. IBM OS/2

Каков неофициальный символ OS/2 в России?
Пьяный опустившийся фидошник?

Обмен репликами на форуме <http://os2.in.ru>

Первая 32-разрядная версия OS/2 2.0 широко использовала плиточную память для организации взаимодействия между 16- и 32-разрядным кодом и представляла собой сочетание 32- и 16-разрядных подсистем. Так, подсистема ввода/вывода была полностью 16-разрядной и тем самым обеспечивала полную совместимость со старыми драйверами и другими модулями ядра.

Тем не менее система в полной мере использовала преимущества, предоставляемые новым процессором, такие как страничная подкачка и режим виртуального 8086 [Минаси/Камарда 1996]. Реализованный в OS/2 2.x эмулятор DOS является одним из крупнейших достижений в сфере разработки виртуальных машин — фирма IBM имеет немалый опыт создания, поддержки и эксплуатации систем виртуальных машин для System/370-390 — и, безусловно, он остается лучшим в мире эмулятором DOS на момент написания книги. Конкуренцию ему может составить разве что находящийся в стадии публичной беты кроссплатформенный эмулятор DOSBox [[sourceforge.net dosbox](http://sourceforge.net/dosbox)], оптимизированный для исполнения старых игровых программ. Впрочем, версия DOSBox доступна также и для OS/2.

Для сравнения, эмулятор DOS в Windows NT/2000/XP уступает ему как по возможностям настройки, так и по универсальности; сессия DOS в Windows 95/98/ME не является эмулятором — запущенное в этой сессии приложение имеет возможность модифицировать критичные для системы данные и проблемы в этом приложении часто приводят к необходимости перезапуска всей ОС, иногда даже холодного. Про эмуляторы DOS в SVR4/x86 и Linux я могу сказать лишь словами поэта:

Иных не стану поминать,
Они под солнцем хладным зреют.
Бумаги даже замарать
И то, как надо, не умеют.

С. Есенин

Система имеет объектно-ориентированный пользовательский интерфейс, основанный на компонентах SOM (System Object Model). Еще одной, менее

известной, но не менее важной на мой взгляд, уникальной особенностью IBM OS/2, является возможность установки пользовательской программой собственного обработчика страничных отказов. Данная особенность уникальна — во всяком случае, среди известных мне промышленно используемых ОС, и позволяет реализовать в пользовательском адресном пространстве функции, которые в других ОС могут исполняться только модулями ядра. Так, свободно распространяемая библиотека EMX использует этот механизм для реализации полного функционального аналога системного вызова `fork` ОС семейства Unix; известен ряд реализаций отображения файлов в адресное пространство памяти.

Развитие системы сопровождалось постепенной заменой 16-разрядных подсистем на 32-разрядные. В версии 4.5 — Warp Server for e-Business — была наконец-то реализована 32-разрядная подсистема ввода/вывода, и это позволило реализовать для OS/2 журнальную ФС JFS. Первая версия JFS была разработана для IBM AIX в 1992 году; в 1998 году для OS/2 была разработана JFS 2, существенно отличающаяся от JFS 1 по возможностям и по формату структур метаданных. Позднее JFS 2 была портирована в AIX, а еще позднее, в 1999 году, исходные тексты JFS были опубликованы и она была портирована для использования в Linux. Реализации JFS 2 для Linux могут монтировать дисковые тома, созданные под OS/2, однако в форумах мне доводилось слышать жалобы на то, что утилита CHKDSK для OS/2 странно реагирует на маски прав доступа, а также жесткие и символические связи, созданные из-под Linux.

В версии 4.0 появился, а в версии 4.5 был включен в стандартную комплектацию стек TCP/IP, совместимый с BSD 4.4, с поддержкой IPSec и фильтрации пакетов [[redbooks.ibm.com sg245393](http://redbooks.ibm.com/sg245393)].

Первые версии системы отличались большими по тем временам требованиями к ресурсам (для нормальной работы нужно было около 16 Мбайт ОЗУ, по меркам начала 90-х — чрезвычайно много), и поэтому тоже имели успех преимущественно в качестве серверов. Некоторые мелкие улучшения позволили в версии 3.0 снизить минимальные требования до 8 Мбайт. Параллельно шло снижение цен на оперативную память, поэтому шансы OS/2 на получение массового признания в качестве ОС для настольного компьютера все возрастили и достигли максимума примерно в 1995—1996 годах.

В это время, однако, подразделение персональных систем IBM увлеклось другим проектом, на который сообщество пользователей OS/2 также возлагало большие надежды, — OS/2 for PPC. Дело в том, что в описываемый период и среди пользователей, и среди производителей вычислительных систем преобладала точка зрения, что Intel исчерпал резервы повышения производительности своих процессоров и не может ни поднять тактовую частоту ЦПУ

выше 60—80 МГц, ни значительно повысить количество операций, исполняемых за один такт. Единственной перспективой повышения производительности представлялись RISC-архитектуры, в том числе — разрабатываемый совместно компаниями IBM, Apple и Motorola микропроцессор архитектуры PowerRISC. Первые процессоры с такой системой команд были разработаны фирмой IBM для рабочих станций и серверов серии RS/6000, но реализовались на нескольких микросхемах. Однокристальный процессор и соответствующие микросхемы окружения (адAPTERЫ системной и периферийной шин и т. д.) должны были резко снизить стоимость системы, переведя ее из категории рабочих станций в персональные компьютеры.

IBM рассчитывала воспроизвести успех IBM PC, опубликовав полные спецификации и таким образом привлечь производителей клонов новой архитектуры, получившей название Power PC. Для этой системы была начата разработка новой версии OS/2. Бинарная совместимость с существующим кодом на новом процессоре была, конечно же, невозможна, поэтому IBM с легким сердцем пошла на пересмотр архитектуры. Новая система должна была стать полностью 32-разрядной и микроядерной, совместимой с OS/2-x86 лишь на уровне исходного кода приложений.

К несчастью, пока новая система разрабатывалась, Intel все-таки посрамил скептиков и преодолел барьеры в 60, а затем и 100 МГц, и выпустил новое семейство суперскалярных микропрограммируемых ядер — сначала двухкристальные сборки Pentium Pro и Pentium II (на самом деле тот же Pro, только с более удачной конструкцией корпуса) и, наконец, однокристальный Pentium III. Новое ядро позволило исполнять по несколько команд за такт и перейти второй якобы непреодолимый для x86 барьер.

Для проекта PowerPC это было крахом. Хотя система по-прежнему превосходила машины на основе x86 как по абсолютной производительности, так и по отношению производительности к цене, теперь разница была не настолько велика, чтобы оправдать для конечного пользователя переход на новую архитектуру и отказ от старых приложений. Микропроцессоры Power имели успех в составе новой линии персональных компьютеров Apple (PowerMac) и рабочих станций, но не смогли составить конкуренции на рынке PC. Эталонная реализация спецификаций PPC была выпущена и изготавливалась небольшими сериями, вышли версии Linux, Solaris и Windows NT 4.0 для новой архитектуры, но коммерческого успеха машина не имела и не могла оправдать разработку полностью новой ОС.

Работы по OS/2 for PPC были свернуты. Разочарование руководства IBM оказалось столь сильным, что были также прекращены работы по другим перспективным технологиям — объектной модели SOM (System Object Model) и

стандарту OpenDOC. Чувствуя потерю интереса к системе со стороны ее поставщиков, многие разработчики приложений отказались от ее поддержки.

К моменту написания книги слухи о смерти OS/2 сильно преувеличены. В 2001 году был выпущен совместный продукт IBM и Serenity Systems — eComstation [www.ecomstation.com], клиентская версия системы, основанная на ядре версии 4.5.

В 2002 году была закрыта возможность свободного скачивания обновлений на сайтах IBM, однако разработка этих обновлений продолжилась — они распространялись по платной подписке среди зарегистрированных пользователей системы. В числе этих обновлений необходимо упомянуть поддержку шины USB и довольно широкого спектра устройств, подключавшихся к этойшине, а также файловой системы UDF (в том числе с возможностью форматирования дисков CD-RW и DVD и записи файлов на эти диски с помощью обычных файловых операций).

Наибольшую опасность для пользователей устаревшей ОС представляет прекращение поддержки со стороны разработчиков оборудования, выражющееся в том, что они перестают писать драйверы для новых устройств. Частично это было скомпенсировано методом "народной стройки" и портированием драйверов из Linux.

Организация "народной стройки" значительно облегчается тем обстоятельством, что исходные тексты значительной части драйверов поставлялись в составе DDK (Driver Development Kit), который долгое время был свободно доступен для скачивания с сайтов IBM. Так, в 1998 году Даниела Энгерт портировала инициализационный код для нескольких популярных контроллеров ATA, не поддерживавшихся в то время IBM, и включила его в исходные тексты драйвера IBM1S506.ADD, которые взяла из DDK. В настоящее время проект Danis506, координатором которого она является, обеспечивает поддержку практически всех встречающихся в новых компьютерах контроллеров ATA (в том числе и SATA), рекомендуется отделом технической поддержки IBM в качестве альтернативы стандартному драйверу и включен в дистрибутив eComStation.

Широкий спектр современных звуковых адаптеров поддерживается драйвером UniAudio, который основан на аналогичных модулях ядра Linux.

Поддержка современных видеоадаптеров осуществляется компанией SciTech Software на коммерческой основе, но они поддерживают практически все популярные современные видеоадаптеры.

Ряд других подсистем ядра OS/2, в том числе файловая система JFS, доступны в исходных текстах. Доступна бета-версия вторичного загрузчика, обеспечивающая загрузку системы с JFS.

Поддержка приложений оставляет желать много лучшего, однако для OS/2 доступны порты многих серверных приложений с открытыми исходными текстами, таких как Apache, MySQL, Sendmail, Bind. Из клиентских приложений необходимо упомянуть Mozilla Firefox, Mozilla Thunderbird, GIMP, tETeX, WarpVision (утилиту для просмотра видеофайлов, в значительной мере основанную на видеоплеере MPlayer для Linux). Доступен также ряд игр, построенных на основе кроссплатформенной библиотеки SDL, в том числе Open Transport Tycoon Deluxe. Некоторые программы для Win32 (к сожалению, Microsoft Office в их число не входит) работают под Odin — эмулятором Win32, частично основанным на исходных текстах Wine. В числе этих программ необходимо упомянуть Acrobat Reader, Macromedia Flash Player, OpenOffice/Win32 и ряд видео- и аудиокодеков — благодаря последним WarpVision без проблем проигрывает практически произвольные современные мультимедийные данные. Таким образом, система остается вполне пригодна для легкого офисного и домашнего использования и в качестве сервера приложений.

Недостатки, связанные с ограниченным набором доступных приложений, в определенной мере компенсируются безопасностью системы: OS/2 неуязвима для вирусов, червей и прочего malware, предназначенных для Win32 и Linux, и практически неуязвима для основной массы "кроссплатформенного" malware.

В 2000—2005 годах можно было отметить признаки превращения системы в маргинальный, но жизнеспособный проект с открытыми исходниками. Сотрудник IBM Скотт Гарфинкель неофициально занимался поддержкой ядра системы и довольно часто публиковал новые версии ядра на FTP-сайте [testcase.ibm.com](http:// testcase.ibm.com). Он принимал патчи (в том числе и патчи для бинарных модулей) от сторонних разработчиков и довольно бурно обсуждал возможные нововведения с пользователями в списках рассылки и группах Usenet. Именно к этому периоду относится появление таких довольно-таки экзотических новшеств, как возможность загрузки одноименных DLL с их идентификацией полным путем (ключ LIBPATHSTRICT).

В 2005 году IBM прекратила прямые продажи дистрибутивов системы с довольно странной формулировкой, сводившейся к тому, что "если вам все-таки нужна система, звоните, попробуем о чем-нибудь договориться". Тем не менее через компанию Serenity Systems до сих пор можно приобрести дистрибутивы системы и поддержку как стандартных пакетов Warp 4 и Warp Server, так и самостоятельно разработанного дистрибутива eComStation. По-видимому, более серьезным ударом для пользователей системы может оказаться переход Гарфинкеля на другую работу, о котором было объявлено в ноябре 2005 года.

В 2004—2005 годах было предпринято несколько попыток организовать сбор подписей под петициями с просьбой опубликовать исходные тексты системы, которые IBM на официальном уровне проигнорировала. Сложно сказать, что именно препятствует публикации исходников — должно понятые коммерческие интересы или обязательства перед контракторами, которые разрабатывали некоторые из подсистем ядра, прежде всего перед Microsoft.

По-видимому, наиболее важный урок, который можно извлечь из всей этой истории, — Ричард Столлмэн, Эрик Рэймонд и другие защитники принципа открытых исходных текстов во многом правы.

Действительно, самая большая опасность для пользователя программной системы с закрытыми исходными текстами — это прекращение поддержки со стороны производителя. Особенно велика такая опасность для пользователей "коробочных" продуктов, которые покупают лицензию на использование продукта, но не контракт на его поддержку. Поэтому само по себе опасение, что производитель может разориться или бросить данный конкретный продукт, может заставить потенциальных покупателей отказаться от его покупки — т. е. такое опасение является в значительной мере самосбывающимся предсказанием. Поэтому тактика FUD (Fear, Uncertainty [and] Doubt — [распространение] страха, неуверенности и подозрений) была важной частью маркетинга IBM в 60-е и Microsoft в 90-е годы XX века. При этом именно FUD нанес наибольший удар по OS/2, т. е. IBM в значительной мере оказалась поражена своим же собственным оружием.

Страх перед прекращением поддержки приводит к тому, что пользователи предпочитают покупать наиболее популярный продукт на рынке — не столько потому, что этот продукт хорош или подходит для них, сколько потому, что вероятность, что этот продукт будет брошен, ниже, чем у конкурентов. Этот эффект играет большую роль, даже если производитель наиболее популярных продуктов не прибегает к открытому FUD.

Неоклассические экономисты называют это "сетевым эффектом" — само по себе наличие большой пользовательской базы делает продукт более привлекательным. На рынке ПО с закрытыми исходниками сетевые эффекты чрезвычайно сильны (хотя большинство экономистов не совсем верно представляет себе причины этого явления), поэтому данный рынок очень сильно подвержен монополизации, гораздо сильнее, чем любой другой известный в истории рынок.

Но против продуктов с открытыми исходниками FUD крайне неэффективен: действительно, даже если с разработчиком продукта что-то случится, пользователи смогут либо скинуться и нанять новых разработчиков для продолжения поддержки и развития продукта, либо — если повезет — организовать такую разработку методом "народной стройки".

При этом, как опять-таки показывает история OS/2, разработчик продукта с закрытыми исходниками может быть не в состоянии опубликовать его исходные тексты. Поскольку сложные продукты часто разрабатываются с привлечением субконтракторов, публикация текстов требует согласия всех этих субконтракторов. Во многих случаях получение такого согласия требует значительного времени и средств, которых у разоряющейся компании или в бюджете подготовленного к закрытию подразделения компании чаще всего просто не оказывается.

Поэтому в современных условиях шансы внедриться на широкий рынок или, наоборот, избежать вытеснения с рынка имеет только продукт с открытыми исходными текстами. Поэтому же многие коммерческие компании, не имеющие доминирующего положения на рынке или рискующие потерять такое положение, публикуют исходные тексты своих продуктов — из ярких примеров можно назвать проекты Mozilla, StarOffice/OpenOffice, Open Solaris, деятельность компании Borland. Mozilla и StarOffice/OpenOffice дают замечательные примеры того, как публикация исходников позволила вдохнуть новую жизнь в продукт, казалось бы неостановимо терявший популярность. Даже Microsoft при продвижении на рынок новых технологий — в частности, многих компонентов .NET — прибегает к раскрытию исходных текстов.

Важным аргументом против открытых исходников часто ошибочно считают опасность неконтролируемого "ветвления" проекта, когда разные группы разработчиков начинают вести развитие продукта в несовместимых направлениях. Среди знаменитых примеров такого ветвления следует упомянуть ветвление проекта BSD Unix на FreeBSD, OpenBSD, NetBSD и коммерческий BSDI, а также отпочкование проекта Linux от Minix.

Но, как мы видели в этом разделе, коммерческие проекты, ведущиеся с закрытыми исходными текстами, также не застрахованы от ветвления, но последствия этого ветвления оказываются гораздо более разрушительными, а возможность слияния ветвей в будущем оказывается практически исключена.

П1.4.6. Windows NT/2000/XP

Гордый профиль, твердый шаг,
Со спины — дак чистый шах!
Только сдвинь корону набок,
Чтоб не висла на ушах!..

Л. Филатов

Наработки Microsoft по OS/2 New Technology были в 1993 году выпущены на рынок под названием Windows NT. Версии 3.x и 4.0 этой системы обеспечивали совместимость с 16-разрядными приложениями для OS/2 1.x в отдельной подсистеме, без возможности обращаться из 16-разрядных приложений к 32-разрядным DLL и наоборот.

В описываемый период из DEC в Microsoft в полном составе перешла команда разработчиков ядра VMS под управлением Д. Катлера. Microsoft широко рекламировал этот факт и утверждал, что Windows NT находится с VMS в гораздо более близком родстве, чем с OS/2 1.x. Из табл. П1.1 видно, что это утверждение не очень-то согласуется с действительностью.

Таблица П1.1. Сравнение OS/2 1.2, Windows NT и VMS

	OS/2 1.x	Windows NT 3.x	VMS
Многозадачность	Вытесняющая	Вытесняющая	Вытесняющая
Ядро	Монолитное	Монолитное	Монолитное
Ввод/вывод	Асинхронный	Асинхронный	Асинхронный
Захист памяти	Сегментная	Страницчная	Страницчная
	Трехуровневая	Двухуровневая	Трехуровневая
Сборка при загрузке	Динамическая	Динамическая	Статическая
Подкачка	Задачная	Страницчная	Страницчная
Поиск жертвы	—	FIFO	FIFO
Файловая система	Без транзакций	Журнальная	Журнальная
Программный RAID	RAID 1	RAID 0 1 5	RAID 0 1
Длина имени файла	256	256	32+16
Версии файлов	Нет	Нет	Да
Форматы файлов	Потоковый	Потоковый	Блочный Относительный Индексно-последовательный
Командный процессор	cmd.exe	cmd.exe	DCL
Граф. подсистема	PM	Win32	X Window
ID пользователя	Вся система	Задача	Задача
БД учетных записей	Распределенная	Распределенная	Локальная
Сетевой протокол	NETBIOS/SMB	NETBIOS/SMB	DECNet

Наиболее важные заимствования из VMS — страницная подкачка и идентификация пользователя на уровне процессов — являлись ответом на насущные требования развития системы и могли быть заимствованы из любой ОС, адекватной времени. В остальном, табл. П1.1 показывает, что OS/2 1.x, безусловно, приходится Windows NT гораздо более близкой родней, чем VMS.

Наиболее важной заимствованной концепцией была журнальная файловая система NTFS, представляющая собой любопытный гибрид HPFS (основной ФС OS/2) и FCS2 (основной ФС VAX/VMS). Это заимствование следует признать довольно удачным.

Гораздо менее удачным было заимствование своеобразной стратегии управления рабочими множествами процессов в ОЗУ, используемой в VMS: разработчики Microsoft устранили из этой стратегии одно из ключевых понятий, квоту размера рабочего множества. В результате получилась система, практически не способная воспользоваться преимуществами страничной подкачки, потому что даже небольшая нехватка оперативной памяти приводит к резкому падению производительности из-за неспособности системы сбалансировать потребности приложений и дискового кэша.

Еще одна ключевая для понимания архитектуры Win32 концепция была позаимствована вовсе не из VMS и даже не из OS/2 1.x, а была, скорее всего, введена по настоятельным просьбам разработчиков графических приложений для Apple Macintosh. Речь идет о *системном реестре* (*system registry*), централизованной базе данных, в которой все модули системы, стандартные утилиты и прикладные программы хранят все, что считают нужным сохранить.

Системный реестр впервые был реализован в Mac OS. Ранние версии этой системы имели довольно простое ядро и небогатый набор системных настроек, поэтому реестр Mac OS в основном содержит настройки прикладных программ и в такой форме вполне терпим. Напротив, довольно сложная многопользовательская Windows NT, поддерживающая широкий спектр внешних устройств, нуждается в большом объеме конфигурационных данных для самой системы. Характер обращений к разным частям этих данных сильно различается — некоторые, например, нужны только при загрузке системы, а изменению подлежат только при изменении аппаратной конфигурации. Другие же меняются при каждом открытии нового окна пользовательской программой. Относительная ценность этих данных также отличается очень резко:искажение некоторых может привести к невозможности загрузить систему или к потере пользователями доступа к ней, некоторые другие можно было бы и не хранить вовсе. В свете этого, идея общей "свалки", в которой содержится все на свете, — начиная от слов, которые произносил пользователь, пытаясь убрать с экрана знаменитую скрепку, и заканчивая БД учетных записей — представляется автору не очень-то здравой мыслью.

В Windows NT этот концептуальный недостаток усугубляется недостатками реализации — реестр не имеет адекватных средств резервного копирования и восстановления (при фатальных повреждениях реестра Microsoft рекомендует переустановку системы) и фактически лишен средств самоконтроля и диагно-

стики. Во всяком случае, в версии 4.0 (автор не имел случая проверить это на более поздних версиях системы, но судя по тому, что исправление этой ошибки не анонсировалось, в 2000/XP ситуация не изменилась) ОС никогда не уменьшала объем реестра, даже после удаления большого количества ключей.

Еще одним важным новшеством была поддержка нескольких процессоров — кроме x86 первые версии Windows NT были реализованы для RISC-процессоров MIPS и DEC Alpha, и, существенно позднее, для PowerPC. Большинство RISC-процессоров не имеют многоуровневых режимов доступа, характерных для VAX и 80286/x86, поэтому разработчики Windows NT были вынуждены отказаться от привилегированных разделяемых библиотек (понятие, которое в той или иной форме присутствовало как в OS/2 1.x, так и в VAX/VMS) и перейти к двухуровневой системе привилегий.

Разработчики приложений не проявили интереса к альтернативным аппаратным архитектурам, поэтому NT на этих архитектурах не имела большого успеха, и в 1999 году без большого шума была прекращена поддержка Windows NT для последнего неинтеловского процессора, который к тому времени уже назывался Compaq Alpha [techupdate.zdnet.com].

Впрочем, в 2005 году Microsoft выпустил на рынок игровую приставку Xbox 360, основанную на процессоре PowerPC. Эта приставка работает под управлением специализированной версии Windows 2000.

В отличие от Unix, система не могла быть перенесена на процессоры с порядком байт, отличным от x86, из-за того, что в коде системы и флагманских приложений широко использовались неявные предположения о порядке байт (преобразование указателей на `short` в указатели на `long` и т. д.). К тому же, перенос на 64-разрядные процессоры оказался сопряжен с неожиданной сложностью: дело в том, что очень многие системные вызовы и документированные структуры данных Win32 используют описатели целочисленных значений, явно указывающие на количество байт в значении, такие как `WORD` (два байта) или `DWORD` (четыре байта). В результате, изменение размера некоторых скалярных значений на восьмибайтовые потребовало бы переделки спецификаций системных вызовов и переработки исходных текстов программ, использующих такие вызовы.

Для сравнения, в Unix везде, где это возможно, используются непрозрачные типы (`off_t` для длины файла и смещения в нем, `size_t` для размера структуры данных в памяти, `pid_t` для идентификатора пользователя), размер которых в байтах явно не указан. Так, на 32-разрядных архитектурах `size_t` имеет размер 32 бита, а на 64-разрядных — 64. Чтобы написать программу, не переносимую между 32- и 64-разрядными системами семейства Unix, программисту надо предпринимать специальные усилия, нарушающие все пра-

вила хорошего тона программирования. Поэтому системы семейства Unix давно перенесены на 64-разрядные микропроцессоры и используют их потенциал полностью.

Ко времени написания второго издания книги была известна распространяющаяся по закрытой подписке 64-разрядная бета-версия Windows XP для IA-32e и бета-версия Windows Vista.

С момента выхода версии 3.51 и до времени написания книги архитектура системы не подвергалась ни пересмотру, ни сколько-нибудь существенному развитию [Андреев/Беззубов 1999]. Наибольшее количество новшеств было введено в Windows 2000, когда в состав системы была включена служба каталогов Active Directory и ряд мелких улучшений. В частности, было исправлено множество мелких, но раздражающих недостатков стека TCP/IP версии 4.0, такие как невозможность сбросить кэш запросов DNS без перезагрузки системы или присвоить интерфейсу дополнительный адрес без сброса всех TCP-соединений на основном адресе того же интерфейса. В этой же версии системы была реализована загрузка драйверов устройств без перезагрузки ОС, весьма полезная для поддержки устройств с горячим подключением, прежде всего — устройств USB. В Windows 2000 и Windows 98 были унифицированы интерфейсы для ряда типов драйверов устройств (так называемая WDM — Windows Driver Model).

В 2002 году произошло разделение системы на две ветви. Ранее "серверная" и "клиентская" версии ОС (Windows NT/2000 Server и Windows NT/2000 Workstation) представляли собой два разных дистрибутива одной и той же системы, отличающиеся комплектацией (в Server были включены программные компоненты, необходимые для того, чтобы играть роль контроллера домена, и ряд сетевых серверных приложений и утилит) и настройками ядра. В 2002 же году была анонсирована чисто клиентская версия системы, Windows XP, а лишь в 2003 году вышла соответствующая ей серверная версия, Windows 2003.

В Windows XP были введены средства импорта реестра систем линии Windows 95/98/ME, сокеты типа `RAW` (до этого низкоуровневые утилиты TCP/IP, такие, как `ping` и `tracert`, были вынуждены работать непосредственно с драйвером сетевого интерфейса и самостоятельно формировать заголовки пакетов IP) и новая схема защиты от неавторизованного копирования [Ахметов 2001].

Как показал опыт эксплуатации XP, система защиты от копирования оказалась относительно малоэффективной и причиняющей больше вреда легитимным пользователям системы, нежели "пиратам". Патчи, позволяющие снять защиту с Windows XP Home Edition, появились буквально через несколько дней после выхода системы на рынок. Впрочем, эти патчи препятствовали

установке обновлений с Windows Software Update, но вскоре в открытом доступе появились и корпоративные версии системы Windows XP Professional, не требовавшие регистрации на сайте Microsoft, т. е. эффективно не защищенные.

Еще одну попытку Microsoft предприняла в 2006 году, начав распространение через канал Windows Software Update своеобразной программы, известной под названием Windows Genuine Software Advantage. Эта программа устанавливается вместе с остальными обновлениями Microsoft и сравнивает серийный номер системы с зашифрованным "черным списком", а также еще несколькими недокументированными способами проверяет систему на "нелегитимность". Из-за риска ложных сработок, а также по другим очевидным соображениям, Microsoft не рискнул встроить в эту программу действительно деструктивные действия. Все, что она реально делает, заподозрив нелегальную инсталляцию, — это выдает при входе в систему и во время работы раздражающие пользователя модальные окна. Часто возникающие в форумах подозрения о том, что Genuine Software Advantage передает какую-то информацию в Microsoft, не имеют документального подтверждения, во всяком случае для версий программы, распространявшихся до выхода второго издания книги.

Вообще, необходимо отметить, что эффективная защита от неавторизованного копирования — штука обоюдоострая. Как уже отмечалось в предыдущем разделе, на рынке программных продуктов очень сильны "сетевые эффекты", когда сам по себе факт широкой распространенности продукта значительно повышает его привлекательность. В этом смысле можно утверждать, что "пираты" во многих случаях объективно действуют в интересах поставщика программного обеспечения, расширяя пользовательскую базу и создавая "сетевой эффект". Понятно так же, что в некоторых случаях "пиратство" действительно снижает доход поставщика ПО. Вопрос о том, какой из эффектов сильнее в каждом конкретном случае, очень сложно убедительно разрешить, потому что чистого эксперимента поставить невозможно.

Пожалуй, самый близкий к чистому эксперименту исторический пример — это сравнение рыночной судьбы двух последовательных версий ОС для выделенных файловых серверов Novell Netware. Netware 3.x имела довольно простую систему защиты от несанкционированного копирования, которая обходилась столь же простым генератором серийных номеров. При этом Netware 3 была несомненным лидером рынка, настолько несомненным, что многие считали Novell столь же серьезным монополистом, как и Microsoft.

Версия Netware 4 была защищена гораздо более серьезной криптографической системой проверки номеров лицензий, для которой до сих пор неизвестно эффективных средств обхода. При этом же Netware 4 предоставляла значительные функциональные преимущества по сравнению с Netware 3, глав-

ным из которых была распределенная реплицируемая служба каталогов NDS (Novell Directory Service). Однако с выходом Netware 4 рост доли рынка Novell прекратился, а затем началось и сжатие пользовательской базы в абсолютном выражении, так что к 2000 году компания была близка к банкротству. Впрочем, даже этот пример нельзя признать чистым экспериментом, потому что приблизительно в это же время началось распространение серверов приложений и вытеснение файловых сервисов на вспомогательные роли. Нельзя не отметить, что Novell Netware всех версий была замечательным файловым сервером, но на роль сервера приложений она мало пригодна в силу архитектурных ограничений.

По-настоящему этапной версией Windows NT, по видимому, следует считать Windows XP Service Pack 2 и соответствующий ему Windows 2003 Service Pack 1. В этой версии под давлением здравого смысла и жалоб пользователей на бесконтрольное размножение разного рода malware была значительно модифицирована система безопасности.

На современных процессорах x86 (AMD Athlon/Pentium IV) XP SP 2 поддерживает бит защиты от исполнения в дескрипторах страниц виртуальной памяти, что существенно усложняет использование хакерами срывов буфера и многих других ошибок программирования, опасных с точки зрения удаленного исполнения кода.

Система переспрашивает пользователя о том, следует ли исполнять файлы, загружаемые с сетевых дисков. Также был значительно улучшен контроль со стороны пользователя над процессом загрузки из Интернета и последующего исполнения компонентов ActiveX, используемых Web-страницами.

Файрволл (маршрутизатор с фильтрацией пакетов), появившийся еще в Windows 2000, в Windows XP SP2 по умолчанию включен, а его настройки гораздо более доступны для пользователя. Это очень важно, потому что по данным проекта Honeynet срок жизни Windows XP SP1 (определенный как математическое ожидание интервала от момента подключения до момента заражения первым червем или ботом), подключенной к Интернету с реальным IP-адресом и настройками по умолчанию — т. е. с выключенным файрволлом — составляет около 20 секунд. Включение файрволла позволяет довести этот срок до двух часов.

Все эти возможности действительно существенно улучшили защиту системы от malware, но они же привели к нарушениям совместимости со многими из старых приложений. С другой стороны, главный недостаток системы безопасности Windows, состоящий в том, что многие приложения не работают или нуждаются в дополнительных настройках, если пользователь не является локальным администратором, в Windows XP SP2 устранен не был.

В следующей версии системы — Windows Vista — первоначально был обещан набор радикальных новшеств, наиболее важным из которых считалась WFS (Windows File System).

WFS — это "сверхоружие", которое куется в глубоких шахтах Редмонда приблизительно с начала 90-х годов. Главным его качеством считается конвергенция между функциями файловой системы и функциями СУБД. По-видимому, в основе концепции WFS лежит впечатление, которое на разработчиков Microsoft произвело знакомство с технологиями Sybase SQL Server — права на использование кода этого продукта Microsoft получил в районе 1989 года, когда совместно с Sybase и Ashton-Tate они начали портировать Sybase SQL 4.0 под OS/2.

Действительно, можно найти некоторый параллелизм между функциями реляционных СУБД и файловой системы. Та и другая системы обязаны отображать большие логически непрерывные объекты (реляционные таблицы и файлы) на фрагментированное запоминающее устройство. Для реляционной БД необходима, а для файловой системы полезна транзакционность исполнения некоторых операций. Для БД также необходимы, а для файловой системы могут быть полезны развитые средства поиска, в том числе по заранее созданным индексам. Поэтому идея конвергенции СУБД и ФС с ряда точек зрения представляется разумной.

С других точек зрения, впрочем, такая конвергенция представляется не столь хорошей идеей. Так, в реляционной СУБД все модификации объединены в транзакции. При работе же с файловой системой приложения не отмечают начало и конец транзакции, поэтому автоматически сделать работу этих приложений транзакционной невозможно. По-видимому, вкладывать значительные ресурсы в обеспечение транзакционности всех операций над ФС нецелесообразно.

Далее, объем индексов реляционной СУБД, которые нередко реализуются с применением хэш-таблиц с низкими коэффициентами заполнения, часто сопоставим с объемом индексируемых данных или даже превосходит его. Пользователи, скорее всего, не готовы к таким большим накладным расходам для файловой системы, поэтому технологии индексирования, применяемые в СУБД, для ФС неприменимы — но что тогда могла бы означать конвергенция?

Так или иначе, WFS несколько раз анонсировалась как технология, разрабатывающаяся для следующей версии Windows. В 2005 году было объявлено, что в Windows Vista она также не будет включена.

Из наиболее важных новшеств, которые, по-видимому, все-таки будут включены в новую версию, следует упомянуть очередную попытку улучшения системы безопасности, в первую очередь — набор мер, которые позволяли бы

работать не из-под локального администратора, сохраняя при этом многие из привычек пользователей Windows 98/XP и совместимость со старыми приложениями.

Другая группа важных новшеств, которую вряд ли следует признать столь же полезной, — это включение в систему средств DRM (Digital Rights Management — цифровое управление правами или, как это предлагает расшифровывать Р. Столлмэн, Digital Restrictions Management — цифровое управление ограничениями). Средства DRM, включаемые в Windows Vista, позволяют создавать файлы (например, электронные книги, видео- и аудиозаписи), которые нельзя будет копировать или даже допускающие лишь определенное количество просмотров. Это будет достигаться за счет шифрования этих файлов и включения в ядро системы криптосредств, работа в обход которых будет затруднена рядом вспомогательных мер. В частности, в 64-битной версии системы будет полностью запрещена установка неподписанных модулей ядра.

Подпись модулей ядра применяется в Windows XP и Windows 2000 и сейчас. Проверка подписи осуществляется в момент установки драйвера. Система принимает только подпись приватным ключом Microsoft. Условия, на которых разработчик драйвера может получить подпись, оговариваются индивидуальными контрактами поставщика оборудования и Microsoft и не разглашаются; по некоторым сведениям, цена за право подписать драйвер может составлять до \$10 за каждое проданное устройство. К счастью, современные версии системы всего лишь проверяют наличие подписи и предупреждают пользователя об его отсутствии. Поэтому драйвера многих устройств, изготовленных небольшими компаниями, да и обновления драйверов для устройств, изготовленных крупными фирмами, часто поставляются без подписи.

Единственной штатной возможностью подписать драйвер для Windows Vista является приобретение SDK от Microsoft. По доступным на время подготовки к печати второго издания книги данным, стоимость SDK составляет \$500, и он допускает подпись неограниченного количества драйверов. В условиях, когда ботнет из 1000 компьютеров продается на черном рынке за \$200—\$300, можно ожидать, что разработчики malware достаточно быстро получат возможность подписывать троянские модули ядра, так что остается лишь строить догадки о том, от кого или от чего реально должны защищать эти подписи.

Поскольку бинарный код системы и, соответственно, ключи шифрования будут доступны всем действительно заинтересованным, важным компонентом данной системы оказывается DMCA (Digital Millennium Copyright Act — акт об издательских правах цифрового тысячелетия). Это федеральный закон США, в соответствии с которым под угрозой уголовного наказания нельзя

модифицировать программные и программно-аппаратные комплексы, использующие криптографические средства защиты, от копирования кода и данных. К счастью, в соответствии с действующей редакцией закона, можно владеть модифицированными средствами такого рода, но нельзя продавать полученные программные и программно-аппаратные продукты. Аналогичные законы уже приняты в ряде развитых стран и осуществляется давление на другие страны, в том числе и на Россию, с требованием принять нечто подобное.

При весьма слабо изменяющемся ядре, развивался преимущественно пользовательский интерфейс. Версии 3.x вместо стандартного к тому времени рабочего стола использовали архаичный Program Manager, аналогичный использовавшемуся в Windows 3.x. В версии 4.0 система получила desktop (рабочий стол), аналогичный Windows 95. В XP все управляющие элементы графического интерфейса подверглись полному перепроектированию и были перекрашены в жуткий ядовито-голубой цвет. К счастью, основная поставка системы включает тему Windows Classic, которая позволяет вернуть внешний вид управляющих элементов к стилю Windows 2000.

В первые годы после ветвления проекта OS/2 на IBM OS/2 и Windows NT реакция пользователей свидетельствовала о том, что решение IBM было более адекватно их потребностям — оценки пользовательской базы IBM OS/2 и Windows NT сравнялись только в 1997—1998 годах. Потребовались выход Windows 95 и титанические усилия по перетягиванию пользователей на приложения для Win32 (так, при разработке Office 95 Microsoft пошла на беспрецедентный шаг и сделала пакет совместимым по формату файлов с предыдущей версией Office), чтобы к 1999—2000 годам Windows NT и ее новые версии, Windows 2000/XP, стали приемлемым решением для настольных персональных компьютеров. Потребовался также провал проекта OS/2 for PPC, чтобы устраниТЬ конкуренцию со стороны ближайшего родственника, претендовавшего (и не без успеха) на ту же рыночную нишу. Тем не менее до сих пор успехи системы на рынке ОС для настольных компьютеров следует охарактеризовать как скромные.

Успехи системы на серверном рынке были более внушительными. Главными недостатками Windows NT в качестве сервера являлись (и до сих пор являются) неэффективная стратегия управления памятью и беспрецедентное для промышленно эксплуатируемого программного продукта количество проблем с безопасностью. Я не имею достоверных сведений на этот счет, но складывается впечатление, что Microsoft тестирует свои продукты только на отсутствие ошибок, от которых программа прекращает работать сама, но не делает даже попыток искать ошибки, используя которые можно целенаправленно нарушить работу системы или получить к ней неавторизованный доступ.

Для сервера, исполняющего стабильную смесь приложений, первый недостаток не очень критичен — многие серверные приложения (особенно серверы СУБД) при старте занимают всю доступную память и практически не запрашивают ее в процессе работы. Благодаря этому система может стабилизировать свой динамический кэш и обеспечивать в стационарном режиме приемлемую производительность.

Второй недостаток является критически важным и одного его — в идеальном мире — было бы достаточно, чтобы Windows NT не использовалась в качестве сервера ни при каких обстоятельствах. Однако давление управленческого персонала во многих организациях было очень сильным, поэтому система получала распространение. Чаще всего ее использовали в локальных сетях, не подключенных к Интернету вообще или закрытых от глобальной сети брандмауэром, что несколько снижало требования к безопасности.

К тому же, пока NT была малоиспользуемой системой с бедным набором сетевых сервисов, мало кто всерьез интересовался ее взломом. Это привело к усилению давления со стороны управленцев — "вот видите, у соседей стоит — и ничего", поэтому серверы под управлением NT все чаще и чаще подключались к Интернету, иногда даже без закрытия каким-либо файрволлом (впрочем, файрволл в данном случае мало чем может помочь — сайт [www.microsoft.com] закрыт маршрутизатором с фильтрацией пакетов "по самые уши", и то его "роняют" несколько раз в год). Распространение системы привело к тому, что взломщики из спортивного интереса заинтересовались ею всерьез.

Первой ласточкой был выпущенный в 1997 году свободно распространяемый продукт Back Orifice (дословно — "задний проход"), демонстрировавший целый набор способов получения неавторизованного доступа (в том числе и с последующей установкой троянских программ) к системам Win32. Устанавливаемый в качестве троянской программы компонент пакета долгое время был лучшим из доступных инструментов удаленного управления Win32-системами, и я знаю немало системных администраторов, которые использовали его в своих сетях [www.sourceforge.net bo2k].

Впрочем, одна ласточка весны не делает, и еще три года пользователи Win32-систем жили относительно спокойно (если можно считать спокойной жизнью постоянную борьбу с макровирусами для MS Office, почтовыми вирусами и другими порождениями большой фантазии). За это время в систему добавились новые сетевые сервисы и расширилась номенклатура сервисов, запускаемых при установке ОС по умолчанию, — например, в их число попал флагманский серверный продукт, сервер FTP/HTTP и ряд других протоколов, IIS (Internet Information Server).

Собственно весна наступила в августе 2001 года с пандемией сетевого червя Code Red, который, как и червь Морриса, использовал несколько каналов распространения, в том числе срывы буфера в сетевых сервисах IIS. Как и червь Морриса, заразив одну из машин домена, Code Red распространялся на другие машины того же домена простым копированием по сети. Дальнейшее развитие событий, впрочем, резко отличалось от последствий атаки червя Морриса: Microsoft довольно быстро выпустила заплаты (patches), исправлявшие часть используемых вирусом ошибок, — однако полное количество ошибок, оставшихся в коде системы и сетевых сервисов, от этого почти не изменилось. Атаки червей и поливалентных (использующих несколько каналов распространения) вирусов, которые легко преодолевают корпоративные файрволлы, продолжались на протяжении всего 2001 года, демонстрируя все новые и новые проблемы в системе безопасности Windows NT/2000/XP.

В опубликованном в сентябре 2001 года докладе аналитическая компания Gartner Group рекомендовала ни при каких обстоятельствах не использовать IIS из-за огромного количества известных и весьма пессимистических прогнозов на количество неизвестных уязвимостей.

К моменту написания книги прогнозировать дальнейшее развитие событий не представлялось возможным. Очевидно, что ситуация с безопасностью Windows может только ухудшаться — или, точнее, абсолютная нетерпимость положения дел с безопасностью в Windows может становиться только более и более очевидна все большему и большему кругу людей. Попытки исправить положение законодательными мерами, например, применяя уголовные наказания к разработчикам вирусов или запрещая публикацию сведений о проблемах с безопасностью, вряд ли могут изменить тенденцию.

Так, наказание для разработчиков вирусов, хотя и морально оправданно, но вряд ли может быть эффективным, потому что в большинстве случаев создателя практически невозможно идентифицировать, а идентифицировав — весьма сложно доказать его вину по стандартам судопроизводства демократических стран.

В свою очередь, законодательный запрет публикации сведений об ошибках, разговоры о котором начались в 2001 году, не только абсолютно не оправдан морально, но и крайне вреден с прагматической точки зрения, хотя бы только потому, что сделает невозможным принятие контрмер администрациями уязвимых систем.

Оптимистический сценарий развития событий может состоять в том, что пользователи начнут массовым образом отказываться от применения Windows, или Microsoft пересмотрит свой подход к проектированию, разработке и тестированию программного обеспечения (или в данном случае включающее). Так или иначе, исправление положения потребует значительных вложе-

ний в перестройку всей вычислительной инфраструктуры и не может пройти безболезненно.

Впрочем, исторический опыт дает мне весьма мало оснований для оптимизма.

П1.4.7 Microsoft Xbox

— А вот если я суп сварил, тоже посадить могли?
— Нет, за суп не сажали.
— А если чаю заварил?
— Вроде нет... не помню. Эх, Лешка, много чего было... такого... — Иван поглаживал стакан, мучительно вспоминая чего-нибудь.

В. Шинкарев

К концу 90-х годов XX века многие аналитики, а иногда и руководители компаний (например, CEO компании Sun Microsystems Скотт МакНили) высказывали мнение, что близится смена поколений вычислительной техники. На практике это могло означать, что какая-то из новых аппаратных платформ убьет рынок персональных компьютеров примерно таким же образом, как в конце 80-х — начале 90-х настольные компьютеры и "подстольные" серверы на основе микропроцессоров уничтожили рынок мини-компьютеров. Среди кандидатов на роль нового лидера отрасли разными людьми назывались различные категории устройств, в том числе карманные компьютеры, "тонкие клиенты" и игровые приставки. Большинство прогнозов давало персональным компьютерам до десяти лет жизни, так что в настоящее время сложно сказать, насколько эти прогнозы реальны.

Трудно сказать, что именно заставило компанию Microsoft предпринять попытку диверсификации своего бизнеса — опасения, продиктованные такими прогнозами, или какие-то другие соображения. Так или иначе, в 2001 году Microsoft объявил о выходе на рынок игровых приставок с продуктом Xbox.

С аппаратной точки зрения, первая версия Xbox представляла аналог "игрового" PC-совместимого компьютера, исполненного в нестандартном конструктиве. Компьютер имел процессор x86, жесткий диск, привод DVD-ROM,

а также видеокарту с довольно мощным графическим ускорителем и видео выходом, который позволял использовать вместо монитора обычный бытовой телевизор. Машина была исполнена в компактном корпусе, который не допускал установки плат расширения; зато этот корпус хорошо смотрелся на тумбе рядом с другой аудио- и видеоаппаратурой. Вместо стандартных клавиатуры и мыши Xbox оснащался характерной консолью с несколькими кнопками (вверх/вниз, стрелять, бежать и т. д.).

Главным отличием Xbox от PC-совместимого компьютера является программное обеспечение. Установленный на материнской плате BIOS допускает загрузку и исполнение только ядра ОС, подписанного приватным ключом, принадлежащим компании Microsoft. Единственная публично доступная версия такой ОС — это специализированная (и сильно урезанная) версия Windows 2000. Помимо урезанности, эта версия системы отличается от стандартной Windows 2000 тем, что она соглашается загружать только программы, подписанные приватным ключом Microsoft.

На практике это означает, что Xbox, строго говоря, не является компьютером — на нем с помощью штатных средств нельзя исполнять произвольные программы, а можно загружать и выполнять только программы, предварительно одобренные Microsoft. В частности, оказываются запрещены любые модифицированные версии программ — например, программы со снятой защитой от несанкционированного копирования.

Эта система защиты сама по себе не очень надежна. Несложно модифицировать ядро ОС так, чтобы оно не проверяло подпись при загрузке пользовательских программ. Но тогда не сойдется подпись ядра, и BIOS откажется загружать такое ядро. То есть придется модифицировать и BIOS. С чисто программной точки зрения это тоже несложно; тот факт, что микросхема BIOS не допускает стирания, не является препятствием для пользователя, у которого есть программатор, чистые микросхемы ПЗУ и паяльник. Даже тот факт, что микросхема на плате залита эпоксидной смолой, сам по себе не способен остановить пользователя, у которого вдобавок к программатору и паяльнику есть острый нож, плоскогубцы и соответствующие моторные навыки. Впрочем, понятно, что каждый эшелон защиты позволяет отсеять немалый процент пользователей, которые хотели бы иметь в своем распоряжении незащищенную систему, но не готовы прикладывать для этого достаточно большие усилия.

Последним — и, наверное, самым важным — эшелоном защиты Xbox является федеральный закон США, известный под названием DMCA (Digital Millennium Copyright Act — закон об издательских правах цифрового тысячеления), в соответствии с которым продажа средств для снятия криптографической защиты является уголовным преступлением. Благодаря чему обладатели плоскогубцев, паяльников и соответствующих навыков не могут легально предложить свои услуги обладателям приставок Xbox.

В связи с этим, определенный теоретический и практический интерес представляет возможность обойти защиту без аппаратной модификации приставки. Это тоже возможно — необходимо лишь найти уязвимость в какой-то из системных или игровых программ, которая допускает исполнение произвольного кода. Все игры на Xbox исполняются с привилегиями, соответствующими привилегиям локального администратора в стандартной поставке

Windows 2000, поэтому, исполнив код от имени игровой программы, пользователь может модифицировать исполняющуюся копию ядра ОС и получить систему в полное свое распоряжение. В 2003 году таким образом была продемонстрирована загрузка Linux на Xbox. Впрочем, понятно, что для массового практического применения этот способ снятия защиты довольно-таки неудобен: необходимо запустить приставку в штатном режиме, запустить игру, в коде которой найдена ошибка, загрузить в ней файл сохранения, который инициирует срабатывание этой ошибки, и лишь потом будет загружена незащищенная ОС.

Сами приставки компания продает ниже себестоимости. Предполагается, что недостающие средства Microsoft собирает с производителей игр — ведь без подписи Microsoft ни одна игра на приставке работать не будет. На практике, стоимость инструментальных средств для разработки игр очень высока (фактически, она включает в себя цену доступа к ключам для подписи) и регулируется закрытыми соглашениями между разработчиком и Microsoft, так что на основании открытых данных сложно даже оценить порядок величины этой цены.

Подобная бизнес-модель имела определенный успех в исполнении других изготовителей игровых приставок — Sony и Nintendo. Впрочем, отрицательные стороны такой бизнес-модели очевидны. Наиболее важная проблема, которую следует отметить, — это тот факт, что необходимость заранее выложить значительную сумму за средства разработки полностью отсекает независимых разработчиков и мелкие компании. Рынок игр для приставки полностью контролируется несколькими крупными компаниями. Но и крупной компании надо обеспечивать возврат на инвестиции. На практике это означает, что компания-разработчик выходит на рынок только с играми, для которых гарантирован определенный уровень успеха — т. е., фактически, заранее отказывается от инновационной деятельности.

Так или иначе, Sony и Nintendo пока что могут получать прибыль в соответствии с данной бизнес-моделью, хотя и непонятно, сколько это продлится в долгосрочной перспективе. Microsoft пока что не смог достичь даже этого — судя по отчетам перед акционерами, приставочный бизнес до сих пор остается убыточным или, как это предпочитают называть представители компаний, инвестиционным направлением.

В конце 2005 года на рынок вышла новая модель приставки, Xbox 360. Главным ее отличием от первой модели является использование другого процессора — Power G5. Интересно, что по времени переход Xbox с архитектуры x86 на архитектуру Power довольно точно совпал с переходом Apple Macintosh (наиболее популярной платформы на основе процессоров Power) в обратном направлении. Тем не менее приставка Xbox 360 заслуживает упоминания хотя бы уже потому, что она поставляется с единственной на сего-

дня поддерживаемой версией Windows NT для процессора с архитектурой, отличной от x86.

П1.4.8. Windows 95/98/ME

Послушай, Зин, не трогай шурина,
Какой ни есть, а он родня.

В. Высоцкий

В первой половине 90-х годов XX века практически всем разработчикам и техническим специалистам было очевидно, что MS и DR DOS доживают последние дни: они не удовлетворяли запросам пользователей практически ни по одному из параметров: приложения требовали больших объемов памяти и перехода к 32-разрядной архитектуре, пользователям требовалась большая надежность, многозадачность, более развитые сетевые средства. Напротив, преимущества DOS, такие как небольшая потребность в памяти, становились все менее и менее критичными.

Основным препятствием на пути перехода пользователей на другие платформы было требование совместимости с существующими приложениями и драйверами нестандартных внешних устройств для DOS. Наилучшим образом удовлетворяла этому требованию IBM OS/2, в виртуальной машине которой можно было запустить не только практически любое приложение DOS, но и использовать многие модули ядра DOS, в том числе — загружая в разных виртуальных машинах разные версии ДОС и разные наборы драйверов. Однако высокие требования этой системы к ресурсам и ориентированная на корпоративных пользователей схема лицензирования приводили к тому, что система не получила большого распространения на массовом рынке.

В 1992—1993 годах Microsoft занялась разработкой системы, которая должна была заполнить перспективную рыночную нишу "многозадачной ДОС защищенного режима". Подобно марксизму, разрабатываемая ОС имела три источника и три составные части:

- Windows NT;
- DesqView и другие многозадачные среды для DOS;
- Windows 3.x.

От Windows NT новая система получила интерфейс системных вызовов — Win32 API — и формат загружаемого модуля PE (Portable Executable — переносимый исполняемый [модуль]).

У многозадачных сред разработчики новой ОС позаимствовали идею преобразования DOS в многозадачную среду защищенного режима: эти среды демонстрировали, что помещение ядра DOS в виртуальный 8086 и окружение

его семафорами позволяет относительно малой кровью получить как много-задачность, так и совместимость. Такая архитектура была довольно-таки трудоемка в реализации и создавала специфические проблемы (так, DOS не отдавала управления при обращениях к приводу гибких дисков, поэтому работа с дискетами из любой сессии приводила к остановке всех остальных сессий), но не представляла непреодолимых концептуальных сложностей и была в целом работоспособна.

Windows 3.x реализовала интерфейс между пользовательскими программами, работающими в защищенном режиме, и ядром DOS, исполняющимся в виртуальном 8086. К 1993–1994 годам на рынке существовало более десятка других продуктов, предоставляющих аналогичный интерфейс, так называемых *расширителей DOS* (DOS Extender), которые вкратце описывались в разд. П1.4.2.

С точки зрения разработчиков новой ОС Windows 3.x представляла наибольший интерес в качестве отправной точки, потому что, в отличие от остальных расширителей DOS, она предоставляла динамическую сборку в момент загрузки и реализовывала также событийно-ориентированную архитектуру, пусть и более примитивную, чем асинхронная очередь сообщений Win32.

К тому же, Windows 3.11 имела собственную дисковую подсистему, позволявшую работать с жестким диском в обход DOS (так называемый 32-битный доступ к диску).

Первым получившим признание результатом работ над новой системой был продукт Win32s — набор DLL и модулей ядра для Windows 3.x, позволявший выполнять загрузочные модули формата PE, использовавшие подмножество Win32 API. Win32s бесплатно поставлялся компанией Microsoft в качестве самостоятельного продукта и входил в поставку ряда продуктов третьих фирм.

После длинной последовательности публичных бета-версий, многократного переноса сроков и большой шумихи в прессе новая система, получившая название Windows 95, вышла на рынок в 1995 году. Система с самого начала задумывалась как переходная, предназначенная для облегчения перевода пользовательской базы DOS на Windows NT, однако прошло не менее 4–5 лет, прежде чем совместимость с приложениями DOS перестала быть решающим параметром при выборе ОС для настольного компьютера. За это время успело выйти несколько версий "переходной" системы (OSR2, 98, 98SE, Millennium Edition) и даже после выхода XP Microsoft еще не готова окончательно объявить о прекращении поддержки этой линии ОС.

П1.4.9. Windows CE

Система, предназначенная для кросс-разработки приложений, прошиваемых в ПЗУ, сверхпортативных компьютеров. К моменту написания книги это единственная система из семейства СР/М, поддерживающая процессоры, отличные от x86. Использование ПЗУ позволяет отказаться от целого набора подсистем, обслуживающих виртуальную память, загрузку исполняемых модулей и сборку в момент загрузки.

Система предоставляет графический пользовательский интерфейс с асинхронной очередью сообщений, вытесняющую многопоточность и базовый стек TCP/IP. В поставку системы входит среда кросс-разработки (компилятор, эмулятор целевого процессора, удаленный отладчик и интегрированная оболочка), работающая под Windows NT [Boling 2001].

Интерфейс системных вызовов этой ОС в целом похож на Win32 API — тем не менее, складывается впечатление, что основным источником требований было не обеспечение совместимости с приложениями для Win32 вообще, а пожелания разработчиков Mobile Office (пакет, включающий в себя функциональные аналоги некоторых программ из пакета Microsoft Office).

Любопытно, что, рекламируя эту систему, Microsoft делает большой упор на то, что она разработана с нуля, т. е. без использования существующего кода Win32-систем. На мой взгляд, это является косвенным признанием той репутации, которой качество кода этих систем заслуженно пользуется среди разработчиков и эксплуатационников.

ПРИЛОЖЕНИЕ 2

Архитектура и язык ассемблера x86

Процессоры с архитектурой x86 на время написания книги являются безусловными лидерами на рынках настольных и портативных компьютеров и маломощных серверов. На рынке доступны также мощные многопроцессорные серверы и массивно параллельные суперкомпьютеры на процессорах x86.

Родоначальником этого семейства является процессор Intel 80386, вышедший на рынок в 1987 году. В настоящее время процессоры с такой системой команд выпускаются компаниями Intel и AMD. В разные годы они выпускались также компаниями Cyrix, NextGen (впоследствии была поглощена AMD), VIA, IBM, UMC, Texas Instruments, Sis, Transmeta и некоторыми другими. Большинство из этих компаний делали лицензионные или сомнительные с точки зрения лицензионной чистоты аналоги или модификации микросхемы Intel 80486. Оригинальные микроархитектуры, предназначенные для реализации системы команд x86, разрабатывались компаниями Intel, AMD, NextGen (разработки NextGen затем легли в основу AMD K6 и последующих устройств компании AMD) и Transmeta.

Ряд источников приводит гораздо более обширный список компаний, производивших совместимые процессоры, но эти списки включают также компании, выпускавшие устройства с архитектурами 8086 и 80286.

Процессоры Intel Pentium IV на время выхода второго издания книги являются безусловными лидерами рынка по тактовой частоте (3.4 ГГц) и на большинстве задач демонстрируют неплохие показатели производительности по отношению к частоте, поэтому на ряде задач они являются абсолютными лидерами по производительности. Впрочем, это лидерство наиболее заметно на задачах, характерных для персональных компьютеров, т. е. преимущественно на однопоточных программах или задачах, когда программа состоит из одного основного потока и нескольких вспомогательных, которые большую часть времени заблокированы.

Процессоры Intel Pentium M (и его двухядерная версия Core Duo) и AMD Athlon/Opteron уступают P IV по тактовым частотам, но имеют существенно лучшие показатели производительности по отношению к частоте, поэтому на ряде задач лидерами по производительности оказываются эти устройства.

Применение современных реализаций x86 во встраиваемых приложениях ограничено из-за высокого энергопотребления и относительно плохих показателей времени переключения контекста процесса и задержки прерывания. Применение процессоров в мощных серверах также ограничено из-за того, что схемы конвейеризации, суперскалярного исполнения и управления кэш-памятью оптимизированы для исполнения однопоточных программ. Частые переключения контекста, характерные для многозадачных и многопоточных серверных приложений приводят к промыванию кэша, перегрузке каналов доступа к памяти и, во многих случаях, к значительным потерям производительности.

Однако тот факт, что процессоры x86 и сопутствующие им вспомогательные микросхемы (так называемый *chipset*) выпускаются большими сериями, приводит к общему снижению цены системы по сравнению с RISC-процессорами, поэтому серверы и встраиваемые платы на основе x86 оказываются привлекательным решением по критерию цена/стоимость.

П2.1. История

Несомненно, что своим доминирующим положением на рынке процессоры x86 обязаны тому факту, что компания IBM в 1982 году при разработке персонального компьютера IBM PC выбрала процессор Intel 8088. История доносит до нас очень мало достоверных сведений о том, по какой причине был сделан этот выбор.

Так, очень популярная легенда, встречающаяся в том числе и на сайте IBM [www.ibm-128.com тсри], гласит, что первоначально разработчики предполагали использовать микропроцессор Motorola MC68000. Но затем было достигнуто соглашение с Intel, в соответствии с которым IBM получала право изготавливать микросхемы 8088; в обмен на это Intel получал технологии изготовления стираемого ПЗУ на основе магнитных доменов (*bubble memory*). Память на магнитных доменах действительно вызывала большой интерес в первой половине 80-х годов, но коммерческого успеха эта технология не имела.

Более правдоподобная версия этой легенды утверждает, что все было гораздо проще: от использования процессоров Motorola отказались потому, что их изготовитель не мог обеспечить необходимые объемы поставок в приемлемые сроки.

Так или иначе, изготовители процессора 8088 не только оказались в нужном месте в нужное время, но и смогли использовать предоставленные возможности, что называется, "до дна".

Система команд 8086/8088 вызывала оторопь у всех программистов и разработчиков компиляторов, которые были знакомы с системами команд других процессоров, например того же MC68000 (то есть практически у всех, кто мог более или менее информирован оценить архитектуру). Процессор удивительным образом сочетал нехватку регистров (всего восемь штук) с принуждением к их использованию (архитектура регистр-память и неявное использование регистров во многих командах). Нарекания вызывала также асимметрия регистров — практически каждый регистр имел какую-то особую функцию, делавшую его незаменимым.

Много проблем также создавала сегментная память в том виде, в каком она была реализована в 8086/8088. Проблема усугублялась тем, что из четырех сегментных регистров три (CS, DS и SS) имели строго определенные функции, так что для произвольного использования оставался один — ES.

В процессоре следующего поколения, 80286, ни один из этих недостатков устранен не был. Тем не менее изменения в архитектуре процессора были значительными. Странная схема адресации "реального режима" была заменена на полноценную сегментацию с неперекрывающимися сегментами. Была добавлена сложная и по-своему элегантная система защиты сегментной памяти на основе колец защиты и шлюзов. Странно, что в дескрипторе сегмента были предусмотрены флаги присутствия, модификации и даже флаг доступа (clock-бит), но полноценного сегментного отказа не было. Исключение по доступу к "отсутствующему" сегменту работало как прерывание, сохраняемый счетчик команд указывал на следующую команду, а не на ту, которая вызвала исключение. Поэтому, изменив дескриптор сегмента, нельзя было перезапустить команду и сегментная подкачка в том виде, в каком она описана в главе 5, была невозможна.

В остальном, как уже говорилось, 80286 полностью сохранил систему команд 8086 вместе со всеми ее недостатками.

Главной собственной проблемой процессора оказался тот факт, что в "зашитченном режиме" было невозможно исполнять программы, рассчитанные на реальный режим: хотя все регистры и команды оставались точно такими же, структура памяти радикально отличалась. Поэтому, работая в защищенным режиме, невозможно было обеспечить совместимость с приложениями для MS/PC DOS — а именно эти приложения и обеспечивали основной спрос на процессоры, совместимые с 8086.

Операционные системы, работавшие в защищенном режиме 80286 — MS/SCO Xenix, OS/2 1.x, Netware 2.0, — имели лишь ограниченный успех на рынке.

Основная масса процессоров 80286 эксплуатировалась под управлением MS/PC/DR DOS в реальном режиме, т. е., по существу, использовалась как высокопроизводительные 8086. Трудно предоставить достоверную статистику, но, я думаю, не будет преувеличением сказать, что подавляющее большинство 80286 за все время своей эксплуатации никогда не переводились в защищенный режим.

Относительно успешная попытка устранить недостатки 8086 и 80286 была предпринята в процессоре 80386, который вышел на рынок в 1987 году.

В этом процессоре архитектура была полностью пересмотрена. Система команд 80386 (известная также под названиями x86 или IA-32) использует набор регистров, очень похожий на регистры 8086, но имеет совершенно другой, гораздо более симметричный, набор режимов адресации. Также была радикально изменена структура адресного пространства. Сегментация была вытеснена на вспомогательные роли. Процессор предоставляет линейное 32-битное адресное пространство с двухуровневой страничной трансляцией. Реализован полноценный механизм страничных отказов. Для тех, кто желал продолжать использовать сегментацию, были добавлены два сегментных регистра.

Сегментация как таковая была сохранена, но вместо трехуровневой трансляции (дескриптор сегмента содержит ссылку на страничный каталог, страничный каталог — на таблицу дескрипторов страниц), которая представляется мне логичной, была сделана довольно странная, на мой взгляд, совмещенная трансляция. Впрочем, это решение в некоторых отношениях может быть признано разумным.

Были продуманы механизмы обеспечения совместимости с 8086 и 80286. В зависимости от типа сегмента кода, процессор мог либо работать в "родном" режиме, интерпретируя команды x86, либо в режиме совместимости, интерпретируя команды 80286. Был реализован режим "виртуального 8086", обеспечивающий полную совместимость с программами для 8086, как по кодам команд, так и по структуре адресного пространства.

Главным недостатком архитектуры, который был унаследован от 8086/80286, следует признать недостаточное количество регистров общего назначения и сложную исторически сложившуюся систему команд. Современные реализации, такие как P6 (Intel Pentium Pro, Pentium II, Pentium III, Celeron), Intel Pentium IV, AMD Athlon, AMD Opteron компенсируют этот недостаток за счет виртуальных регистров и спекулятивного суперскалярного исполнения. Так, Pentium IV имеет 160 физических регистров и семь арифметико-логических устройств, в том числе два быстрых сумматора, работающих на удвоенной тактовой частоте. Интерпретируя программу, процессор просматривает код на несколько десятков команд вперед и динамически отображает

виртуальные регистры на физические, пытаясь избежать взаимозависимостей по данным между командами и обеспечить параллельное исполнение максимального количества операций.

При прерываниях и переключениях контекста суперскалярный процессор вынужден осуществлять так называемую сериализацию, т. е. приводить процесс вычислений в такое состояние, в котором он мог бы находиться, если бы команды исполнялись последовательно. Критерием успешной сериализации является то, что можно указать в коде такую команду, что все операции, закодированные до нее, уже выполнены, и результаты этого выполнения находятся в регистрах или памяти (или хотя бы в кэше), а ни одна операция, закодированная после нее, еще не выполнялась или, точнее, даже если такие команды и пытались выполняться, то их результаты и побочные эффекты отброшены.

Таким образом, прерывания приводят к разрывам конвейера и нарушениям процесса спекулятивного исполнения, т. е. к резкому снижению общей производительности.

В процессорах P III Xeon, P IV и AMD Opteron была сделана попытка компенсировать этот недостаток, введя так называемый *hyperthreading* (это слово еще не имеет общепринятого русскоязычного эквивалента, обычно используют кальку "гипертрединг"). Гипертрединговый процессор может исполнять одновременно несколько суперскалярных потоков команд x86. Прерывание приводит к разрыву конвейера и перезагрузке контекста только в одном из потоков исполнения, второй же поток продолжает исполняться в оптимальном режиме.

Принятие нового процессора рынком происходило долго и болезненно. Действительно, новая система команд и новая схема адресации обеспечивала огромные преимущества по сравнению с 8086/80286. Но для того, чтобы этими преимуществами воспользоваться, нужны были новые системы программирования и новые операционные системы — а для того, чтобы новая ОС завоевала рынок, необходимы были приложения для этой ОС. Получалась своего рода проблема курицы и яйца.

Первыми этот барьер преодолели поставщики Unix-систем. Еще в 80-е годы начались поставки BSD Unix и MS/SCO Xenix 386, а затем и портов Unix System V, сначала Release 3 — SCO Unix, Interactive Unix, позднее Release 4 — Esix, Microport, Solaris/x86, UnixWare.

В районе 1990 года произошло ветвление OS/2 на IBM OS/2 2.x и Microsoft Windows NT (эта история вкратце рассказывается в *приложении 1*). Обе возникшие ветви системы использовали все возможности нового процессора — страничную виртуальную память, страничную подкачуку, 32-разрядную плоскую адресацию и возможности эмуляции 8086 для запуска приложений

DOS. Однако из-за больших по тем временам требований к ОЗУ (16 Мбайт у Windows NT 3.51 и OS/2 2.x, 8 Мбайт у OS/2 3.0) и недостаточной поддержки со стороны производителей оборудования эти системы также не смогли завоевать рынок.

Приблизительно до 1993—94 года большинство машин на основе 80386 по прежнему работали под управлением DOS, т. е. использовались в качестве быстрого — еще более быстрого, чем 80286 — варианта 8086.

Первый серьезный прорыв 32-битных приложений защищенного режима на массовый рынок произошел в 1993—94 годах, после принятия стандарта DPMI (DOS Protected Mode Interface). Эти приложения могли работать не только под чистым DOS (в отличие от приложений, разработанных для DOS-расширителей первых поколений), но и в DOS-эмulyаторах OS/2 и Windows NT, поэтому пользователи таких приложений были более или менее уверены в безопасности своих вложений.

Решающим событием в завоевании рынка 32-битными ОС и приложениями был выход Windows 95, которая обеспечивала как совместимость с приложениями и драйверами устройств для DOS, так и возможность запускать 32-разрядные приложения DPMI и Win32. Таким образом, от момента выхода 80386 до времени, когда массовый пользователь смог воспользоваться всеми преимуществами новой архитектуры, прошло почти десятилетие.

В 2004 году компания AMD анонсировала процессоры Opteron, которые реализовали 64-разрядную архитектуру, сохраняя при этом совместимость с 32-разрядными приложениями x86. Кроме расширения разрядности, новые процессоры значительно расширили номенклатуру регистров общего назначения (теперь их стало 16). После некоторого сопротивления компания Intel реализовала поддержку новой системы команд в новых процессорах линии Pentium IV под названием IA-32e (название IA-64 было зарезервировано для процессоров Itanium). По существу, это решение означает смертный приговор линии процессоров Itanium, но на время подготовки книги к печати Intel еще не анонсировал прекращения поставок этих устройств.

Поставщики ОС довольно быстро поддержали новый процессор — так, доступна бета-версия 64-разрядной Windows XP, а дистрибуторы Linux уже давно коммерчески поддерживают его. Однако новые версии ОС не поддерживают драйверы и другие модули ядра для 32-разрядных аналогов; также не допускается сборка "смешанных" программ, т. е. таких, что часть программы использует 64-разрядную адресацию и систему команд, а другая часть — 32-разрядную. Иными словами, 64-разрядные приложения не могут использовать 32-разрядные DLL. Поэтому есть причины опасаться, что путь новой архитектуры на рынок также окажется длительным и болезненным. Хотелось бы, чтобы он не оказался таким же длительным, как у x86.

Процессоры x86 используются главным образом в персональных компьютерах и маломощных серверах. Особенно это справедливо для процессоров Intel Pentium IV.

Большинство вычислительных систем на основе x86 (настольные и портативные персональные компьютеры, серверы и даже большинство плат для встраиваемых приложений) — это так называемые PC-совместимые компьютеры, обеспечивающие имитацию аппаратуры оригинальных компьютеров IBM PC. Требования PC-совместимости определяют:

- интерфейс ПЗУ первичного загрузчика (BIOS — Basic Input/Output System — базовая система ввода/вывода), который обеспечивает доступ к гибким и жестким дискам, чтение с клавиатуры, примитивные средства вывода на графический адаптер, переключение графических режимов и ряд других сервисных функций. В современных компьютерах BIOS выполняет также определение и конфигурацию установленных в компьютере устройств PCI (а в не столь современных также и ISA P'n'P), управление питанием и некоторые другие задачи, но ПЗУ современных плат все равно сохраняют совместимость по API с оригинальным IBM PC BIOS;
- определенную конфигурацию внешнего контроллера прерываний 8529;
- определенную номенклатуру внешних устройств — программируемые таймеры, контроллер ПДП, контроллеры клавиатуры и видеоадаптера и ряд необязательных устройств: до четырех контроллеров последовательных портов RS232, один или два контроллера параллельных портов Centronics;
- в IBM PC/AT был добавлен второй 8529, а к номенклатуре внешних устройств были добавлены часы реального времени, работающие от собственной батареи и не теряющие время даже при выключениях компьютера, и своеобразная группа устройств, известная под не совсем корректным называнием gate A20 controller. В IBM PS/2 был добавлен порт PS/2, предназначенный для подключения мышей. К середине 90-х годов порт PS/2 и периферийная шина PCI стали стандартом де-факто, а к концу 90-х была прекращена поддержка шины ISA.

Для объяснения функций gate A20 необходимо глубокое понимание работы подсистем виртуальной памяти 80286 и x86, а также способов формирования адреса во всех процессорах линии, начиная от 8086. В разд. П2.2.3 я попытался вкратце описать назначение этого устройства, но этого описания, разумеется, недостаточно для самостоятельного программирования. В действительности, мало кто — даже среди опытных разработчиков ОС для PC-совместимых компьютеров — понимает, что именно делает это устройство и зачем.

Разумеется, за четверть века, прошедшую со времени выпуска оригинальной IBM PC, спецификации ПЗУ и аппаратуры неоднократно менялись. Так, оригинальный BIOS не мог работать с дисками, имеющими более 1024 цилиндров и/или более 64 секторов на дорожке. Таким образом, одна поверхность диска не могла содержать более 64К секторов. В начале 90-х это ограничение было обойдено за счет введения так называемого LBA (Logical Block Addressing). В действительности, LBA предполагает отказ от адресации блоков в терминах поверхности, цилиндра и сектора и переход к линейной логической нумерации блоков. Но с точки зрения BIOS, диски LBA выглядят как диски с 1024 цилиндрами, 64 секторами на дорожку и большим количеством поверхностей, гораздо больше, чем на физическом диске. Номер поверхности у BIOS кодируется байтом, так что количество логических "поверхностей" могло достигать 256!

Впрочем, когда объемы дисков превысили 8 Гбайт, эта схема адресации также дошла до своего предела, так что в конце 90-х годов многие системные администраторы вынуждены были при установке ОС учитывать ограничения BIOS и размещать загрузочный раздел в пределах первых 1024 логических цилиндров. Впоследствии появились расширенные функции BIOS, способные адресовать диски большего объема, но старые загрузчики, не умеющие пользоваться этими функциями, оказываются по-прежнему ограничены первыми 8 Гбайт диска.

В интерфейсы BIOS и оборудования вносились и другие изменения, полный список которых слишком обширен, чтобы приводить его в этом приложении, даже если посвящать каждой группе новых функций одно предложение. Большинство этих изменений оставили следы в интерфейсах BIOS, так что полное описание API современного BIOS по объему сопоставимо с описанием программного интерфейса сложной многозадачной ОС, но при этом гораздо менее логично и умопостижимо.

Некоторые ОС, например MS DOS и Windows 95/98/ME, имеют жестко закодированную в ядро логику работы с контроллерами прерываний и стандартной периферией и/или сервисами BIOS, и поэтому могут загружаться и работать только на PC-совместимых компьютерах. Другие ОС, например Windows NT и системы семейства Unix, могут работать на не-PC-совместимых машинах, но это требует замены вторичного загрузчика и определенных модулей ядра. Конкретно у Windows NT эти модули объединены под названием HAL (Hardware Abstraction Layer — слой абстракции оборудования).

Среди не-PC-совместимых машин на основе x86 следует упомянуть:

- рабочие станции Sun Roadrunner, выпускавшиеся в конце 80-х — начале 90-х годов. Они работали под управлением специальной версии Sun OS, в настоящее время их поддерживают некоторые дистрибутивы Linux;

- массивно многопроцессорные комплексы Sequent с архитектурой NUMA, использовавшиеся как для вычислительных задач, так и в качестве серверов баз данных. В 1999 году компания Sequent была приобретена IBM; машины, основанные на технологиях Sequent, в настоящее время выпускаются под названием IBM NUMA-Q. Для Sequent была разработана специализированная версия SCO Unix, сейчас для IBM NUMA-Q доступен HAL для Windows 2000, также поддерживаются SCO Unix, SCO UnixWare и Linux;
- игровую приставку Microsoft Xbox. Эти машины оснащены криптографическими микросхемами DRM, так что их BIOS отказывается загружать и исполнять вторичные загрузчики ОС, не имеющие цифровой подписи Microsoft. Эти устройства рассчитаны только на работу со специализированной (и сильно урезанной) версией Windows 2000. Специализация включает в себя не только замену HAL, но и добавление средств DRM, так что система отказывается загружать неподписанный код. Это вынуждает разработчиков игровых программ для XBox платить отчисления Microsoft, а также не позволяет модифицировать код программ и удалять из них средства защиты от несанкционированного копирования. После замены BIOS на Xbox возможна загрузка нестандартных ОС, в том числе Linux, но продажа модифицированных BIOS в США преследуется в уголовном порядке на основе закона DMCA (Digital Millennium Copyright Act — закон об издательских правах цифрового тысячелетия);
- персональные компьютеры и ноутбуки Apple iMac, вышедшие на рынок в 2005 году. Отличительной особенностью этих машин является использование загрузочного ПЗУ EFI (Extensible Firmware Interface), спецификации которого разрабатывались специалистами Intel с конца 90-х годов. EFI несовместим с PC BIOS во многих отношениях, начиная, хотя бы, с того, что PC BIOS работает в реальном режиме процессора, так что ядро и/или вторичный загрузчик ОС должны сами переводить процессор в защищенный режим, а EFI работает в защищенном режиме с самого начала. Сейчас основная ОС для этих машин — специализированная версия MacOS X. Продемонстрирована возможность загрузки Linux с модифицированным загрузчиком и соответствующим набором драйверов но ко времени выхода второго издания этой книги ни один крупный дистрибутор Linux еще не объявил о поддержке EFI. Технически возможна разработка специализированного HAL для загрузки Windows XP и/или Vista на этих машинах, но Microsoft пока что не опубликовал никаких однозначных утверждений по поводу того, будет ли он поддерживать EFI, и если да, то в какие сроки и в какой форме. Весной 2006 года Apple опубликовала неподдерживаемый вторичный загрузчик BootCamp, который обеспечивает загрузку Windows XP на Intel Mac и сосуществование Windows и MacOS на жестком диске.

Intel неоднократно предпринимала попытки избавиться от архитектуры x86 путем создания несовместимого с ней, но обладающего решающими преимуществами конкурента. Среди этих попыток следует упомянуть:

- экстравагантную CISC-архитектуру Intel 432 (iAPX 432) с сегментной виртуальной памятью, реализующей взаимно недоверяющие подсистемы;
- RISC-процессор i960;
- конвейерный RISC-процессор i860. Конвейерность выражалась в том, что компилятор и ОС знали, что команды не всегда исполняются в той последовательности, в которой закодированы, и могли в определенных пределах управлять их перестановкой;
- другой весьма экстравагантный конвейерный 64-разрядный RISC-процессор Itanium, разрабатывавшийся совместно с Hewlett Packard, частично под влиянием архитектур DEC Alpha и HP PA-RISC.

Ни одна из этих попыток не увенчалась успехом. iAPX 432 вообще не был доведен до серийного производства — неясно только, по чисто технологическим причинам или также и из-за осознания бесперспективности проекта.

i960 пошел в серию без диспетчера памяти и имел определенный успех во встраиваемых приложениях и в качестве периферийного контроллера в мощных серверах.

Для i860 не удалось обеспечить приемлемой скорости переключения контекста и даже приблизиться к теоретическим оценкам производительности при исполнении однопоточных программ. Единственным реальным достоинством оказалась хорошая для того времени производительность при вычислениях с плавающей точкой. Процессор выпускался небольшими сериями и применялся в массивно-параллельных суперкомпьютерах под управлением версии Unix, использовавшихся для научных и инженерных расчетов.

Itanium обеспечивал неплохие, сравнимые с DEC/Compaq Alpha, показатели производительности, но из-за несовместимости с x86 оказался вытеснен в небольшую по объему рыночную нишу. Системы на основе Itanium поставлялись малыми сериями в качестве серверов среднего уровня под управлением Unix-системы Thru64 и OpenVMS, но не оказали значительного влияния на рынок. На время написания второго издания книги Intel еще не объявил о прекращении поставок процессоров этого семейства, но очевидно, что такое прекращение — лишь вопрос времени, и, скорее всего, небольшого.

П2.2. Архитектура процессора

В этом разделе будут описаны лишь основные черты системы команд x86, необходимые для понимания приводимых в книге примеров кода и других случаев, где я ссылался на архитектуру x86. Полное описание x86, необходи-

мое для самостоятельного программирования и даже для разработки собственной ОС, может быть найдено в документации на сайте компании Intel [www.intel.com Pentium4] (этот источник полезен также и с той точки зрения, что в нем детально описана архитектура 64-битного расширения IA-32e), а также во многих книгах, например [Паппас/Марри 1993, Пирогов 2005]. Доступно много другой литературы, как в виде книг, так и в электронной форме. Поэтому далее я не буду описывать ряд особенностей архитектуры x86. В частности, практически не будут описываться служебные и отладочные регистры процессора; также не будет рассматриваться появившийся в микрархитектуре P6 режим гипервизора, используемый системами виртуальных машин, такими как VMWare.

П2.2.1. Регистры

Процессор имеет следующие группы регистров:

- восемь регистров общего назначения;
- шесть сегментных регистров;
- восемь регистров математического сопроцессора (в процессорах, поддерживающих расширение системы команд SSE, количество этих регистров расширено до 16, но дополнительные регистры доступны только командам SSE);
- счетчик команд и слово состояния процессора;
- управляющие регистры, в том числе управляющие регистры диспетчера памяти;
- служебные, машинно-зависимые и отладочные регистры.

Управляющие и машинно-зависимые регистры в этой книге будут рассматриваться лишь вкратце. Служебные и отладочные регистры не существенны для понимания примеров ассемблерного кода, приводимых в этой книге, и поэтому вообще не будут рассматриваться.

П2.2.1.1. Регистры общего назначения

Процессоры x86 обеспечивают бинарную совместимость с процессорами 8086 и 80286 и совместимость по языку ассемблера с процессорами 8080, 8085, Zilog Z80, а также с родоначальниками микропроцессорного бизнеса компании Intel, кристаллами Intel 8008 и 4004. Номенклатура регистров, таким образом, до сих пор сохраняет память о четырехразрядном 4004, первом в мире полностью программируемом микропроцессоре, который был анонсирован в 1969 году. Следы этой памяти до сих пор можно найти и в некоторых других архаичных чертаках системы команд.

Процессор имеет восемь 32-разрядных регистров общего назначения, обозначаемых EAX (A — accumulator), EBX (B — Base), ECX (C — Counter), EDX (D — Data), ESP (SP — Stack Pointer), EBP (BP — Base Pointer), ESI (SI — Source Index), EDI (DI — Destination Index). Буква Е используется, чтобы отличить эти регистры от одноименных 16-разрядных регистров 8086/80286. В режиме совместимости младшие 16 бит каждого из перечисленных регистров доступны в качестве соответствующего регистра 80286 (ах, вх и т. д.) (см. рис. 2.2).

Младшие 16 бит регистров EAX, EBX, ECX и EDX также доступны в качестве регистровых пар AH/AL, BH/BL, CH/CL и DH/DL. Восемь байтовых регистров, образующих эти пары, соответствуют регистрам процессоров 8080/8085 и 8008.

У IA-32e, как уже говорилось, регистры расширены до 64 бит, а их количество увеличено до 16. Первые восемь из этих регистров называются RAX, RBX и т. д.; их младшие 32 бита доступны как EAX, EBX и т. д. Дополнительные восемь регистров обозначаются буквой R с номером — от R8 до R15.

П2.2.1.2. Сегментные регистры

x86 имеет шесть сегментных регистров:

- CS (Code Segment — сегмент кода);
- DS (Data Segment — сегмент данных);
- ES (Extended Segment — сегмент расширения);
- SS (Stack Segment — сегмент стека);
- GS и FS (названия не имеют осмыслинной расшифровки).

Программисту эти регистры доступны в виде шестнадцатиразрядных регистров, в которые загружается селектор сегмента. С каждым из этих регистров также связан "теневой" (shadow) регистр; при загрузке любого шестнадцатибитного значения в основной регистр процессор интерпретирует его как селектор сегмента и автоматически пытается загрузить дескриптор соответствующего сегмента в теневой регистр. Если такого сегмента не существует или доступ к нему запрещен, происходит ошибка сегментации. Так или иначе, загрузка сегментного регистра приводит к дорогостоящей попытке изменить состояние диспетчера памяти, потому эти регистры не следует использовать для хранения произвольных 16-битных значений.

Регистры CS, DS, ES и SS соответствуют одноименным регистрам 8086/8088/80286.

П2.2.1.3. Регистры математического сопроцессора

x86 имеет восемь 80-разрядных арифметических регистров, которые могут использоваться для хранения 80-разрядных чисел с плавающей точкой (пол-

ная точность, full precision), 64-разрядных чисел с плавающей точкой (двойная точность, double precision), 32-разрядных чисел с плавающей точкой (одинарная точность, single precision), двоично-десятичных чисел с плавающей точкой и, наконец, целых чисел.

Каждый арифметический регистр имеет трехбитовый тег, указывающий, значение какого типа сейчас лежит в этом регистре. При операциях над значениями разного типа происходит преобразование точности и, если это необходимо, представления данных. Перенос чисел разных форматов из ОЗУ в арифметический регистр осуществляется разными командами, которые устанавливают разные значения тега.

Арифметические регистры не имеют собственных имен и называются по номерам, от ST0 до ST7. При этом нумерация регистров не фиксированная — они объединены в кольцевой стек. Регистр ST0 соответствует вершине стека, а регистр FP7 — его дну. При проталкивании в стек и выталкивании из него нумерация всех регистров сдвигается. Кольцевая природа стека выражается в том, что в стек можно протолкнуть более восьми значений, но при этом будут потеряны ранее протолкнутые значения. Аналогично, может быть вытолкнуто более восьми значений, но при этом при последних выталкиваниях будет получено не то, что следовало бы ожидать при бесконечном стеке. Слежение за глубиной стека возлагается на программиста или на среду исполнения языка высокого уровня.

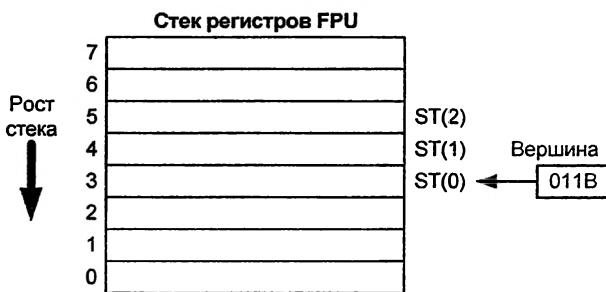


Рис. П2.1. Регистровый стек арифметического сопроцессора

В старых реализациях x86 эти регистры не были обязательной частью архитектуры, поэтому их часто называют регистрами сопроцессора. В 80386 эти регистры и логика, реализовавшая команды работы с плавающей точкой, разместились на отдельной микросхеме, обозначавшейся 80387, которая так и называлась — сопроцессор. Дешевые вычислительные системы, как правило, поставлялись без сопроцессора; в лучшем случае такие системы имели гнездо для его последующей установки.

Процессор 80486 имел сопроцессор на основном кристалле, но поставлялся на рынок в двух модификациях — 80486DX, в котором сопроцессор работал, и 80486SX, в котором сопроцессор физически присутствовал, но был логически выключен при распайке кристалла в корпус. Как правило, в качестве 486SX поставлялись кристаллы с бракованным сопроцессором но работоспособным основным процессором. В спецификациях упоминался кристалл 80487, который якобы мог использоваться для снабжения машин с 486SX аппаратным сопроцессором. Предположительно, 80487 должен был делаться на основе кристаллов 80486 с дефектным основным процессором, но работоспособным сопроцессором.

Мне доводилось видеть много материнских плат с нераспаянной площадкой под такой сопроцессор или даже с гнездом для него (количество и размещение контактов в гнезде в точности соответствует корпусу 80486). Я лично ни разу не видел работающего сопроцессора в таких платах и ни разу не видел упоминания 80487 в каталогах поставлявшихся микросхем, поэтому я не уверен, что такая микросхема вообще существовала в природе.

При исполнении команд с плавающей точкой на машинах без сопроцессора генерировалось исключение по недопустимому коду команды. Обработчик этого исключения может содержать эмулятор сопроцессора; ядра всех популярных ОС и среды исполнения большинства систем программирования для DOS содержат или, во всяком случае, допускают включение и выключение такого эмулятора в зависимости от конфигурации.

Практика реализации математического сопроцессора на отдельной микросхеме широко применялась в мини- и микрокомпьютерах в 60-е, 70-е и первой половине 80-х годов XX века; разумеется, в миникомпьютерах речь шла не об отдельной микросхеме, а об отдельной плате. Именно так была реализована плавающая точка в популярных миникомпьютерах PDP-11 компании DEC. Так же, как и в x86, в PDP-11 при отсутствии физического сопроцессора допускалась его программная эмуляция путем перехвата прерывания по недопустимой команде.

Точно так же широко встречается практика выпуска частично работоспособных микросхем с другой маркировкой и, обычно, по более низкой цене. Действительно, при изготовлении микросхем выход годных кристаллов с одной пластины низок и для микросхем с высокой степенью интеграции он измеряется десятками процентов и даже одиночными процентами. Порогом пригодности к серийному производству считается выход годных кристаллов 3%. Разумеется, если частично работоспособное устройство может найти себе какое-то применение, изготовитель микросхем не упускает такой возможности.

Так, производители микроконтроллеров выпускают на рынок широкий спектр микросхем с разными наборами интегрированных периферийных уст-

ройств (портов ввода/вывода, ЦАП/АЦП, адаптеров USB, I²O, контроллеров жидкокристаллических дисплеев и т. д.). Как правило, кристалл такого микроконтроллера содержит все устройства, соответствующие наиболее мощной из моделей соответствующего ряда, но часть из этих устройств неработоспособна и отключена от шин питания и логики центрального процессора.

В процессорах Pentium и последующих устройствах компания Intel сделала сопроцессор обязательной частью архитектуры процессора. Кроме того, скорость работы сопроцессора была резко повышена.

Процессоры, поддерживающие расширение системы команд MMX (Pentium MMX, P6, AMD Athlon), могут использовать регистры сопроцессора в качестве регистровых массивов для хранения групп целых чисел и чисел с плавающей точкой одинарной точности. Команды MMX обеспечивают выполнение над этими числами групповых и векторных операций.

У процессоров, поддерживающих расширение системы команд SSE (P6, AMD Opteron), количество регистров сопроцессора увеличено до 16. Команды SSE предоставляют операции SIMD (Single Instruction, Many Data — одна команда, много данных), позволяющие совершать сложные операции между вектором значений в регистрах SSE и длинными векторами, размещенными в ОЗУ — матричное умножение, свертку и т. д.

Расширения MMX и SSE не будут рассматриваться в этой книге; команды работы с плавающей точкой рассмотрим лишь вкратце.

П2.2.1.4. Счетчик команд и слово состояния процессора

Процессор имеет 32-разрядный счетчик команд EIP, который не входит в список регистров общего назначения. В системе команд не предусмотрено средств для прямой модификации этого регистра; этот регистр неявно сохраняется в памяти при исполнении команд вызова и переключения задачи, а также при прерываниях, исключениях и эмулируемых программных прерываниях. Кроме того, он модифицируется при исполнении команд вызова, условного и безусловного перехода, возврата из вызова и прерывания, а также команды переключения задачи.

Полный адрес команды образуется из 16-разрядного селектора сегмента, хранящегося в регистре CS, и значения регистра EIP.

Слово состояния процессора (регистр EFLAGS) содержит шесть арифметических флагов:

- z — нулевой результат (равен 0, если результат последней арифметической или логической операции содержит хотя бы один ненулевой бит, и 1 — в противном случае);

- s — знак результата (соответствует старшему биту результата последней арифметической и логической операции);
- c — перенос, равен переносу в бит 33 при арифметических операциях и сдвигах;
- o — переполнение. Равен 1, если при арифметической операции был перенос в 32 бит, но не было переноса в 33 бит, и 0 — в противоположном случае;
- a — межтетрадный перенос, соответствует переносу из 4 в 5 бит при сложениях и вычитаниях. Используется для коррекции при операциях с упакованными двоично-десятичными числами (эти операции не будут рассматриваться в этой книге);
- p — четность. Равен 1, если количество единичных бит в результате последней операции четное, и 0 — в противоположном случае. Или, что то же самое, этот бит равен результату операции XOR или сложения по модулю 2 всех бит результата. Данный бит используется, главным образом, при реализации телекоммуникационных протоколов, использующих бит четности в качестве контрольной суммы. Контроль четности ОЗУ реализуется отдельно, внешними по отношению к процессору средствами.

Кроме этого, слово состояния содержит ряд флагов, управляющих работой процессора, среди которых следует упомянуть:

- двухбитовое поле CPL (Current Privilege Level — текущий уровень привилегий);
- двухбитовое поле IOPL (Input/Output Prvilege Level — уровень привилегий ввода/вывода), которое используется для проверки, имеет ли текущая задача достаточный уровень привилегий для исполнения операций ввода/вывода;
- флаг запрета прерывания;
- флаг направления для строковых команд.

Некоторые из этих бит, например флаг направления, могут произвольно устанавливаться пользовательской программой. Другие биты, например флаг запрета, прерывания или IOPL, могут устанавливаться только программой с достаточно высоким уровнем привилегий. Слово состояния сохраняется как целое при внешних и программных прерываниях и исключениях и при переключении задачи; восстанавливается как целое при возврате из прерывания и при переключении задачи. Арифметические флаги могут копироваться как целое в байтовый регистр AL и восстанавливаться из него специальными командами. Для манипуляции отдельными битами существуют специальные команды, например команды CLI (разрешить прерывания), STI (запретить

прерывания), CLC (очистить флаг переноса), STC (установить флаг переноса) и др.

Существует также отдельный регистр состояния сопроцессора, который, в свою очередь, состоит из трехбитового указателя регистрового стека, арифметических флагов сопроцессора и битов управления сопроцессором, которые управляют преобразованием точности. Регистр тегов сопроцессора также доступен в виде отдельного регистра; доступ к нему необходим при сохранении и восстановлении контекста процесса.

Большинство команд с плавающей точкой устанавливают флаги условий в слове состояния сопроцессора вместо EFLAGS; для использования этих флагов в командах условного перехода необходимо скопировать флаги сопроцессора в AH и проверить соответствующие биты AH логическими битовыми операциями. Это наследие тех времен, когда сопроцессор был отдельным устройством. В P6 (Pentium Pro/II/III) появились дополнительные команды сравнения, непосредственно устанавливающие арифметические признаки в EFLAGS.

П2.2.1.5. Регистры управления диспетчером памяти

Два основных регистра сегментного диспетчера памяти — это регистры GDTR (Global Descriptor Table Register — регистр глобальной таблицы дескрипторов) и LDTR (Local Descriptor Table Register — регистр локальной таблицы дескрипторов).

48-битный регистр GDTR содержит 32-битный физический адрес таблицы дескрипторов сегментов GDT (Global Descriptor Table) и 16-битный ограничитель ее длины. В действительности, размер GDT ограничен 32К дескрипторов (это объясняется структурой селектора сегмента).

LDTR по структуре аналогичен сегментным регистрам — он состоит из доступного программисту 16-битового поля селектора сегмента и теневых полей. При загрузке селектора сегмента в LDTR, в теневые поля автоматически загружается дескриптор соответствующего сегмента.

Еще один важный регистр — TR (Task Register — регистр задачи), который содержит селектор сегмента текущей задачи. Задача описывается специальным сегментом TSS (Task Selector Segment — селектор сегмента задачи). Для каждой задачи должны быть определены селекторы кода и данных, четыре селектора сегментов стека для каждого из уровней привилегий, и по одному указателю стека для каждого из этих сегментов стека.

К управлению сегментным диспетчером памяти также следует отнести регистр IDTR (Interrupt Descriptor Table Register — регистр таблицы дескрипторов прерываний), которые устроены аналогично LDTR: в них загружаются 16-битные селекторы сегмента, но при этих загрузках также происходит загрузка дескриптора сегмента в теневые поля регистра.

Страницочный диспетчер памяти управляет регистром CR3 (Control Register 3 — управляющий регистр 3), который содержит указатель на таблицу верхнего уровня страницной трансляции (так называемый каталог страниц). Каталог страниц должен быть выровнен на страницу, при этом младшие биты регистра CR3 используются для хранения флагов. Кроме этого, диспетчер памяти управляет флагами в регистре CR4. Необходимо также упомянуть регистр CR2; при страниценных отказах и других исключениях, возникающих при страницной трансляции, в этом регистре хранится адрес, вызвавший исключение.

Важной частью диспетчера памяти является TLB (Translation Lookaside Buffer, справочный буфер трансляции), или кэш дескрипторов страниц и страницных каталогов. Этот кэш не доступен для прямой модификации; загрузка дескрипторов страниц в TLB происходит неявно, при первом обращении к странице. Для модификации записи в TLB используется команда INVLPG, операндом которой является виртуальный адрес. Если дескриптор соответствующей этому адресу страницы находится в TLB, то он помечается как некорректный (устаревший) и при следующем обращении к странице дескриптор будет перезагружен из памяти. Обычно INVPGLG вызывается после модификации дескриптора страницы в ОЗУ.

П2.2.2. Адресное пространство и диспетчер памяти

Процессор имеет два основных режима работы — реальный и защищенный. В реальном режиме диспетчер памяти практически выключен и процессор имитирует работу процессора 8086.

Процессор имеет два физических адресных пространства: адресное пространство памяти (32-битное у большинства процессоров семейства, 36-битное у новых моделей, поддерживающих режим расширения адреса и 40-битное у IA-32e) и адресное пространство ввода/вывода (16-битное). Доступ к адресному пространству ввода/вывода осуществляется командами IN, OUT и некоторыми другими. Допустим также ввод/вывод с отображением регистров устройств на адреса памяти; в устройствах ISA и PCI это используется относительно редко, главным образом для отображения буферов большого объема. Наиболее известные устройства, использующие отображеные на память ввод/вывод — это видеоадаптеры. Как правило, управляющие регистры видеoadаптера размещены в адресном пространстве ввода/вывода, а видеобуфер — в адресном пространстве памяти.

Доступ к памяти осуществляется посредством интегрированного диспетчера памяти. Как мы видели, интеграция зашла настолько глубоко, что часть управляющих регистров диспетчера памяти (сегментные регистры) доступны

прикладному программисту. Доступ к адресному пространству ввода/вывода всегда происходит напрямую, без какой-либо трансляции.

Адресация памяти осуществляется с точностью до байта. Порядок байт в словах и двойных словах — little-endian (младший байт расположении по меньшему адресу). Доступ к памяти может осуществляться по отдельным байтам, словами (два байта) или двойными словами (четыре байта), а в архитектуре IA-32e — четвертыми словами (quadword длиной восемь байт). Отдельные команды могут работать со структурами данных других размеров, например 80-разрядными числами с плавающей точкой "полной" точности, 48-битными указателями и др.

Доступ к невыровненным словам и более крупным структурам возможен, но осуществляется за два цикла шины, поэтому, когда это возможно, "длинные" значения рекомендуется выравнивать. Современные реализации архитектуры имеют бит принудительного выравнивания в регистре CR0. Установка этого бита приводит к тому, что обращение к невыровненным словам, двойным словам и др. структурам приводит к генерации исключения, аналогичного ошибке шины у SPARC.

Ширина физической шины данных у разных реализаций архитектуры менялась в очень широких пределах, от 16 бит у 80386SX до 128 бит у Intel Pentium IV и AMD Athlon. Физическое адресное пространство также менялось. Так, 80386SX имел шину адреса шириной 24 бита, т. е. мог адресовать не более 16 Мбайт ОЗУ. Современные процессоры имеют 36-разрядную шину адреса, т. е. могут адресовать до 64 Гбайт ОЗУ. Большинство же процессоров семейства — 80386DX, 80486, Pentium, Pentium Pro/II/III, Celeron и др. — использовали 32-разрядную шину адреса.

П2.2.2.1. Сегментация

В реальном режиме 20-битный физический адрес образуется из 16-битного смещения и 16-битного "селектора" сегмента по очень простой формуле: значение "селектора" сдвигается влево на четыре бита и складывается со смещением. В действительности, таким образом можно получать 21-битные адреса в диапазоне от 0 до 0x10FFEF. В оригинальных 8086/8088 адресная шина была 20-битной, и дополнительные "почти 64 Кбайт" (64 Кбайт без 16 байт) были недоступны. На современных компьютерах DOS с драйвером HIMEM.SYS видит эти дополнительные килобайты под названием HMA (High Memory Area — область высокой памяти, не путать с Upper memory — ОЗУ в диапазоне от 640 Кбайт до границы 1 Мбайт) и может загружать в эту память драйверы, резидентные программы и даже часть своего собственного ядра.

Для обеспечения совместимости с программами для 8086, которые считали, что HMA должна отображаться в младшие адреса памяти, в материнскую

плату IBM PC AT (первый серийный компьютер на основе процессора 80286) была добавлена специальная логика, известная как gate A20 controller. При работе в реальном режиме процессора это устройство обрезало старшие биты физического адреса, оставляя ровно 1 Мбайт физического адресного пространства. При переходе в защищенный режим это устройство необходимо было перепрограммировать.

Кроме обрезания адреса, gate A20 controller выполняет еще одну своеобразную функцию. Дело в том, что при включении питания процессор запускается в "реальном" режиме. Затем его можно перевести в защищенный режим, но штатными средствами перевести его из защищенного обратно в реальный режим невозможно, или, точнее, для этого необходимо сброс процессора. Однако при работе Windows 3.x и DOS-эмулятора OS/2 необходимо было многократно переводить процессор из защищенного режима в реальный. Gate A20 controller позволяет сбросить процессор, не сбрасывая все остальные устройства компьютера и сохраняя содержимое оперативной памяти.

В защищенном режиме формат адреса зависит от типа сегмента. Два основных типа сегментов — это 16-разрядные и 32-разрядные сегменты кода и данных. Виртуальный адрес в 16-разрядном сегменте имеет длину 32 бита и состоит из 16-разрядного селектора и 16-разрядного беззнакового смещения. Адрес в 32-разрядном сегменте состоит из 16-разрядного сегмента и 32-разрядного смещения и имеет общую длину 48 бит.

Старшие 16 бит виртуальных адресов защищенного режима представляют собой селектор сегмента. Старшие два бита селектора указывают уровень доступа сегмента. Текущий уровень доступа процессора (CPL, Current Privilege Level), как правило, равен уровню доступа сегмента кода, который в данный момент исполняется: 0 соответствует наивысшему уровню, 3 — низшему.

Следующий (третий по старшинству) бит селектора указывает, в соответствии с какой из таблиц дескрипторов — LDT или GDT — будет осуществляться дальнейшая трансляция адреса.

Предполагается, что дескрипторы разделяемых сегментов ОС должна размещать в GDT, а приватные сегменты задачи — в LDT. Тогда при переключении задач достаточно перегрузить LDTR.

Процессор может осуществлять доступ только к сегментам с тем же или более низким уровнем доступа, чем CPL. Таким образом, код, исполняющийся на уровне 0, может иметь доступ к любым сегментам в системе; код, исполняющийся на уровне доступа 3, может работать только с сегментами уровня 3. Главный фокус состоит в том, что код с уровнем доступа 3 (или, как говорят программисты, работающий в кольце 3) не может вызывать процедуры в более высоких кольцах доступа и, таким образом, не может произвольно

повысить свой уровень доступа (знатоки архитектур 80286 и x86 должны отметить, что в действительности все немного сложнее, но для наших целей это описание удовлетворительно).

Благодаря этому, старшие биты селектора сегмента можно использовать в качестве полномочия, несмотря на то, что x86 разрешает битовые операции над указателями и позволяет сформировать произвольный селектор. Действительно, низкопrivилегированный код легко может синтезировать себе высокопrivилегированный селектор, но не сможет этим селектором воспользоваться.

Для повышения CPL предусмотрен специальный механизм, называемый шлюзом (*gate*), который рассматривается далее в этом разделе.

Процедуры из привилегированных сегментов могут понижать CPL при межсегментном возврате. Напротив, они не могут вызывать процедуры из менее привилегированных сегментов, ни напрямую, ни через шлюз. Поэтому нет необходимости в разрешении повышения CPL при возврате. К тому же такое разрешение привело бы к тому, что задачи могли бы повышать CPL, создавая в стеке фальшивые адреса возврата.

Дескриптор 16-битного сегмента содержит:

- тип сегмента (код, данные или специализированный);
- права доступа к сегменту (для сегментов данных это чтение и запись);
- направление роста (т. е. конец сегмента, от которого отсчитывается длина);
- линейный адрес начала сегмента с точностью до байта;
- 16-битную длину, также с точностью до байта (знатоки архитектуры x86 здесь вынуждены будут меня несколько подкорректировать — в зависимости от флага *Granularity* в дескрипторе, длина может изменяться как в байтах, так и в страницах). Если направление роста положительное, длина отсчитывается от начала сегмента, если отрицательное, то от его конца.

Дескриптор 32-битного сегмента также содержит тип, права, направление роста и длину. Длина сегмента, однако, занимает 20 бит и измеряется не в байтах, а в 4-килобайтных страницах.

После сегментной трансляции из виртуального адреса получается 32-битный линейный адрес, который подвергается страничной трансляции (рис. П2.2).

Эту своеобразную схему, по-видимому, невозможно объяснить требованиями 32-разрядных приложений, но она довольно-таки разумна, если вспомнить о необходимости обеспечивать совместимость с приложениями 8086 и 80286.

Так, при переходе в режим "виртуального 8086" процессор выключает сегментацию (рис. П2.3). При этом адрес, как и в реальном режиме, образуется

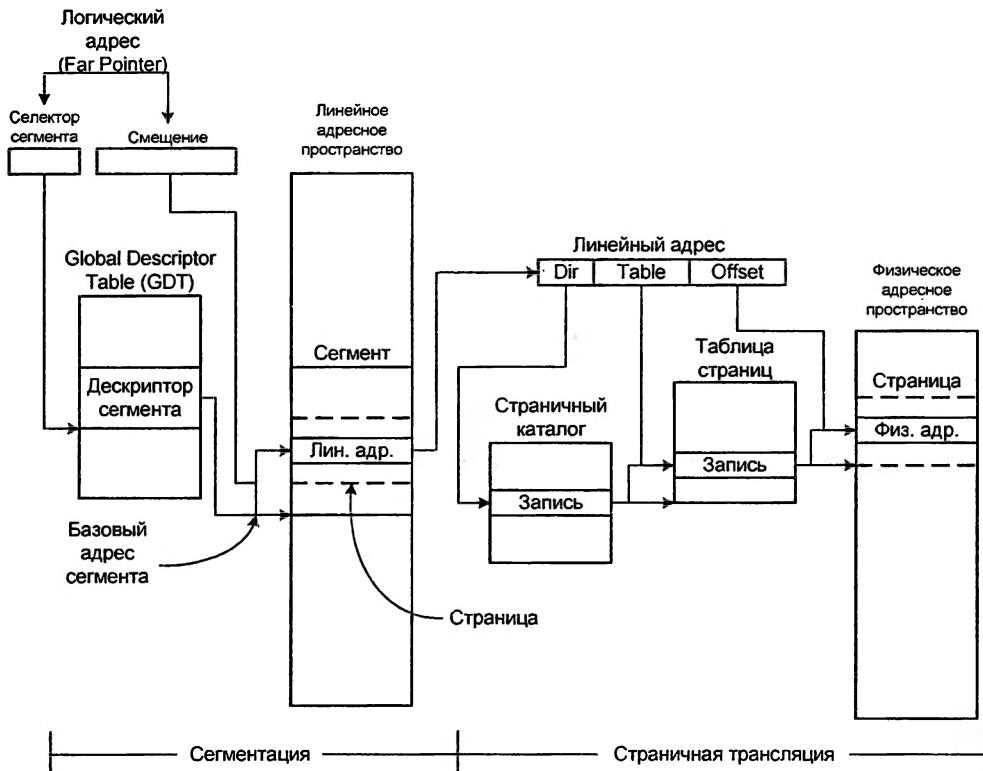


Рис. П2.2. Сегментная и страницчная трансляция

сложением селектора сегмента и смещения в нем, но на выходе получается не физический, а линейный виртуальный адрес. Этот адрес подвергается страницной трансляции так же, как и линейные адреса, получающиеся на выходе системы сегментации в защищенном режиме. Благодаря этому, процессор может иметь несколько образов адресного пространства 8086, отображенных на разные (возможно, но не обязательно, пересекающиеся) области физического ОЗУ. В каждой из таких областей можно создать собственную среду исполнения для программ, предназначенных для DOS.

В DOS-эмulyаторах OS/2, Windows NT и Linux в каждую такую область загружается собственный экземпляр DOS, модифицированный таким образом, чтобы обращения к дисковым устройствам транслировались в системные вызовы к ядру настоящей ОС (рис. П2.4). Если программа непосредственно обращается к портам ввода/вывода, это приводит к исключению по вызову привилегированной команды и опять-таки передаче управления ядру настоящей ОС.

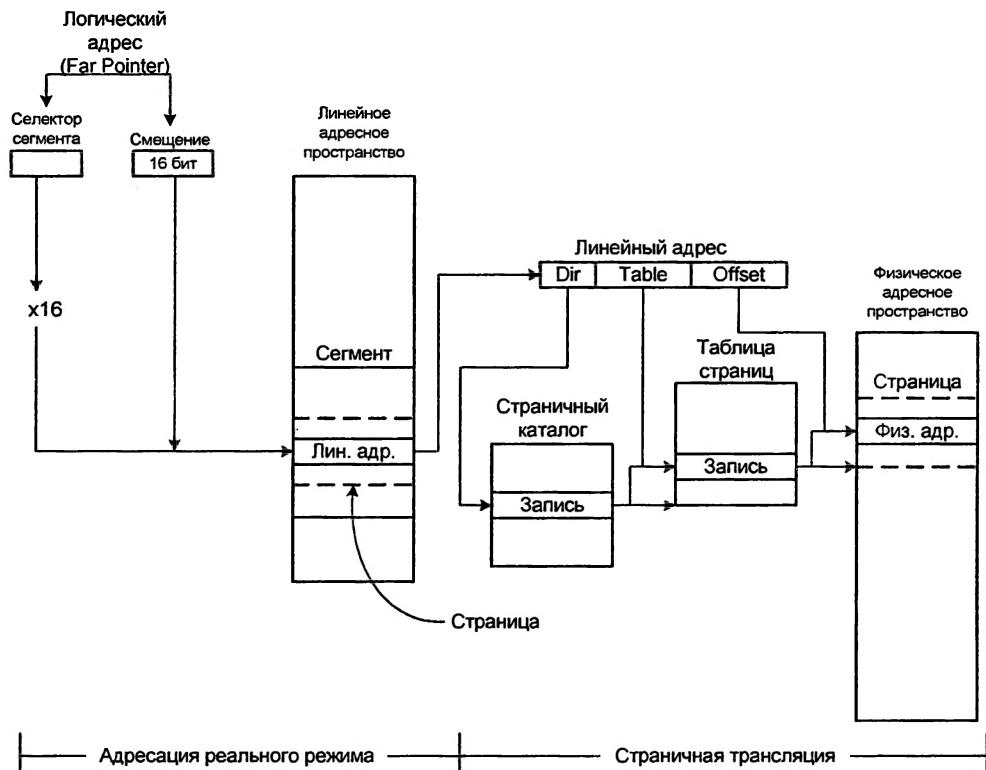


Рис. П2.3. Страницчная трансляция в "виртуальном 8086"

В Windows 95 и DesqView все виртуальные машины разделяют сегменты кода и данных DOS (рис. П2.5). Сама DOS при этом защищена группой семафоров, по одному семафору на каждую из нереентерабельных групп функций. Благодаря чему можно организовать довольно хорошее приближение к вытесняющей многозадачности, сохранив совместимость не только с программами для DOS, но и с драйверами и другими модулями ядра, предназначенными для этой системы.

В защищенном режиме страницно-сегментная трансляция позволяет организовать еще одну экстравагантную структуру адресного пространства, известную как *tiled memory* (плиточная память). При этом 512 Мбайт адресного пространства, которые можно адресовать, используя LDT и селекторы третьего уровня доступа, адресуются двумя способами: с помощью 32-битного линейного адреса и с помощью набора селекторов 16-битных сегментов. Это позволяет собрать единую программу из 32-битных и 16-битных модулей и передавать указатели на данные между этими модулями, подвергая их преобразованию по довольно простому алгоритму. Аналогичным образом можно

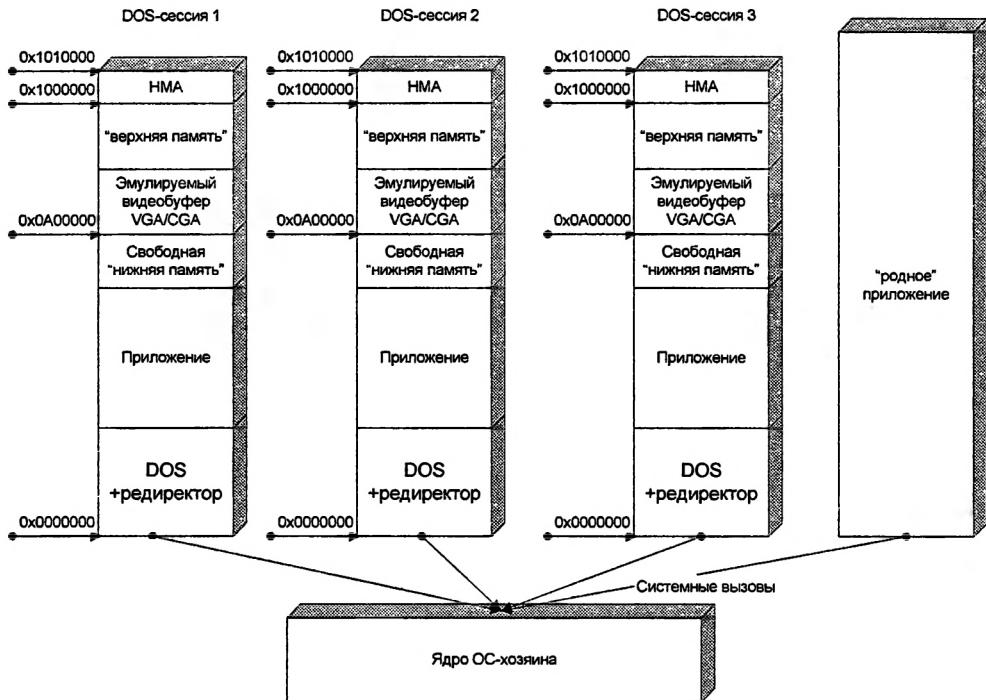


Рис. П2.4. Множественные сессии DOS в OS/2 и Windows NT

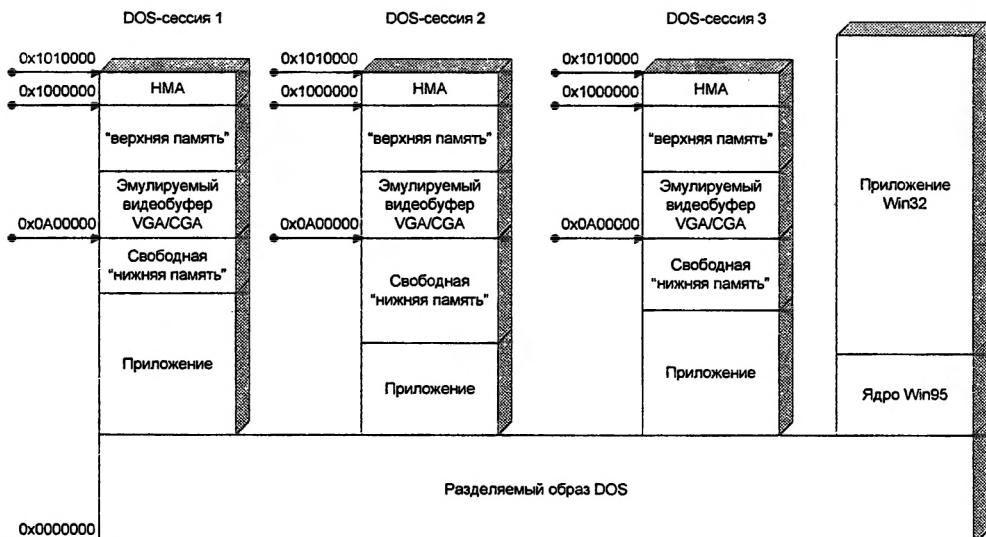
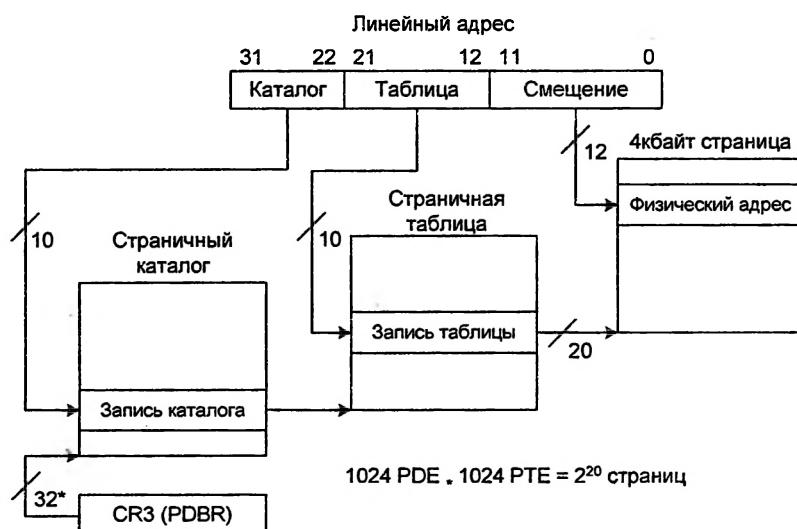


Рис. П2.5. Множественные сессии DOS в DesqView и Windows 95

создать 512-мегабайтное адресное пространство, адресуемое селекторами GDT нулевого уровня доступа, и использовать эти адреса для ядра. Именно таким образом 32-разрядные версии IBM OS/2 организуют совместное использование 16- и 32-битных DLL, а также 16- и 32-битных модулей ядра.

П2.2.2.2. Страницчная трансляция

Линейный адрес в 32-битном режиме разбит на три поля: 10-битный номер страницного каталога, 10-битный номер страницы в каталоге и 12-битное смещение на странице. Таким образом, линейное адресное пространство разбито на 1024 страницных каталога и 1 048 576 страниц по 4096 байт каждая (рис. П2.6).



*32 бита выровненные на 4кбайта

Рис. П2.6. Страницочные каталоги и страницы

Каждая запись страницного каталога описывает блок из 1024 последовательных дескрипторов страниц. Адресные пространства большинства задач занимают лишь малую долю 32-битного адресного пространства. Поэтому, как правило, большая часть записей страницного каталога не используется и соответствующие блоки страницных дескрипторов не создаются. Таким образом, таблица дескрипторов страниц обычно разрежена.

Дескрипторы страницных каталогов и страниц аналогичны по структуре (рис. П2.7). Каждый дескриптор содержит права доступа (чтение и запись, а в

последних версиях P IV и AMD Opteron также бит защиты от исполнения), бит присутствия, бит модификации (dirty), clock-бит или бит доступа (accessed), бит супервизора (supervisor), некоторые другие флаги и 20-битный физический адрес. Для страниц это адрес самой страницы, для страницных каталогов — начало таблицы страниц. Таким образом, и страница, и таблица страницных дескрипторов всегда должны быть выровнены на 4096 байт.

Запись страницного каталога (4кбайт страницы)

	31	12	11	9	8	7	6	5	4	3	2	1	0
Базовый адрес таблицы страниц		доступно	G	P	S	0	A	P	P	U	R	/	P
Доступно системным программистам	—												
Глобальная страница (игнорируется)	—												
Размер страницы (0 – 4 кбайт)	—												
Зарезервировано (должно быть 0)	—												
Страница использовалась	—												
Кэширование запрещено	—												
Сквозная запись	—												
Пользователь/супервизор	—												
Чтение/запись	—												
Присутствует	—												

Запись таблицы страниц (4кбайт страницы)

	31	12	11	9	8	7	6	5	4	3	2	1	0
Базовый адрес страницы		доступно	G	A	D	A	P	P	U	R	/	P	
Доступно системным программистам	—												
Глобальная страница	—												
Индекс атрибутов таблицы	—												
Грязная	—												
Использовалась	—												
Кэширование запрещено	—												
Сквозная запись	—												
Пользователь/супервизор	—												
Чтение/запись	—												
Присутствует	—												

Рис. П2.7. Структура дескриптора страницы

Любопытно, что размер дескриптора страницы — 4 байта, поэтому блок таблицы трансляции занимает 4096 байт — ровно одну страницу. Разумеется, это сильно упрощает жизнь операционной системе, полностью исключая внешнюю фрагментацию при управлении как страницами, так и таблицами трансляции.

Современные реализации x86 разрешают выключать второй уровень трансляции для всех или некоторых страничных каталогов — при этом запись страничного каталога превращается в дескриптор страницы размером 4 Мбайт. Также современные процессоры разрешают использовать четыре зарезервированных бита в дескрипторе для расширения физического адреса страницы, доводя таким образом физическое адресное пространство до 2^{36} байт или 64 Гбайт.

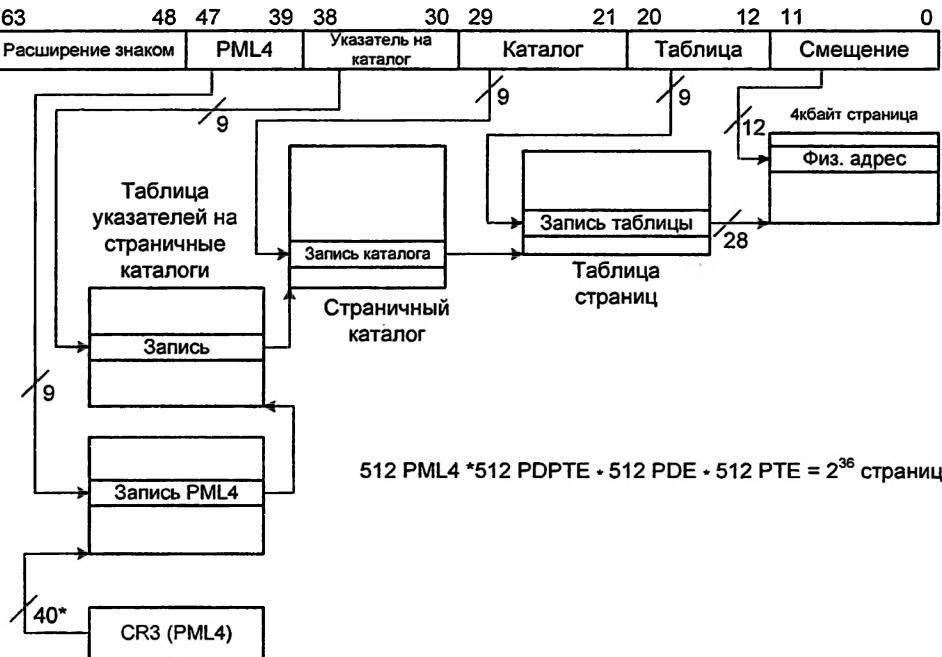
В IA-32e для перехода в 64-разрядный режим необходимо осуществить две операции:

1. Переключить формат таблиц трансляции. При этом длина дескриптора страничного каталога и страницы увеличивается до 64 бит. Размер страничного каталога при этом не изменяется, так что один страничный каталог теперь вмещает только 512 дескрипторов.
2. Переключить формат трансляции.

Трансляция (рис. П2.8) при этом становится четырехуровневой: значение CR3 теперь интерпретируется как указатель на PML4 (Page Map Level 4, страничную карту четвертого уровня), которая содержит указатели на таблицу указателей страничных каталогов.

Как уже говорилось, таблицы каждого из уровней трансляции имеют размер 4096 байт и содержат 512 дескрипторов таблиц следующего уровня. Поэтому адрес в режиме IA-32e разбит иначе, чем в 32-разрядном режиме. Младшие 12 бит представляют собой смещение в 4-килобайтной странице. Затем следуют четыре группы по 9 бит, соответствующие уровням трансляции: смещение в страничной таблице, страничном каталоге, таблице указателей страничных каталогов и, наконец, в PML4. Эти группы в совокупности образуют 48 бит. Старшие 16 бит адреса не используются, но зарезервированы для использования в перспективных реализациях IA-32e. Настоятельно рекомендуется расширять 48-битный адрес до 64 бит со знаком, т. е. так, чтобы старшие 16 бит адреса были равны 48-му биту.

Большинство ОС и систем программирования для x86 практически не используют сегментацию. Пользовательская задача получает один или два селектора сегмента — для кода и для данных, которые имеют нулевой начальный адрес и длину, соответствующую всему адресному пространству задачи. Затем задача использует 32-разрядные смещения в этих сегментах как линейные адреса, и работает с ними как с линейным адресным пространством. Сегментные регистры при этом никогда не перегружаются, да их, собственно, и нечем перегружать — это единственный валидный селектор сегмента, доступный для третьего кольца, в котором работает задача. Исключения из этого правила редки.



* 40 бит, выровненные на 4 килобайта

Рис. П2.8. Страницчная трансляция в 64-битном режиме

Так, в Solaris/x86 можно запретить исполнение стека даже на тех процессорах x86, которые не имеют бита защиты от исполнения в дескрипторах страниц. Для этого сегменты кода и данных делаются не полностью идентичными — сегмент кода не включает в себя страниц, соответствующих стеку. Это достигается уменьшением длины сегмента данных по сравнению с длиной сегмента кода (рис. П2.9).

В Solaris и Linux 2.6 в сегментном регистре **gs** в многопоточных программах хранится адрес дескриптора текущей нити. Поскольку стандартные 32-разрядные компиляторы не используют **gs**, этот регистр можно применять для относительно быстрого доступа к дескриптору и приватным данным текущей нити (*thread-specific data*). Аналогичным образом хранится ссылка на приватные данные текущей нити в Win32, но вместо регистра **gs** используется регистр **fs**.

Кроме сегментов кода и данных, x86 поддерживает следующие типы системных дескрипторов:

- LDT, Local Descriptor Table — локальная таблица дескрипторов;
- TSS, Task State Segment — сегмент состояния задачи;

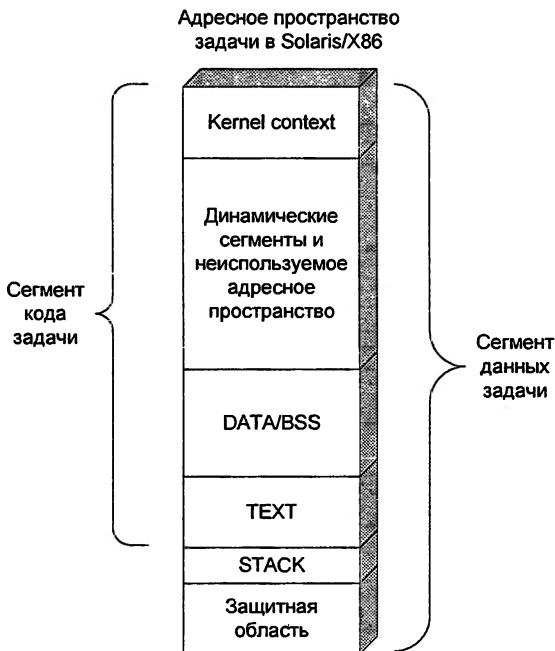


Рис. П2.9. Защита стека от исполнения в Solaris/x86

- Call Gate — шлюз (это понятие будет разъяснено далее);
- Interrupt Gate — шлюз прерывания, аналог вектора прерывания в более простых процессорах;
- Trap Gate — шлюз исключения;
- Task Gate — шлюз задачи, используется для переключения между задачами.

Сегмент состояния задачи содержит описание состояния задачи, которое включает в себя:

- значения сегментных регистров и регистров общего назначения;
- значение слова состояния;
- управляющий регистр CR3 (указатель на таблицу страничной трансляции);
- селектор сегмента LDT;
- три указателя стека, используемые на уровнях привилегий 0, 1 и 2;
- некоторые другие параметры.

При передаче управления на сегмент задачи текущее состояние процессора сохраняется в текущем сегменте задачи (селектор которого хранится в ре-

стре TR). Затем новое состояние восстанавливается из указанного сегмента, а его селектор загружается в TR.

По идее, этот механизм можно было бы использовать в планировщике процессов, но большинство современных ОС обходятся одним селектором задачи на всю систему, а переключение контекстов процессов осуществляют вручную. Главной функцией сегмента задачи оказывается поддержание трех разных указателей стека для разных уровней привилегий — эти указатели используются при повышении уровня привилегий через шлюз и при прерываниях.

Основная функция шлюзов упоминалась ранее в этом разделе. Шлюз обеспечивает контролируемое повышение уровня привилегий процессора. Вместо физического адреса сегмента и его длины дескриптор шлюза содержит 16-битный селектор какого-то другого сегмента (это должен быть сегмент кода) и 16- или 32-битное смещение в этом сегменте, в зависимости от типа целевого сегмента. Кроме того, четыре байта в селекторе шлюза отведено под счетчик параметров.

Вызов шлюза приводит к тому, что управление передается на точку, на которую указывает шлюз. Отличие от обычного вызова состоит в том, что при этом допускается повышение уровня привилегий. Шлюз можно вызывать обычной командой CALL. Необходимо использовать полный адрес (32-разрядный в 16-битном режиме, 48-разрядный в 32-битном), но при этом только селектор сегмента имеет значение; смещение в сегменте игнорируется.

При повышении привилегий через шлюз происходит переключение стека. При этом в регистры SS и ESP загружаются значения из дескриптора задачи, а старые селектор сегмента и указатель стека сохраняются в новом стеке. Затем происходит копирование параметров: из селектора шлюза берется счетчик параметров и соответствующее количество слов копируется из старого стека в новый. Затем в новом стеке сохраняются CS:EIP и происходит передача управления на адрес, определенный дескриптором шлюза.

Как уже говорилось, при возврате допустимо понижение привилегий. При обработке команды возврата процессор видит по уровню привилегий селектора адреса возврата, что необходимо переключение стека, и восстанавливает регистры CS:EIP и SS:ESP из текущего стека. Повышение привилегий при возврате недопустимо.

Шлюзы прерываний и исключений в норме не вызываются напрямую. Селекторы соответствующих сегментов перечислены в таблице IDT (Interrupt Descriptor Table); номер прерывания или исключения является индексом в этой таблице. При возникновении прерывания происходит вызов процедуры, описываемой соответствующим шлюзом.

Современные ОС для x86 используют для реализации системных вызовов как шлюзы вызова, так и шлюзы прерывания. Так, ОС, реализующие стандарт iBCS (Intel Binary Compatibilty Standard) — SCO Unix, SCO UnixWare, Sun Solaris/x86, эмуляторы iBCS для Linux и ветвей BSD, — используют для системных вызовов шлюз с селектором 0. Linux/x86 применяет для системных вызовов команду Int 0x83 (программное прерывание с номером 0x83). Системы семейства Win32 используют для той же цели прерывание 0x2E.

П2.2.3. Прерывания и исключения

Внешние прерывания обрабатываются внешней микросхемой APIC (Advanced Programmable Interrupt Controller). Как правило, APIC имеет несколько входных линий запроса прерывания (IRQ, Interrupt ReQuest) и одну выходную. Выходная линия может подключаться как к процессору, так и к другому APIC, допуская, таким образом, их каскадное соединение.

В однопроцессорных PC-совместимых компьютерах используются микросхемы, совместимые с Intel 8259. В современных процессорах Intel функциональный аналог 8259 реализован на кристалле процессора, однако логически процессор должен рассматривать APIC как внешнее устройство, отображенное на память, и обращаться к его регистрам командами обращения к ОЗУ по соответствующим адресам.

Общение между APIC и процессором в разных реализациях происходит либо с помощью циклов системной шины, либо по специализированной последовательной шине. Получив прерывание от внешнего источника (или от периферийного APIC), контроллер прерываний выставляет запрос на линии запроса прерывания процессора. Когда процессор оказывается готов обработать прерывание (у современных суперскалярных процессоров это может потребовать значительного количества тактов), он выставляет сигнал подтверждения. Затем APIC передает процессору номер прерывания. Процессор принимает его, выбирает из IDT селектор с соответствующим номером (это должен быть селектор шлюза прерывания или шлюза задачи), сохраняет в стеке соответствующего уровня привилегий CS:EIP и слово состояния процессора и вызывает процедуру, описываемую этим шлюзом. Стек выбирается из текущего TSS в соответствии с уровнем привилегий обработчика прерывания. Сохранение остальных регистров процессора возлагается на обработчик прерывания.

Начав обработку прерывания, процессор обязан подтвердить, что он ее начал, установив соответствующие биты в одном из регистров APIC. Особенно это важно в многопроцессорных plataх. Также обычно требуется перепрограммировать APIC, указав ему новый уровень приоритета. Последовательность операций над APIC, которая это делает, называется прологом прерывания.

При возврате из прерывания процессор автоматически не восстанавливает приоритет APIC, поэтому перед возвратом обработчик также должен совершить серию магических действий над контроллером, так называемый "эпилог прерывания".

В ядрах большинства современных ОС существует системный сервис, реализующий пролог и эпилог прерывания, причем делающий это корректно как для одно-, так и для многопроцессорных систем, так что разработчику драйвера обычно не нужно связываться с программированием APIC, а достаточно зарегистрировать в качестве обработчика прерывания обычную функцию, написанную на языке С.

Контроллер 8529 имеет восемь входных IRQ. В первых IBM PC использовался один такой контроллер; в IBM PC/AT и более современных системах установлены два каскадно подключенных контроллера, обеспечивающие обработку 15 линий запроса прерывания. Большинство из этих линий заняты стандартными периферийными устройствами IBM PC (см. табл. П2.1). В современных серверных платах с несколькими шинами PCI количество IRQ расширено: как правило, каждая шина PCI получает четыре линии запроса прерывания. Но в платах для персональных компьютеров с одной шиной PCI общее количество линий осталось тем же самым.

Таблица П2.1. Распределение линий запроса прерывания в IBM PC/AT

Номер	Описание
IRQ0	Системный таймер (8253/8254 Programmable Interval Timer)
IRQ1	Клавиатура
IRQ2	Каскадирование ко второму APIC
IRQ3	Порты RS232 COM2 и COM4
IRQ4	Порты RS232 COM1 и COM3
IRQ5	Параллельный порт Centronix LPT2
IRQ6	Контроллер гибких дисков
IRQ7	Параллельный порт Centronix LPT1
IRQ8	Часы реального времени (RTC)
IRQ9	IRQ 2 шины ISA (кадровый гасящий импульс EGA/VGA)
IRQ10	Используются для сетевых карт, SCSI HBA и других нестандартных устройств
IRQ11	
IRQ12	Мышь PS/2
IRQ13	Сопроцессор

Таблица П2.1 (окончание)

Номер	Описание
IRQ14	Первый контроллер ST506/ESDI/ATA
IRQ15	Второй контроллер ST506/ESDI/ATA

Таким образом, при подключении нестандартных устройств (звуковых карт, SCSI HBA, сетевых адаптеров) всегда очень острой была проблема нехватки IRQ и конфликты между драйверами по их поводу. Как правило, поставщики таких устройств делали на плате блок перемычек ("джамперов", от англ. jumper), с помощью которых можно было выбирать IRQ, используемый устройством. Обычно допустимые значения включали в себя линии 3, 5 и 7 (такие платы можно было использовать в компьютерах IBM PC/XT, в которых был только один 8529); у плат, рассчитанных на IBM PC/AT, допускались также линии 10 и 11. Мне доводилось видеть платы, позволявшие задать любое значение в диапазоне от 2 до 11.

Переход к периферийным шинам с автоконфигурацией (ISA PnP и, затем, PCI) не решил этой проблемы и даже, пожалуй, усугубил ее. Действительно, без автоконфигурации владелец системы хотя бы знал, что на какой IRQ он "повесил" — если только на плате было подписано, что данная конфигурация перемычек означает, или где-то под рукой была документация к этой плате. При автоконфигурации же владелец системы зачастую полностью терял контроль над происходящим — целые подсистемы ОС молча отказывались работать, ОС падала в "синий экран" при загрузке или при первой операции с устройством и т. д. Особенно серьезной была эта проблема при совместном использовании старых, не-PnP, устройств и новых, допускающих автоконфигурацию, — практически всегда BIOS назначал PnP-устройству IRQ, занятое старой платой.

PnP BIOS второго поколения, появившиеся на рынке в 1996—97 годах, позволяют хотя бы указать, какие из IRQ не следует занимать при автоконфигурации. Значительного улучшения ситуации смогли добиться лишь поставщики ОС и драйверов, которые научили драйверы разделять IRQ между собой. Тем не менее многие, особенно старые, драйверы (в том числе и драйверы устройств PnP) так этому и не научились. Не имея возможности переставить перемычки, администратор системы часто оказывается вынужден совершать совершенно магические действия, переставляя устройства из разъема в разъем с целью изменить порядок, в котором BIOS назначает им IRQ. Не случайно среди расшифровок аббревиатуры PnP встречаются такие варианты, как Plug and Pray (подключи и молись).

При рекламе шины USB среди преимуществ часто называлось решение проблемы IRQ — действительно, контроллер USB может использоваться для подключения принтеров, последовательных устройств, мыши, клавиатуры и т. д. Однако действительность оказалась богаче. Так, контроллер USB 2.0 стандарта EHCI поддерживает до восьми портов. При подключении к этим портам устройств USB 1.x он переходит в режим эмуляции контроллера OHCI, который поддерживал только два порта. Таким образом, контроллер EHCI может содержать четыре эмулируемых устройства OHCI, каждому из которых требуется своя линия IRQ — т. е. один контроллер EHCI в полной конфигурации занимает 5 IRQ!

Важно отметить, что нехватка IRQ — это проблема не процессоров x86, а материнских плат IBM PC и периферийной шины ISA, в которой количество линий IRQ также ограничено. Каскадно подключая APIC, можно получать практически неограниченное количество линий запроса прерывания, как это, собственно, и делается в серверных материнских платах.

В многопроцессорных системах, кроме локального APIC у каждого процессора, система должна иметь один или несколько периферийных APIC (как правило, по одному на каждую из периферийных шин). Эти контроллеры обеспечивают маршрутизацию внешних прерываний и их доставку процессорам. Также эти APIC можно использовать для посылки прерываний между процессорами. Материнские платы с количеством процессоров от двух до четырех обычно используют контроллеры прерываний, совместимые с Intel 82093AA; как правило, эти устройства не реализуются на отдельной микросхеме, а интегрируются в микросхему контроллера системной шины (так называемый "чипсет").

Процессор имеет две линии запроса прерывания — маскируемое и немаскируемое прерывания. Теоретически, APIC можно присоединить к любой из этих линий, но обычно его присоединяют только к линии маскируемого прерывания.

Маскируемые прерывания могут блокироваться флагом запрета прерывания в слове состояния процессора (откуда и название). Немаскируемое прерывание происходит всегда, независимо от каких бы то ни было битов. В PC-совместимых компьютерах, в которых используется память с контролем четности, немаскируемое прерывание генерируется при ошибках четности ОЗУ; ОС при таком прерывании немедленно прекращает работу и рисует "экран смерти" с соответствующим сообщением.

Кроме внешних прерываний, процессоры x86 поддерживают достаточно богатый набор аппаратных исключений, в том числе — ошибки сегментации, ошибки страничного доступа и страничные отказы, арифметические исключения и т. д.

П2.3. Система команд и режимы адресации

Процессор реализует две разные системы команд — собственно x86 (32-разрядные команды) и 8086/80286 (16-разрядные команды). Коды команд в этих системах практически совпадают, но наборы режимов адресации и способ их кодирования сильно отличаются. Процессор определяет систему команд в зависимости от типа сегмента кода: в 16-битовых сегментах используются 16-разрядные команды, а в 32-битовых, соответственно, 32-разрядные.

Используя префиксы размера адреса и размера данных, можно вставлять в 16-битовые сегменты одиночные 32-разрядные команды, и наоборот, в 32-битовые — одиночные 16-разрядные. Далее в этом разделе будет обсуждаться только 32-разрядная система команд.

x86 имеет систему команд регистр-память с довольно богатым и почти симметричным набором режимов адресации. Основные команды (передача данных, сложение, вычитание и т. д.) двухадресные.

Некоторые специализированные команды (например, строковые операции) работают в режиме память-память и используют нестандартные режимы адресации. Как правило, эти команды используют регистры ESI и EDI в качестве указателей operandов.

Команды работы со строками (CMPS, STOS, LODS, INS, OUTS) используют в качестве неявных operandов косвенную адресацию по регистрам ESI/EDI с автоВинкрементом или автоДекрементом. Направление движения по строке (т. е. происходит ли инкремент или декремент) определяется битом направления в слове состояния процессора. Эти команды могут использоваться с префиксами повторения; в этом случае регистр ECX применяется в качестве счетчика количества повторений.

В системе команд также определено довольно много одноадресных команд, но трех- и более адресных команд не предусмотрено. В целом, систему команд трудно назвать ортогональной, потому что очень много команд используют определенные регистры в качестве неявного operandана.

Так, команды целочисленного умножения и деления кодируются как одноадресные, при этом регистровая пара EAX:EDX используется в качестве неявного operandана.

При умножении 32-разрядный operand команды умножается на 32-разрядное значение регистра EAX. Получившееся 64-разрядное число размещается в регистрах EAX:EDX (младшие биты в EAX, старшие в EDX).

При делении 64-разрядное делимое, размещенное в регистрах EAX:EDX, делится на 32-разрядный operand-делитель. Частное размещается в EAX, остаток — в EDX.

Команды имеют переменную длину. Минимальная длина составляет один байт, максимальную длину определить затруднительно, потому что код команды может расширяться за счет префиксов, но многие префиксы работают лишь с некоторыми командами. По сообщению [Паппас/Марри 1993], процессоры 80386 генерировали исключение по недопустимой команде, если команда имела длину 15 байт или более, но современная документация Intel ни о чем подобном не сообщает.

Формат команды приводится на рис. П2.10. Видно, что команда состоит из нуля, одного или нескольких префиксов, одного или двух байт кода команды, одного или двух байт, кодирующих режим адресации, а также, возможно, адресного смещения длиной от одного до четырех байт и литерала длиной от одного до четырех байт. Наличие адресного смещения определяется режимом адресации, наличие литерала — кодом команды.

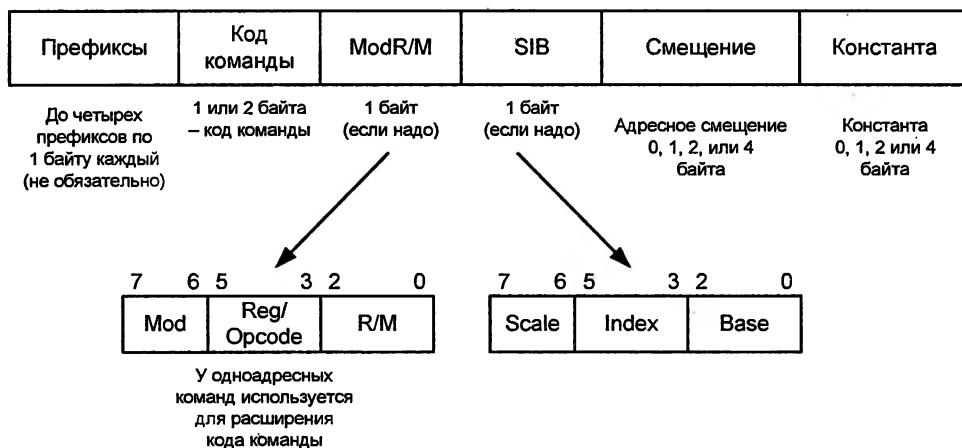


Рис. П2.10. Формат команды x86, цит. по www.intel.com/Pentium4

Таким образом, максимальная длина команды без префиксов составляет 12 байт. С такой командой гарантированно могут использоваться два префикса — префикс 32-битной адресации и префикс замены сегментного регистра, что расширяет длину команды до 14 байт.

Некоторые команды, а именно, межсегментный переход и межсегментный вызов, используют шестибайтовый адрес формата сегмент:смещение. К счастью, такие команды не могут иметь других необязательных полей (например, литерала) и не могут использовать большинство префиксов.

Вариации формата типичной двухадресной команды, такой как ADD или MOV, приведены на рис. П2.11. Видно, что каждая такая команда — это, в действительности, группа команд с разными форматами и разными кодами операций.

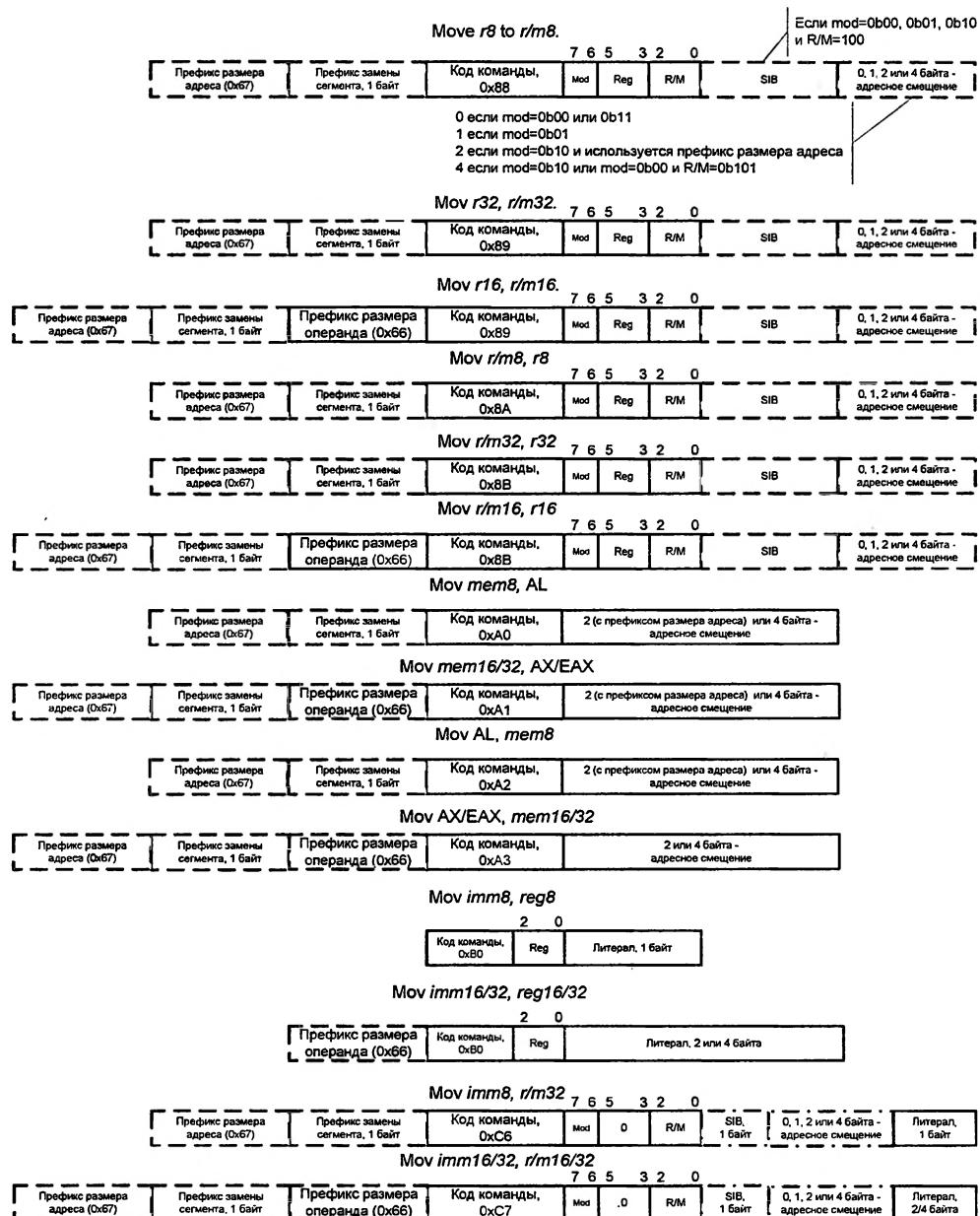


Рис. П.2.11. Варианты формата команды MOV
(команды IA-32e с префиксом RAX не показаны)

Так, для основных арифметических операций предоставляются следующие команды:

- универсальная команда регистр-память. Первый операнд размещается в регистре, второй в регистре или памяти, допустимы все режимы адресации;
- команда память-регистр. Первый операнд размещается в регистре или памяти, допустимы все режимы адресации. Второй операнд размещается в регистре. Так, команда `mov` формата регистр-память перемещает операнд из памяти в регистр, а формата память-регистр — из регистра в память. Пересылку регистр-регистр можно закодировать двумя разными способами;
- команда литерал-память. Один операнд — регистр или память, допустимы все режимы адресации. Второй операнд — литерал;
- сокращенная форма команды регистр-память. Код операции однобайтовый, один операнд размещен в регистре второй операнд размещен в памяти, допустима только абсолютная адресация в сегменте DS; если команда имеет префикс замены сегмента, вместо DS может использоваться другой сегментный регистр;
- сокращенная форма команды литерал-регистр. Операнд размещен в регистре или даже только в регистре EAX;

Далеко не все команды имеют сокращенные формы.

Команды для операций с плавающей точкой обычно имеют две формы:

- регистр-память. Один операнд размещается в регистре сопроцессора, другой в регистре сопроцессора либо в памяти. Допустимы все режимы адресации, с тем отличием, что в регистровом режиме вместо регистров общего назначения используются соответствующие по номеру регистры сопроцессора; в косвенно-регистровых и т. д. режимах адресации используются регистры общего назначения;
- стековые. Команда выталкивает операнды из регистрового стека и проталкивает в этот стек результат. Команды сравнения имеют еще одну любопытную форму, которая выталкивает из стека два операнда и ничего не проталкивает в стек. Некоторые из стековых команд, например `FSINCOS`, имеют два результата — в случае команды `FSINCOS` это синус и косинус операнда.

Редко используемые команды, такие как вычисление тригонометрических функций, имеют только стековую форму.

Как уже говорилось, оба операнда двухадресной команды кодируются байтом, разбитым на три битовых поля: `mod reg r/m`. В зависимости от режима адресации, код операнда может быть расширен байтом `SIB` и/или адресным смещением, которое может иметь длину от одного до четырех байт.

Поля `reg` и `r/m` трехбитовые. Один из операндов обязан находиться в регистре, номер которого кодируется полем `reg`. При этом, в зависимости от команды, это может быть либо регистр общего назначения, либо регистр сопроцессора, либо управляющий, отладочный или служебный регистр. У одноадресных команд поле `reg` используется для расширения кода команды.

Второй operand кодируется битами `mod` и `r/m` и может храниться либо в памяти, либо в регистре общего назначения. Многие способы кодирования операнда предполагают увеличение длины команды за счет байта `SIB` (его формат описывается далее) и/или адресного смещения. Дополнительные поля следуют за байтом операнда.

Двухбитовое поле `mod` обозначает режим адресации. В первом приближении схема кодирования проста и ортогональна:

- `Mod=00` — косвенно-регистровый режим адресации (номер регистра в поле `r/m`);
- `Mod=01` — косвенно-регистровый режим с байтовым (8-разрядным) адресным смещением;
- `Mod=10` — косвенно-регистровый режим с четырехбайтовым (32-разрядным) смещением;
- `Mod=11` — регистровый режим (номер регистра в поле `r/m`).

В обоих режимах со смещением адресное поле представляет собой число в двоично-дополнительном коде, т. е. байтовое смещение соответствует смещениям от -128 до $+127$.

Впрочем, в этой ортогональной схеме есть два исключения:

- недопустимы ни косвенно-регистровая, ни косвенно-регистровая со смещением адресации с использованием `ESP`. Соответствующие комбинации кодов используются для кодирования базово-индексной адресации и базово-индексной адресации со смещением с использованием байта `SIB`;
- недопустима косвенно-регистровая адресация с использованием `EBP`; соответствующая комбинация кодов обозначает абсолютный адрес, размещаемый в четырехбайтовом поле смещения. При этом адресация со смещением (как байтовым, так и четырехбайтовым) относительно `EBP` допустима.

Формат байта `SIB` приводится на рис. П2.12. Адрес определяется по формуле $\text{BASE} + \text{INDEX} * 2^{\text{SS}} + \text{OFFSET}$, где:

- `BASE` — значение регистра с номером, определяемым битовым полем `BASE`;
- `INDEX` — соответственно, значение регистра с номером из битового поля `INDEX`;

- ss — размер элемента массива в байтах, или, точнее, двоичный логарифм этого размера, значение в диапазоне от 0 до 3. Таким образом, размер элемента массива может быть от 1 до 8 байт;
- OFFSET — адресное поле команды, размер которого кодируется полем mod основной команды и может составлять 0, 8 или 32 бит.

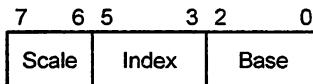


Рис. П2.12. Формат байта SIB

Как и при косвенно-регистровой адресации, есть исключение — при базово-регистровой адресации без смещения код регистра ЕВР обозначает использование в качестве базы абсолютного 32-разрядного адресного смещения.

Нелюбовь к регистру ЕВР имеет прагматическое объяснение — разработчикам системы команд просто не хватало кодов для обозначения абсолютной адресации. Кроме того, в норме регистр ЕВР используется для формирования стекового кадра; при этом регистр указывает на заголовок стекового кадра, конкретно — на сохраненное значение ЕВР. Обращения к этому полю в обычных программах происходят настолько редко, что вполне допустимо при таких обращениях использовать косвенно-регистровый режим со смещением.

Нелюбовь к регистру ESP могла бы быть также объяснена прагматическим соображением — нехваткой кодов, однако знакомые с набором режимов адресации 8086/80286 должны признать, что эта нелюбовь очень последовательна: данные процессоры также не допускали адресации относительно SP. Поэтому для формирования стекового кадра и адресации размещаемых в стеке локальных переменных разработчики компиляторов для 8086 должны были выделять регистр ЕВР.

Для сравнения, компиляторы С и Pascal для PDP-11 адресуют локальные переменные и параметры функций относительно SP. Это несколько усложняет для отладчика отслеживание цепочки стековых кадров, но экономит регистр — как и у x86, у PDP-11 всего восемь регистров, поэтому их экономия очень актуальна. Для сравнения, первыми выделили регистр для организации стекового кадра разработчики процессора VAX, у которого 16 регистров общего назначения.

Кроме регистров общего назначения, при каждой адресации памяти используется один из сегментных регистров. По умолчанию, при адресации данных с использованием регистра ЕВР и в командах PUSH, POP и других операциях со стеком используется сегментный регистр ss, а при адресации абсолютным адресом и при использовании остальных регистров общего назначения — DS.

Если необходимо использовать другие сегментные регистры, следует указать перед командой префикс замены сегмента.

Вся адресация кода, разумеется, осуществляется относительно CS.

В 64-разрядном режиме IA-32e схема кодирования команд и режимов адресации не изменяется, но 32-битные смещения становятся 64-битными. Для использования регистров R8-R15 необходимо указывать перед командой префикс расширенных номеров регистров REX. Префикс REX использует коды 0x40—0x4F, которые в 32-разрядном режиме обозначали команды INC и DEC для регистров. Младшие четыре бита кода префикса REX соответствуют старшим битам номеров всех четырех регистров, которые могут использоваться в команде: регистра-источника, регистра-назначения, базового и индексного регистров.

Приведем мнемоники, форматы и коды основных и привилегированных команд x86 в табл. П2.2. Здесь бит w обозначает размер операнда (0 — 1 байт, 1 — 4 байт); бит s обозначает расширение литерала со знаком. Значения поля tttn показаны в табл. П2.3. Мнемоники и коды команд сопроцессора (для операций с плавающей точкой) даны в табл. П2.4.

Таблица П2.2. Мнемоники, форматы и коды основных и привилегированных команд x86

Команда и формат	Код
AAA — ASCII Adjust after Addition (ASCII-коррекция сложения)	0011 0111
AAD — ASCII Adjust AX before Division (ASCII-коррекция деления)	1101 0101 : 0000 1010
AAM — ASCII Adjust AX after Multiply (ASCII-коррекция умножения)	1101 0100 : 0000 1010
AAS — ASCII Adjust AL after Subtraction (ASCII-коррекция вычитания)	0011 1111
ADC — ADD with Carry (сложение с переносом)	
регистр1 с регистр2	0001 000w : 11 reg1 reg2
регистр2 с регистр1	0001 001w : 11 reg1 reg2
память с регистром	0001 001w : mod reg r/m
регистр с памятью	0001 000w : mod reg r/m
литерал с регистром	1000 00sw : 11 010 reg : immediate data

Таблица П2.2 (продолжение)

Команда и формат	Код
литерал с AL, AX или EAX	0001 010w : immediate data
литерал с памятью	1000 00sw : mod 010 r/m : immediate data
ADD — Add (сложение)	
регистр1 с регистр2	0000 000w : 11 reg1 reg2
регистр2 с регистр1	0000 001w : 11 reg1 reg2
память с регистром	0000 001w : mod reg r/m
регистр с памятью	0000 000w : mod reg r/m
литерал с регистром	1000 00sw : 11 000 reg : immediate data
литерал с AL, AX или EAX	0000 010w : immediate data
литерал с памятью	1000 00sw : mod 000 r/m : immediate data
AND — Logical AND (логическое И)	
регистр1 с регистр2	0010 000w : 11 reg1 reg2
регистр2 с регистр1	0010 001w : 11 reg1 reg2
память с регистром	0010 001w : mod reg r/m
регистр с памятью	0010 000w : mod reg r/m
литерал с регистром	1000 00sw : 11 100 reg : immediate data
литерал с AL, AX или EAX	0010 010w : immediate data
литерал с памятью	1000 00sw : mod 100 r/m : immediate data
ARPL — Adjust RPL Field of Selector (изменить поле уровня привилегий селектора)	
из регистра	0110 0011 : 11 reg1 reg2
из памяти	0110 0011 : mod reg r/m
BOUND — Check Array Against Bounds (проверка границ индекса)	
	0110 0010 : modAreg r/m
BSF — Bit Scan Forward (сканирование бит вперед)	
регистр1, регистр2	0000 1111 : 1011 1100 : 11 reg1 reg2
память, регистр	0000 1111 : 1011 1100 : mod reg r/m

Таблица П2.2 (продолжение)

Команда и формат	Код
BSR — Bit Scan Reverse (сканирование бит назад)	
регистр1, регистр2	0000 1111 : 1011 1101 : 11 reg1 reg2
память, регистр	0000 1111 : 1011 1101 : mod reg r/m
BSWAP — Byte Swap (перестановка байтов)	0000 1111 : 1100 1 reg
BT — Bit Test (проверка бита)	
регистр, литерал	0000 1111 : 1011 1010 : 11 100 reg: imm8 data
память, литерал	0000 1111 : 1011 1010 : mod 100 r/m : imm8 data
регистр1, регистр2	0000 1111 : 1010 0011 : 11 reg2 reg1
память, регистр	0000 1111 : 1010 0011 : mod reg r/m
BTC — Bit Test and Complement (проверка и инверсия бита)	
регистр, литерал	0000 1111 : 1011 1010 : 11 111 reg: imm8 data
память, литерал	0000 1111 : 1011 1010 : mod 111 r/m : imm8 data
регистр1, регистр2	0000 1111 : 1011 1011 : 11 reg2 reg1
память, регистр	0000 1111 : 1011 1011 : mod reg r/m
BTR — Bit Test and Reset (проверка и сброс бита)	
регистр, литерал	0000 1111 : 1011 1010 : 11 110 reg: imm8 data
память, литерал	0000 1111 : 1011 1010 : mod 110 r/m : imm8 data
регистр1, регистр2	0000 1111 : 1011 0011 : 11 reg2 reg1
память, регистр	0000 1111 : 1011 0011 : mod reg r/m
BTS — Bit Test and Set (проверка и установка бита)	
регистр, литерал	0000 1111 : 1011 1010 : 11 101 reg: imm8 data
память, литерал	0000 1111 : 1011 1010 : mod 101 r/m : imm8 data

Таблица П2.2 (продолжение)

Команда и формат	Код
регистр1, регистр2	0000 1111 : 1010 1011 : 11 reg2 reg1
память, регистр	0000 1111 : 1010 1011 : mod reg r/m
CALL — Call Procedure — вызов процедуры (в том же сегменте)	
прямой	1110 1000 : full displacement
регистровый косвенный	1111 1111 : 11 010 reg
косвенный через память	1111 1111 : mod 010 r/m
CALL — Call Procedure — вызов процедуры (в другом сегменте)	
прямой	1001 1010 : unsigned full offset, selector
косвенный	1111 1111 : mod 011 r/m
CBW — Convert Byte to Word — преобразовать байт в слово	1001 1000
CDQ — Convert Doubleword to Qword — преобразовать двойное слово в четверное слово	1001 1001
CLC — Clear Carry Flag — очистить флаг переноса	1111 1000
CLD — Clear Direction Flag — очистить флаг направления	1111 1100
CLI — Clear Interrupt Flag — очистить флаг [разрешения] прерываний	1111 1010
CLTS — Clear Task-Switched Flag in CR0 — очистить флаг переключения задачи в CR0	0000 1111 : 0000 0110
CMC — Complement Carry Flag — обратить флаг переноса	1111 0101
CMP — Compare Two Operands — сравнить	
регистр1 с регистром2	0011 100w : 11 reg1 reg2
регистр2 с регистром1	0011 101w : 11 reg1 reg2
память с регистром	0011 100w : mod reg r/m
регистр с памятью	0011 101w : mod reg r/m
литерал с регистром	1000 00sw : 11 111 reg : immediate data

Таблица П2.2 (продолжение)

Команда и формат	Код
литерал с AL, AX или EAX	0011 110w : immediate data
литерал с памятью	1000 00sw : mod 111 r/m : immediate data
CMPS/CMPSB/CMPSW/CMPSD — Compare String Operands — строковое сравнение	1010 011w
CMPXCHG — Compare and Exchange — сравнить и обменять	
регистр1, регистр2	0000 1111 : 1011 000w : 11 reg2 reg1
память, регистр	0000 1111 : 1011 000w : mod reg r/m
CPUID — CPU Identification — идентификатор процессора	0000 1111 : 1010 0010
CWD — Convert Word to Doubleword — преобразовать слово в двойное слово	1001 1001
CWDE — Convert Word to Doubleword — преобразовать слово в двойное слово	1001 1000
DAA — Decimal Adjust AL after Addition — десятичная коррекция после сложения	0010 0111
DAS — Decimal Adjust AL after Subtraction — десятичная коррекция после вычитания	0010 1111
DEC — Decrement by 1 — уменьшить на 1	
регистр	1111 111w : 11 001 reg
регистр (альтернативное кодирование)	0100 1 reg
память	1111 111w : mod 001 r/m
DIV — Unsigned Divide — беззнаковое деление	
AL, AX или EAX с регистром	1111 011w : 11 110 reg
AL, AX или EAX с памятью	1111 011w : mod 110 r/m
ENTER — Make Stack Frame for High Level Procedure — создать стековый кадр для процедуры ЯВУ	1100 1000 : 16-bit displacement : 8-bit level (L)
HLT — Halt — останов процессора	1111 0100

Таблица П2.2 (продолжение)

Команда и формат	Код
IDIV — Signed Divide — знаковое деление	
AL, AX или EAX с регистром	1111 011w : 11 111 reg
AL, AX или EAX с памятью	1111 011w : mod 111 r/m
IMUL — Signed Multiply — знаковое умножение	
AL, AX или EAX с регистром	1111 011w : 11 101 reg
AL, AX или EAX с регистром	1111 011w : mod 101 reg
регистр1 с регистром 2	0000 1111 : 1010 1111 : 11 : reg1 reg2
регистр с памятью	0000 1111 : 1010 1111 : mod reg r/m
регистр с литералом с регистром 2	0110 10s1 : 11 reg1 reg2 : immediate data
память с литералом с регистром	0110 10s1 : mod reg r/m : immediate data
IN — Input From Port — ввод из порта	
фиксированный порт	1110 010w : port number
переменный порт (номер порта в DX)	1110 110w
INC — Increment by 1 — увеличить на 1	
регистр	1111 111w : 11 000 reg
регистр (альтернативное кодирование)	0100 0 reg
память	1111 111w : mod 000 r/m
INS — Input from DX Port — ввод строки (адрес порта в DX)	0110 110w
INT n — Interrupt Type n — прерывание n	1100 1101 : type
INT — Single-Step Interrupt 3 — прерывание 3 (отладочная точка останова)	1100 1100
INTO — Interrupt 4 on Overflow — прерывание 4 при переполнении	1100 1110
INVD — Invalidate Cache — сбросить кэш	0000 1111 : 0000 1000
INVLPG — Invalidate TLB Entry — сбросить запись TLB	0000 1111 : 0000 0001 : mod 111 r/m
IRET/IRETД — Interrupt Return — возврат из прерывания	1100 1111

Таблица П2.2 (продолжение)

Команда и формат	Код
Jcc — Jump if Condition is Met — условный переход	
8-битное смещение	0111 tttn : 8-bit displacement
полное смещение	0000 1111 : 1000 tttn : full displacement
JCXZ/JECXZ — Jump on CX/ECX Zero — переход по CX/ECX==0 Предфикс размера адреса различает CX и ECX	1110 0011 : 8-bit displacement
JMP — Unconditional Jump (to same segment) — безусловный переход (внутрисегментный)	
короткий	1110 1011 : 8-bit displacement
прямой	1110 1001 : full displacement
регистровый косвенный	1111 1111 : 11 100 reg
косвенный через память	1111 1111 : mod 100 r/m
JMP — Unconditional Jump (to other segment) безусловный переход (межсегментный)	
прямой межсегментный	1110 1010 : unsigned full offset, selector
косвенный межсегментный	1111 1111 : mod 101 r/m
LAHF — Load Flags into AH Register — загрузить флаги в AH	1001 1111
LAR — Load Access Rights Byte — загрузить байт прав доступа	
из регистра	0000 1111 : 0000 0010 : 11 reg1 reg2
из памяти	0000 1111 : 0000 0010 : mod reg r/m
LDS — Load Pointer to DS — загрузить указатель в DS Загрузка указателя в формате селектор:смещение, смещение размещается в reg, селектор — в DS.	1100 0101 : mod reg r/m
LEA — Load Effective Address — загрузить эффективный адрес	1000 1101 : mod reg r/m

Таблица П2.2 (продолжение)

Команда и формат	Код
LEAVE — High Level Procedure Exit — выход из процедуры ЯВУ (удаление стекового кадра)	1100 1001
LES — Load Pointer to ES — загрузка указателя в ES	1100 0100 : mod reg r/m
LFS — Load Pointer to FS — загрузка указателя в FS	0000 1111 : 1011 0100 : mod reg r/m
LGDT — Load Global Descriptor Table Register — загрузка GDTR	0000 1111 : 0000 0001 : mod 010 r/m
LGS — Load Pointer to GS — загрузка указателя в GS	0000 1111 : 1011 0101 : mod reg r/m
LIDT — Load Interrupt Descriptor Table Register — загрузка IDTR	0000 1111 : 0000 0001 : mod 011 r/m
LLDT — Load Local Descriptor Table Register — загрузка LDTR	
LDTR из регистра	0000 1111 : 0000 0000 : 11 010 reg
LDTR из памяти	0000 1111 : 0000 0000 : mod 010 r/m
LMSW — Load Machine Status Word — загрузка слова состояния машины	
из регистра	0000 1111 : 0000 0001 : 11 110 reg
из памяти	0000 1111 : 0000 0001 : mod 110 r/m
LOCK — Assert LOCK# Signal Prefix — префикс захвата шины	1111 0000
LODS/LODSB/LODSW/LODSD — Load String Operand — загрузка строкового операнда	1010 110w
LOOP — Loop Count — цикл со счетчиком [в CX]	1110 0010 : 8-bit displacement
LOOPZ/LOOPE — Loop Count while Zero/Equal — цикл с условием	1110 0001 : 8-bit displacement
LOOPNZ/LOOPNE — Loop Count while not Zero/Equal — цикл со счетчиком и условием	1110 0000 : 8-bit displacement
LSL — Load Segment Limit — загрузка ограничителя стека	
из регистра	0000 1111 : 0000 0011 : 11 reg1 reg2
из памяти	0000 1111 : 0000 0011 : mod reg r/m

Таблица П2.2 (продолжение)

Команда и формат	Код
LSS — Load Pointer to SS — загрузка указателя стека	0000 1111 : 1011 0010 : mod reg r/m
LTR — Load Task Register — загрузка регистра задачи	
из регистра	0000 1111 : 0000 0000 : 11 011 reg
из памяти	0000 1111 : 0000 0000 : mod 011 r/m
MOV — Move Data — перенос данных	
регистр1 в регистр2	1000 100w : 11 reg1 reg2
регистр2 в регистр1	1000 101w : 11 reg1 reg2
память в регистр	1000 101w : mod reg r/m
регистр в память	1000 100w : mod reg r/m
литерал в регистр	1100 011w : 11 000 reg : immediate data
литерал в регистр (альтернативное кодирование)	1011 w reg : immediate data
литерал в память	1100 011w : mod 000 r/m : immediate data
память в AL, AX или EAX	1010 000w : full displacement
AL, AX или EAX в память	1010 001w : full displacement
MOV — Move to/from Control Registers — перенос данных в управляющие регистры и обратно	
CR0 из регистра	0000 1111 : 0010 0010 : 11 000 reg
CR2 из регистра	0000 1111 : 0010 0010 : 11 010 reg
CR3 из регистра	0000 1111 : 0010 0010 : 11 011 reg
CR4 из регистра	0000 1111 : 0010 0010 : 11 100 reg
регистр из CR0-CR4	0000 1111 : 0010 0000 : 11 eee reg
MOV — Move to/from Debug Registers — перенос данных в управляющие регистры и обратно	
DR0-DR3 из регистра	0000 1111 : 0010 0011 : 11 eee reg
DR4-DR5 из регистра	0000 1111 : 0010 0011 : 11 eee reg
DR6-DR7 из регистра	0000 1111 : 0010 0011 : 11 eee reg

Таблица П2.2 (продолжение)

Команда и формат	Код
регистр из DR6-DR7	0000 1111 : 0010 0001 : 11 eee reg
регистр из DR4-DR5	0000 1111 : 0010 0001 : 11 eee reg
регистр из DR0-DR3	0000 1111 : 0010 0001 : 11 eee reg
MOV — Move to/from Segment Registers — перенос данных в сегментные регистры и обратно	
регистр в сегментный регистр	1000 1110 : 11 sreg3 reg
регистр в SS	1000 1110 : 11 sreg3 reg
память в сегментный регистр	1000 1110 : mod sreg3 r/m
память в SS	1000 1110 : mod sreg3 r/m
сегментный регистр в регистр	1000 1100 : 11 sreg3 reg
сегментный регистр в память	1000 1100 : mod sreg3 r/m
MOVS/MOVSB/MOVSW/MOVSD — Move Data from String to String — перенос данных память-память (строка-строка)	1010 010w
MOVSX — Move with Sign-Extend — перенос с расширением знаком	
регистр2 в регистр1	0000 1111 : 1011 111w : 11 reg1 reg2
память в регистр	0000 1111 : 1011 111w : mod reg r/m
MOVZX — Move with Zero-Extend — перенос с расширением нулями	
регистр2 с регистр1	0000 1111 : 1011 011w : 11 reg1 reg2
память с регистром	0000 1111 : 1011 011w : mod reg r/m
MUL — Unsigned Multiply — беззнаковое умножение	
AL, AX или EAX с регистром	1111 011w : 11 100 reg
AL, AX или EAX с памятью	1111 011w : mod 100 reg
NEG — Two's Complement Negation — обращение знака в двоично-дополнительном представлении	
регистр	1111 011w : 11 011 reg
память	1111 011w : mod 011 r/m

Таблица П2.2 (продолжение)

Команда и формат	Код
NOP — No Operation — пустая команда	1001 0000
NOT — One's Complement Negation — логическое отрицание	
регистр	1111 011w : 11 010 reg
память	1111 011w : mod 010 r/m
OR — Logical Inclusive OR — логическое ИЛИ	
регистр1 с регистр2	0000 100w : 11 reg1 reg2
регистр2 с регистр1	0000 101w : 11 reg1 reg2
память с регистром	0000 101w : mod reg r/m
регистр с памятью	0000 100w : mod reg r/m
литерал с регистром	1000 00sw : 11 001 reg : immediate data
литерал с AL, AX или EAX	0000 110w : immediate data
литерал с памятью	1000 00sw : mod 001 r/m : immediate data
OUT — Output to Port — вывод	
фиксированный порт	1110 011w : port number
переменный порт (номер порта в DX)	1110 111w
OUTS — Output to DX Port — вывод (порт в регистре DX)	0110 111w
POP — Pop a Word from the Stack — вытолкнуть слово из стека	
регистр	1000 1111 : 11 000 reg
регистр (альтернативное кодирование)	0101 1 reg
память	1000 1111 : mod 000 r/m
POP — Pop a Segment Register from the Stack — вытолкнуть стековый регистр (кроме CS)	
сегментные регистры DS, ES	000 sreg2 111
сегментный регистр SS	000 sreg2 111
сегментные регистры FS, GS	0000 1111: 10 sreg3 001

Таблица П2.2 (продолжение)

Команда и формат	Код
POPA/POPAD — Pop All General Registers — вытолкнуть из стека все регистры	0110 0001
POPF/POPFD — Pop Stack into FLAGS or EFLAGS Register — вытолкнуть регистр флагов из стека	1001 1101
PUSH — Push Operand onto the Stack — протолкнуть operand в стек	
регистр	1111 1111 : 11 110 reg
регистр (альтернативное кодирование)	0101 0 reg
память	1111 1111 : mod 110 r/m
литерал	0110 10s0 : immediate data
PUSH — Push Segment Register onto the Stack — протолкнуть в стек сегментный регистр	
сегментный регистр CS, DS, ES, SS	000 sreg2 110
сегментный регистр FS, GS	0000 1111: 10 sreg3 000
PUSHA/PUSHAD — Push All General Registers — протолкнуть в стек все регистры	0110 0000
PUSHF/PUSHFD — Push Flags Register onto the Stack — протолкнуть в стек регистр флагов	1001 1100
RCL — Rotate thru Carry Left — циклический сдвиг через перенос влево	
регистр на 1 бит	1101 000w : 11 010 reg
память на 1	1101 000w : mod 010 r/m
регистр на CL	1101 001w : 11 010 reg
память на CL	1101 001w : mod 010 r/m
регистр на литерал	1100 000w : 11 010 reg : imm8 data
память на литерал	1100 000w : mod 010 r/m : imm8 data
RCR — Rotate thru Carry Right — циклический сдвиг через перенос вправо	
регистр на 1	1101 000w : 11 011 reg

Таблица П2.2 (продолжение)

Команда и формат	Код
память на 1	1101 000w : mod 011 r/m
регистр на CL	1101 001w : 11 011 reg
память на CL	1101 001w : mod 011 r/m
регистр на литерал	1100 000w : 11 011 reg : imm8 data
память на литерал	1100 000w : mod 011 r/m : imm8 data
RDMSR — Read from Model-Specific Register — читать из MSR	0000 1111 : 0011 0010
RDPMC — Read Performance Monitoring Counters — считать счетчики монитора производительности	0000 1111 : 0011 0011
RDTSC — Read Time-Stamp Counter — считать таймер	0000 1111 : 0011 0001
REP INS — Input String — ввод строки [из порта ввода/вывода]	1111 0011 : 0110 110w
REP LODS — Load String — загрузка строки	1111 0011 : 1010 110w
REP MOVS — Move String — перемещение строки	1111 0011 : 1010 010w
REP OUTS — Output String — вывод строки [в порт ввода/вывода]	1111 0011 : 0110 111w
REP STOS — Store String — копирование строки	1111 0011 : 1010 101w
REPE CMPS — Compare String — сравнение строк	1111 0011 : 1010 011w
REPE SCAS — Scan String — сканирование строки	1111 0011 : 1010 111w
REPNE CMPS — Compare String — сравнение строк	1111 0010 : 1010 011w
REPNE SCAS — Scan String — сканирование строки	1111 0010 : 1010 111w
RET — Return from Procedure (to same segment) — внутрисегментный возврат	
без operandов	1100 0011
добавить литерал к SP/ESP (очистка стека от параметров)	1100 0010 : 16-bit displacement

Таблица П2.2 (продолжение)

Команда и формат	Код
RET — Return from Procedure (to other segment) — межсегментный возврат	
без операндов	1100 1011
добавить литерал к SP/ESP	1100 1010 : 16-bit displacement
ROL — Rotate Left — циклический сдвиг влево	
регистр на 1	1101 000w : 11 000 reg
память на 1	1101 000w : mod 000 r/m
регистр на CL	1101 001w : 11 000 reg
память на CL	1101 001w : mod 000 r/m
регистр на литерал	1100 000w : 11 000 reg : imm8 data
память на литерал	1100 000w : mod 000 r/m : imm8 data
ROR — Rotate Right — циклический сдвиг вправо	
регистр на 1	1101 000w : 11 001 reg
память на 1	1101 000w : mod 001 r/m
регистр на CL	1101 001w : 11 001 reg
память на CL	1101 001w : mod 001 r/m
регистр на литерал	1100 000w : 11 001 reg : imm8 data
память на литерал	1100 000w : mod 001 r/m : imm8 data
RSM — Resume from System Management Mode — выйти из режима управления системой	0000 1111 : 1010 1010
SAHF — Store AH into Flags — сохранить AH в регистре флагов	1001 1110
SAL — Shift Arithmetic Left — арифметический сдвиг влево	same instruction as SHL
SAR — Shift Arithmetic Right — арифметический [с расширением знаком] сдвиг вправо	
регистр на 1	1101 000w : 11 111 reg
память на 1	1101 000w : mod 111 r/m
регистр на CL	1101 001w : 11 111 reg

Таблица П2.2 (продолжение)

Команда и формат	Код
память на CL	1101 001w : mod 111 r/m
регистр на литерал	1100 000w : 11 111 reg : imm8 data
память на литерал	1100 000w : mod 111 r/m : imm8 data
SBB — Integer Subtraction with Borrow — вычитание с переносом	
регистр1 с регистр2	0001 100w : 11 reg1 reg2
регистр2 с регистр1	0001 101w : 11 reg1 reg2
память с регистром	0001 101w : mod reg r/m
регистр с памятью	0001 100w : mod reg r/m
литерал с регистром	1000 00sw : 11 011 reg : immediate data
литерал с AL, AX или EAX	0001 110w : immediate data
литерал с памятью	1000 00sw : mod 011 r/m : immediate data
SCAS/SCASB/SCASW/SCASD — Scan String — сканирование строки	1010 111w
SETcc — Byte Set on Condition — установить байт по условию	
регистр	0000 1111 : 1001 tttt : 11 000 reg
память	0000 1111 : 1001 tttt : mod 000 r/m
SGDT — Store Global Descriptor Table Register — сохранить GDTR	0000 1111 : 0000 0001 : modA000 r/m
SHL — Shift Left — сдвиг влево	
регистр на 1	1101 000w : 11 100 reg
память на 1	1101 000w : mod 100 r/m
регистр на CL	1101 001w : 11 100 reg
память на CL	1101 001w : mod 100 r/m
регистр на литерал	1100 000w : 11 100 reg : imm8 data
память на литерал	1100 000w : mod 100 r/m : imm8 data
SHLD — Double Precision Shift Left — сдвиг влево с двойной точностью	
регистр на литерал	0000 1111 : 1010 0100 : 11 reg2 reg1 : imm8

Таблица П2.2 (продолжение)

Команда и формат	Код
память на литерал	0000 1111 : 1010 0100 : mod reg r/m : imm8
регистр на CL	0000 1111 : 1010 0101 : 11 reg2 reg1
память на CL	0000 1111 : 1010 0101 : mod reg r/m
SHR — Shift Right — сдвиг вправо	
регистр на 1	1101 000w : 11 101 reg
память на 1	1101 000w : mod 101 r/m
регистр на CL	1101 001w : 11 101 reg
память на CL	1101 001w : mod 101 r/m
регистр на литерал	1100 000w : 11 101 reg : imm8 data
память на литерал	1100 000w : mod 101 r/m : imm8 data
SHRD — Double Precision Shift Right — сдвиг вправо с двойной точностью	
регистр на литерал	0000 1111 : 1010 1100 : 11 reg2 reg1 : imm8
память на литерал	0000 1111 : 1010 1100 : mod reg r/m : imm8
регистр на CL	0000 1111 : 1010 1101 : 11 reg2 reg1
память на CL	0000 1111 : 1010 1101 : mod reg r/m
SIDT — Store Interrupt Descriptor Table Register — сохранить IDTR	0000 1111 : 0000 0001 : modA001 r/m
SLDT — Store Local Descriptor Table Register — сохранить LDTR	
в регистр	0000 1111 : 0000 0000 : 11 000 reg
в память	0000 1111 : 0000 0000 : mod 000 r/m
SMSW — Store Machine Status Word — сохранить слово состояния машины	
в регистр	0000 1111 : 0000 0001 : 11 100 reg
в память	0000 1111 : 0000 0001 : mod 100 r/m
STC — Set Carry Flag — установить флаг переноса	1111 1001
STD — Set Direction Flag — установить флаг направления	1111 1101

Таблица П2.2 (продолжение)

Команда и формат	Код
STI — Set Interrupt Flag — установить флаг [разрешения] прерывания	1111 1011
STOS/STOSB/STOSW/STOSD — Store String Data — сохранить строковые данные	1010 101w
STR — Store Task Register — сохранить регистр задачи	
в регистр	0000 1111 : 0000 0000 : 11 001 reg
в память	0000 1111 : 0000 0000 : mod 001 r/m
SUB — Integer Subtraction — вычитание	
регистр1 с регистр2	0010 100w : 11 reg1 reg2
регистр2 с регистр1	0010 101w : 11 reg1 reg2
память с регистром	0010 101w : mod reg r/m
регистр с памятью	0010 100w : mod reg r/m
литерал с регистром	1000 00sw : 11 101 reg : immediate data
литерал с AL, AX или EAX	0010 110w : immediate data
литерал с памятью	1000 00sw : mod 101 r/m : immediate data
TEST — Logical Compare — логическое сравнение	
регистр1 с регистр2	1000 010w : 11 reg1 reg2
память с регистром	1000 010w : mod reg r/m
литерал с регистром	1111 011w : 11 000 reg : immediate data
литерал и AL, AX или EAX	1010 100w : immediate data
литерал с памятью	1111 011w : mod 000 r/m : immediate data
UD2 — Undefined instruction — неопределенная команда	0000 1111 : 0000 1011
VERR — Verify a Segment for Reading — проверить доступность сегмента для чтения	
регистр	0000 1111 : 0000 0000 : 11 100 reg
память	0000 1111 : 0000 0000 : mod 100 r/m

Таблица П2.2 (продолжение)

Команда и формат	Код
VERW — Verify a Segment for Writing — проверить доступность сегмента для записи	
регистр	0000 1111 : 0000 0000 : 11 101 reg
память	0000 1111 : 0000 0000 : mod 101 r/m
WAIT — Wait — ожидание	1001 1011
WBINVD — Writeback and Invalidate Data Cache — [фоновая] запись и инвалидация кэша данных	0000 1111 : 0000 1001
WRMSR — Write to Model-Specific Register — запись в управляющий регистр процессора	0000 1111 : 0011 0000
XADD — Exchange and Add — обмен со сложением	
регистр1, регистр2	0000 1111 : 1100 000w : 11 reg2 reg1
память, регистр	0000 1111 : 1100 000w : mod reg r/m
XCHG — Exchange — обменять регистр с памятью	
регистр1 с регистр2	1000 011w : 11 reg1 reg2
AX или EAX с регистром	1001 0 reg
память с регистром	1000 011w : mod reg r/m
XLAT/XLATB — Table Look-up Translation — трансляция по таблице	1101 0111
XOR — Logical Exclusive OR — побитовое иллюстратор ИЛИ	
регистр1 с регистр2	0011 000w : 11 reg1 reg2
регистр2 с регистр1	0011 001w : 11 reg1 reg2
память с регистром	0011 001w : mod reg1 r/m
регистр с памятью	0011 000w : mod reg r/m
литерал с регистром	1000 00sw : 11 110 reg : immediate data
литерал с AL, AX или EAX	0011 010w : immediate data
литерал с памятью	1000 00sw : mod 110 r/m : immediate data

Таблица П2.2 (окончание)

Коды префиксов	
Размер адреса	0110 0111
LOCK — блокировка шины	1111 0000
Размер операнда	0110 0110
Замена сегмента на CS	0010 1110
Замена сегмента на DS	0011 1110
Замена сегмента на ES	0010 0110
Замена сегмента на FS	0110 0100
Замена сегмента на GS	0110 0101
Замена сегмента на SS	0011 0110

Таблица П2.3. Значения битового поля *tttn* в командах условного перехода и условной установки байта

tttn	Мнемоника	Условие
0000	O	Overflow (переполнение)
0001	NO	No overflow (нет переполнения)
0010	B, NAE	Below, Not above or equal (ниже, не выше или равно)
0011	NB, AE	Not below, Above or equal (не ниже, выше или равно)
0100	E, Z	Equal, Zero (равно, ноль)
0101	NE, NZ	Not equal, Not zero (не равно, не ноль)
0110	BE, NA	Below or equal, Not above (ниже или равно, не выше)
0111	NBE, A	Not below or equal, Above (не ниже или равно, выше)
1000	S	Sign (знак, отрицательное число)
1001	NS	Not sign (не знак, положительное число)
1010	P, PE	Parity, Parity Even (четность, четная четность)
1011	NP, PO	Not parity, Parity Odd (не четность, нечетная четность)
1100	L, NGE	Less than, Not greater than or equal to (меньше, не больше или равно)
1101	NL, GE	Not less than, Greater than or equal to (не меньше, больше или равно)

Таблица П2.3 (окончание)

tttn	Мнемоника	Условие
1110	LE, NG	Less than or equal to, Not greater than (меньше или равно, не больше)
1111	NLE, G	Not less than or equal to, Greater than (не меньше или равно, больше)

Таблица П2.4. Мнемоники и коды команд сопроцессора
(операции с плавающей точкой)

Команда и формат	Код операции
F2XM1 — вычислить $2^{ST(0)} - 1$	11011 001 : 1111 0000
FABS — абсолютное значение	11011 001 : 1110 0001
FADD — Add — сложить	
ST(0) \leftarrow ST(0) +32бит память	11011 000 : mod 000 r/m
ST(0) \leftarrow ST(0) +64бит память	11011 100 : mod 000 r/m
ST(d) \leftarrow ST(0) +ST(i)	11011 d00 : 11 000 ST(i)
FADDP — Add and Pop — сложить и вытолкнуть [из стека]	
ST(0) \leftarrow ST(0) +ST(i)	11011 110 : 11 000 ST(i)
FBLD — Load Binary Coded Decimal — загрузить двоично-десятичное число	11011 111 : mod 100 r/m
FBSTP — Store Binary Coded Decimal and Pop — сохранить двоично-десятичное число и вытолкнуть из стека	11011 111 : mod 110 r/m
FCHS — Change Sign — обратить знак	11011 001 : 1110 0000
FCLEX — Clear Exceptions — сбросить исключения	11011 011 : 1110 0010
FCOM — Compare Real — сравнить (действительное число)	
32бит память	11011 000 : mod 010 r/m
64бит память	11011 100 : mod 010 r/m
ST(i)	11011 000 : 11 010 ST(i)
FCOMP — Compare Real and Pop — сравнить (действительное число) и вытолкнуть из стека	
32бит память	11011 000 : mod 011 r/m

Таблица П2.4 (продолжение)

Команда и формат	Код операции
64бит память	11011 100 : mod 011 r/m
ST(i)	11011 000 : 11 011 ST(i)
FCOMPP — Compare Real and Pop Twice — сравнить и вытолкнуть из стека дважды	11011 110 : 11 011 001
FCOMIP — Compare Real, Set EFLAGS, and Pop — сравнить, установить EFLAGS и вытолкнуть из стека	11011 111 : 11 110 ST(i)
FCOS — Cosine of ST(0) — косинус	11011 001 : 1111 1111
FDECSTP — Decrement Stack-Top Pointer — уменьшил указатель стека (вытолкнуть с отбрасыванием значений)	11011 001 : 1111 0110
FDIV — Divide — деление	
ST(0) ← ST(0) ÷ 32бит память	11011 000 : mod 110 r/m
ST(0) ← ST(0) ÷ 64бит память	11011 100 : mod 110 r/m
ST(d) ← ST(0) ÷ ST(i)	11011 d00 : 1111 R ST(i)
FDIVP — Divide and Pop — деление с выталкиванием	
ST(0) ← ST(0) ÷ ST(i)	11011 110 : 1111 1 ST(i)
FDIVR — Reverse Divide — деление с обратным порядком operandов	
ST(0) ← 32бит память ÷ ST(0)	11011 000 : mod 111 r/m
ST(0) ← 64бит память ÷ ST(0)	11011 100 : mod 111 r/m
ST(d) ← ST(i) ÷ ST(0)	11011 d00 : 1111 R ST(i)
FDIVRP — Reverse Divide and Pop — деление с обратным порядком operandов и выталкивание из стека	
ST(0) ← ST(i) ÷ ST(0)	11011 110 : 1111 0 ST(i)
FFREE — Free ST(i) Register — освободить регистр	11011 101 : 1100 0 ST(i)
FIADD — Add Integer — сложить с целым числом	
ST(0) ← ST(0) + 16бит память	11011 110 : mod 000 r/m
ST(0) ← ST(0) + 32бит память	11011 010 : mod 000 r/m
FICOM — Compare Integer — сравнить с целым числом	
16бит память	11011 110 : mod 010 r/m
32бит память	11011 010 : mod 010 r/m

Таблица П2.4 (продолжение)

Команда и формат	Код операции
FICOMP — Compare Integer and Pop — сравнить с целым числом и вытолкнуть из стека	
16бит память	11011 110 : mod 011 r/m
32бит память	11011 010 : mod 011 r/m
FIDIV — деление на целое число	
ST(0) ← ST(0) ÷ 16бит память	11011 110 : mod 110 r/m
ST(0) ← ST(0) ÷ 32бит память	11011 010 : mod 110 r/m
FIDIVR	
ST(0) ← 16бит память ÷ ST(0)	11011 110 : mod 111 r/m
ST(0) ← 32бит память ÷ ST(0)	11011 010 : mod 111 r/m
FILD — Load Integer — загрузка целого числа	
16бит память	11011 111 : mod 000 r/m
32бит память	11011 011 : mod 000 r/m
64бит память	11011 111 : mod 101 r/m
FIMUL	
ST(0) ← ST(0) × 16бит память	11011 110 : mod 001 r/m
ST(0) ← ST(0) × 32бит память	11011 010 : mod 001 r/m
FINCSTP — Increment Stack Pointer — увеличить указатель стека	11011 001 : 1111 0111
FINIT — Initialize Floating-Point Unit — инициализация сопроцессора	
FIST — Store Integer — сохранить целое число	
16бит память	11011 111 : mod 010 r/m
32бит память	11011 011 : mod 010 r/m
FISTP — Store Integer and Pop — вытолкнуть из стека и сохранить целое число	
16бит память	11011 111 : mod 011 r/m
32бит память	11011 011 : mod 011 r/m
64бит память	11011 111 : mod 111 r/m
FISUB	
ST(0) ← ST(0) — 16бит память	11011 110 : mod 100 r/m
ST(0) ← ST(0) — 32бит память	11011 010 : mod 100 r/m

Таблица П2.4 (продолжение)

Команда и формат	Код операции
FISUBR	
ST(0) ← 16бит память – ST(0)	11011 110 : mod 101 r/m
ST(0) ← 32бит память – ST(0)	11011 010 : mod 101 r/m
FLD — Load Real — загрузить действительное число	
32бит память	11011 001 : mod 000 r/m
64бит память	11011 101 : mod 000 r/m
80бит память	11011 011 : mod 101 r/m
ST(i)	11011.001 : 11 000 ST(i)
FLD1 — Load +1.0 into ST(0) — загрузить 1.0 в ST (0)	11011 001 : 1110 1000
FLDCW — Load Control Word — загрузить управляющее слово	11011 001 : mod 101 r/m
FLDENV — Load FPU Environment — загрузить контекст сопроцессора	11011 001 : mod 100 r/m
FLDL2E — загрузить $\log_2(\epsilon)$ в ST(0)	11011 001 : 1110 1010
FLDL2T — загрузить $\log_2(10)$ в ST(0)	11011 001 : 1110 1001
FLDLG2 — загрузить $\log_{10}(2)$ в ST(0)	11011 001 : 1110 1100
FLDLN2 — загрузить $\log_e(2)$ в ST(0)	11011 001 : 1110 1101
FLDPI — загрузить π в ST(0)	11011 001 : 1110 1011
FLDZ — загрузить +0.0 в ST(0)	11011 001 : 1110 1110
FMUL — Multiply — умножение	
ST(0) ← ST(0) × 32бит память	11011 000 : mod 001 r/m
ST(0) ← ST(0) × 64бит память	11011 100 : mod 001 r/m
ST(d) ← ST(0) × ST(i)	11011 d00 : 1100 1 ST(i)
FMULP — Multiply — умножение с выталкиванием	
ST(i) ← ST(0) × ST(i)	11011 110 : 1100 1 ST(i)
FNOP — No Operation — пустая команда	11011 001 : 1101 0000
FPATAN — Partial Arctangent — дробный арктангенс	11011 001 : 1111 0011
FPREM — Partial Remainder — остаток	11011 001 : 1111 1000
FPREM1 — Partial Remainder (IEEE) — остаток в формате IEEE	11011 001 : 1111 0101

Таблица П2.4 (продолжение)

Команда и формат	Код операции
FPTAN — Partial Tangent — дробный тангенс	11011 001 : 1111 0010
FRNDINT — Round to Integer — округлить к целому	11011 001 : 1111 1100
FRSTOR — Restore FPU State — восстановить состояние сопроцессора	11011 101 : mod 100 r/m
FSAVE — Store FPU State — сохранить состояние сопроцессора	11011 101 : mod 110 r/m
FSCALE — Scale — масштабирование	11011 001 : 1111 1101
FSIN — Sine — синус	11011 001 : 1111 1110
FSINCOS — Sine and Cosine — синус и косинус	11011 001 : 1111 1011
FSQRT — Square Root — квадратный корень	11011 001 : 1111 1010
FST — Store Real — сохранить целое число	
32бит память	11011 001 : mod 010 r/m
64бит память	11011 101 : mod 010 r/m
ST(i)	11011 101 : 11 010 ST(i)
FSTCW — Store Control Word — сохранить управляющее слово	11011 001 : mod 111 r/m
FSTENV — Store FPU Environment — сохранить контекст сопроцессора	11011 001 : mod 110 r/m
FSTP — Store Real and Pop — сохранить действительное число и вытолкнуть из стека	
32бит память	11011 001 : mod 011 r/m
64бит память	11011 101 : mod 011 r/m
80бит память	11011 011 : mod 111 r/m
ST(i)	11011 101 : 11 011 ST(i)
FSTSW — Store Status Word into AX — сохранить слово состояния в AX	11011 111 : 1110 0000
FSTSW — Store Status Word into Memory — сохранить слово состояния в памяти	11011 101 : mod 111 r/m
FSUB — Subtract — вычитание	
ST(0) ← ST(0) — 32бит память	11011 000 : mod 100 r/m
ST(0) ← ST(0) — 64бит память	11011 100 : mod 100 r/m
ST(d) ← ST(0) — ST(i)	11011 d00 : 1110 R ST(i)

Таблица П2.4 (продолжение)

Команда и формат	Код операции
FSUBP — Subtract and Pop — вычитание с выталкиванием из стека	
ST(0) ← ST(0) — ST(i)	11011 110 : 1110 1 ST(i)
FSUBR — Reverse Subtract — вычитание с обратным порядком operandов	
ST(0) ← 32бит память — ST(0)	11011 000 : mod 101 r/m
ST(0) ← 64бит память — ST(0)	11011 100 : mod 101 r/m
ST(d) ← ST(i) — ST(0)	11011 d00 : 1110 R ST(i)
FSUBRP — Reverse Subtract and Pop — вычитание с обратным порядком operandов и выталкиванием	
ST(i) ← ST(i) — ST(0)	11011 110 : 1110 0 ST(i)
FTST — Test — проверка (сравнение с 0)	11011 001 : 1110 0100
FUCOM — Unordered Compare Real — неупорядоченное сравнение	11011 101 : 1110 0 ST(i)
FUCOMP — Unordered Compare Real and Pop — неупорядоченное сравнение с выталкиванием	11011 101 : 1110 1 ST(i)
FUCOMPP — Unordered Compare Real and Pop Twice — неупорядоченное сравнение с двойным выталкиванием	11011 010 : 1110 1001
FUCOMI — Unordered Compare Real and Set EFLAGS — неупорядоченное сравнение с установкой EFLAGS	11011 011 : 11 101 ST(i)
FUCOMIP — Unordered Compare Real, Set EFLAGS, and Pop — неупорядоченное сравнение с установкой EFLAGS и выталкиванием	11011 111 : 11 101 ST(i)
FXAM — Examine — проверка знака	11011 001 : 1110 0101
FXCH — Exchange ST(0) and ST(i) — обмен	11011 001 : 1100 1 ST(i)
FTRACT — Extract Exponent and Significand — извлечь порядок и мантиссу	11011 001 : 1111 0100
FYL2X — ST(1) × log ₂ (ST(0))	11011 001 : 1111 0001
FYL2XP1 — ST(1) × log ₂ (ST(0) + 1.0)	11011 001 : 1111 1001
FWAIT — Wait until FPU Ready — ожидание сопроцессора	1001 1011

d — направление операции.

d=0 → ST(0):=ST(0) op ST(i).

d=1 → ST(i):=ST(i) op ST(0).

П2.4. Язык ассемблера

Для x86 используется два типа ассемблеров с существенно различным синтаксисом. Первый тип — ассемблеры с синтаксисом Intel. Этот синтаксис с небольшими вариациями используют такие популярные продукты, как Microsoft Assembler и Turbo Assembler компании Borland.

Второй тип — ассемблеры, использующие синтаксис AT&T. Такие ассемблеры первоначально применялись в системах семейства Unix. Наиболее популярный продукт, поддерживающий этот синтаксис, — GNU Assembler (gas), который доступен не только для систем семейства Unix, но и для DOS, OS/2 и Win32.

Большинство примеров кода в этой книге приведено в синтаксисе AT&T, поэтому далее в этом разделе будет рассматриваться именно он.

Читателям, знакомым с синтаксисом Intel, необходимо первым делом сообщить, что masm/tasm и ассемблеры с синтаксисом AT&T используют разный порядок operandов. В синтаксисе Intel запись `mov eax ebx` обозначает `eax:=ebx`. В AT&T команда `mov %eax, %ebx` обозначает `ebx:=eax`.

Другое отличие состоит в том, что ассемблер Intel определяет тип команды (байтовая, 16-битная или 32-битная) по размеру operandов. Для регистров тип операнда однозначно определяется по имени: `eax` обозначает 32-битовый регистр, `ax` — 16-битовый, `a1` — байтовый. Для объектов в памяти тип необходимо указывать неявно: `mov ax word ptr 20[ebp]`.

Напротив, ассемблеры AT&T используют суффикс, добавляемый после мемоники команды, и обозначающий тип operandов: `b`, `w` или `l` (byte, word, long — соответственно, 8-битный, 16-битный и 32-битный). Несоответствие суффикса и типа operandса, например команда `movw $20, %eax`, приводит к синтаксической ошибке. В действительности, если суффикс не указать, ассемблер пытается догадаться самостоятельно. Поскольку основная масса команд требует в качестве одного из operandов регистр, в большинстве случаев это возможно. Однако код, порожденный компилятором, всегда содержит суффиксы типа.

При таком подходе определенную сложность представляет запись команд преобразования типа, `movsx` и `movzx` (`MOVe [with] Sign Extend` и `MOVe [with] Zero Extend` соответственно, т. е. пересылка с расширением знаком и с расширением нулями). Ассемблер Intel может определять требуемый тип преобразования неявно по типам operandов: так, не представляет сложности понять, что означает команда `movzx eax byte ptr [edx]`. В синтаксисе AT&T возможные варианты этих команд обозначаются мемониками `movsbl`, `movsbw`, `movswl`, `movzb1`, `movzbw` и `movzwl`.

П2.4.1. Синтаксис

Описание синтаксиса GNU Assembler размещено на сайте [[www.gnu.org gas-2.9.1](http://www.gnu.org/gas-2.9.1)]. Описание особенностей gas, относящихся к x86, находится на странице [[www.gnu.org gas-2.9.1](http://www.gnu.org/gas-2.9.1) 16] Синтаксис языка ассемблера очень прост. Оператор занимает одну строку и может быть пустым либо содержать мнемонику команды или директиву. То и другое может иметь один или несколько операндов, в зависимости от типа команды и директивы. Имена директив начинаются с точки, мнемоники команд начинаются с алфавитного символа.

Оператор (в том числе и пустой) может иметь метку или не иметь ее. Метка начинается с первого символа строки и заканчивается двоеточием. Если строка начинается с пробела или символа табуляции, она не имеет метки.

Комментарии бывают двух типов. Строчные комментарии начинаются с символа ! и заканчиваются концом строки. Многострочные комментарии начинаются с пары символов /* и заканчиваются парой символов */. Допускаются многострочные комментарии, занимающие одну строку.

Везде, где синтаксис допускает пробел, может стоять несколько пробелов, символов табуляции или коментариев. Несколько последовательных пробелов и табуляций эквивалентны одному пробелу.

П2.4.2. Символы

Символы — это ключевое понятие. Программисты используют символы, чтобы именовать предметы. Сборщик использует символы, чтобы собирать. Отладчик использует символы, чтобы отлаживать.

Д. Элспер, Дж. Фенласон и др.

Символ — это именованное значение. Символы и арифметические выражения с их участием можно использовать в операндах команд и директив. Полученные таким образом значения будут включаться в порождаемый ассемблером код в качестве адресных полей команд, литералов, стартовых значений переменных и т. д.

Символы существуют только на этапе компиляции, а некоторые символы — также на этапе сборки программы. При сборке готовой программы символы отбрасываются.

Дотошный читатель должен отметить, что при динамической сборке и при символьной отладке части таблицы символов сохраняются и во время исполнения. Тут надо дополнительно отметить, что динамически собираемую программу трудно назвать готовой, ведь сборка ее еще не завершена. В свою очередь, символьный отладчик, конечно же, активно пользуется таблицей

символов отлаживаемой программы — но для самой программы эти символы во время исполнения не имеют значения.

П2.4.2.1. Имена символов

Имя символа формируется по правилам, похожим на правила формирования идентификаторов в языках высокого уровня. Имена символов могут начинаться с алфавитных символов кодировки ASCII, точки (ASCII '.') или подчерка (ASCII '_'). Имя может содержать алфавитные символы, цифры и подчерки. На некоторых системах имя также может содержать символ ASCII '\$'. Поскольку ассемблер генерирует таблицы символов для внешних по отношению к нему программ (статического редактора связей, загрузчика, редактора связей времени исполнения и отладчиков), ассемблер должен соответствовать требованиям, которые эти программы накладывают.

Имя чувствительно к регистру, т. е. `Function` и `function` — это два разных имени.

Кроме именованных символов, программист может определить до десяти локальных символов, обозначаемых цифрами от 0 до 9. Эти символы могут переопределяться многократно; при ссылке на такой символ используется последнее по порядку из предыдущих (по тексту программы) определение или первое из последующих. При ссылке на такой символ надо указывать, будет ли это ссылка назад (тогда необходимо писать `0b`) или вперед (`0f`).

Символы делятся на внутренние (*internal*) и глобальные (*global*). Глобальные символы также называют внешними (*external*). Внутренние символы доступны только в пределах одного файла ассемблерного текста. Внешние символы сохраняются в таблицах глобальных символов порожденного объектного файла и доступны при сборке всем остальным модулям, включаемым в собираемую программу.

Допускается использование одноименных внутренних символов в двух разных ассемблерных файлах. Напротив, определение двух одноименных внешних символов в разных файлах приведет к ошибке при сборке.

По умолчанию символы, определяемые в ассемблерном тексте, считаются внутренними. Чтобы сделать символ внешним, необходимо указать его операндом директивы `.globl` или `.global` (это две разные формы записи одной и той же директивы, введенные для совместимости с другими ассемблерами).

Напротив, все символы, используемые, но не определяемые в текущем файле, считаются внешними. Директива `.extern` считается допустимой (для совместимости с другими ассемблерами), но игнорируется.

Некоторые форматы объектных файлов, например ELF, допускают "слабые" (*weak*) определения символов. Если при сборке будут обнаружены один или несколько "слабых" символов и один одноименный с ними "сильный", то

компоновщик проигнорирует "слабые" определения символов и будет использовать "сильное" определение. Напротив, если будет обнаружено одно "слабое" определение и ни одного сильного, то будет использоваться "слабое" определение.

П2.4.2.2. Значения символов

Присваивание символу значения — это определение символа. Значения символов могут определяться двумя основными способами.

Во-первых, можно использовать имя символа в качестве метки — тогда символ приобретет значение, равное адресу текущего оператора или, точнее, адресу, который будет соответствовать адресу кода или данных, которые были порождены этим оператором. Если метка указывает на оператор, не порождающий кода (таковы пустые операторы и многие директивы), то ее значение будет соответствовать значению первого из последующих операторов, порождающих код.

Во-вторых, символу можно назначить значение директивой `.set` или некоторыми другими директивами.

Определение символа не обязано предшествовать его первому использованию. Для генерации кода в таких условиях используется второй проход. Впрочем, некоторые применения символов — например, выражения в условиях директив условной компиляции — все-таки требуют, чтобы символы были определены до использования.

Кроме значения, символ имеет тип и может иметь другие атрибуты. Некоторые форматы объектных файлов, например COFF и ELF, допускают снабжение символов произвольными именованными атрибутами.

Тип символа может быть абсолютным, перемещаемым или неопределенным. Независимо от типа, значение символа является 32-битовым целым числом при асSEMBЛИРОвании кода для 32-разрядных процессоров и 64-разрядным — при асSEMBЛИРОвании кода для 64-разрядных процессоров.

Абсолютные значения — это значения, которые могут быть определены при компиляции. Примеры абсолютных значений — это 0, 255, 0x2128506.

Перемещаемые значения — это значения, которые зависят от положения генерируемого кода в собранной программе, т. е. значения, которые определяются только в момент сборки или даже загрузки программы. Адрес метки как правило является перемещаемым значением. Перемещаемое значение всегда принадлежит к определенной секции (понятие секции объясняется в следующем разделе).

Разность двух перемещаемых значений, принадлежащих к одной секции, представляет собой абсолютное значение. Сумма абсолютного и перемещае-

мого значений представляет собой перемещаемое значение, принадлежащее к той же секции. Складывать перемещаемые значения, особенно принадлежащие к разным секциям, нельзя.

Если в момент компиляции ссылки на символ ассемблер еще не видел его определения, он неявно определяет такой символ как неопределенный. Если до конца файла определение не будет найдено, ассемблер считает символ внешней ссылкой и надеется, что его определение будет найдено компоновщиком в каком-то другом файле.

Существует специальный символ . (точка). Значение этого символа всегда равно текущему адресу, т. е. адресу, по которому будет размещен код, порожденный следующей командой или директивой, генерирующей код. Точке (.) можно присваивать значения директивой .set и специальной директивой .org.

П2.4.3. Секции

Секции и субсекции — это средство группировки кода, переменных и символов. Секции имеют имена, а субсекции — имена родительских секций и номера.

При сборке линкер размещает все элементы кода и данных, принадлежащие к одной секции, но определенные в разных файлах, в последовательных адресах памяти (рис. П2.13). В действительности все несколько сложнее — при определенных обстоятельствах секция может занимать несмежные области памяти. Более корректное (но неявное и менее понятное) определение секции состоит в том, что секция имеет определенный адрес, назначаемый ей при сборке с учетом размеров и расположения остальных секций. Все перемещаемые символы, принадлежащие к этой секции, получают значения, которые равны сумме начального адреса секции и смещения символа от начала секции. Поэтому секция после сборки всегда перемещается как целое.

Принадлежность кода и символов к секции определяется очень просто. Программист указывает в тексте директиву начала секции. Весь код и все метки, размещенные после этой директивы, вплоть до другой директивы начала секции или до конца файла, считаются принадлежащими к этой секции.

GNU assembler поддерживает четыре предопределенные секции: `text`, `data`, `bss` и `absolute`. С учетом существования последней секции, можно сказать, что абсолютные значения тоже принадлежат к определенной секции; в отличие от "нормальных" секций эта секция никогда не перемещается, а загрузчик не выделяет под нее памяти.

При сборке программ в формате `a.out` поддерживаются только эти секции. При сборке программ в других форматах, таких как COFF и ELF, допускается создание дополнительных секций с практически произвольными именами.

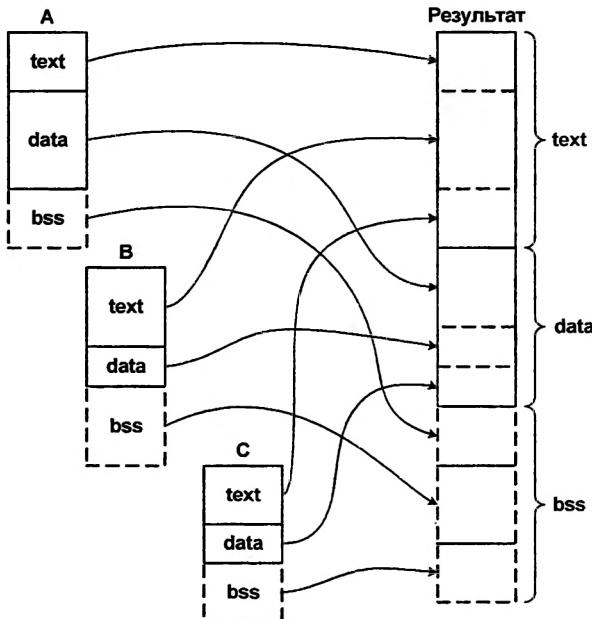


Рис. П2.13. Сборка программы из нескольких модулей с использованием секций

Директива начала секции — это директива `.section name, [attrib]`. Список и формат записи атрибутов зависят от формата исполняемого модуля и обычно позволяют указать права доступа к секции — доступна ли она для чтения, записи и исполнения. Также обычно существует атрибут, позволяющий указать, что секция содержит неинициализированные данные и должна создаваться при загрузке программы путем выделения задаче страниц, заполненных нулями.

Для начала секций `text` и `data` можно использовать специальные директивы `.text` и `.data`. По умолчанию, секция `text` доступна только для чтения и исполнения, секция `data` — для записи и чтения, секция `bss` — для записи и чтения. При этом память под секцию `bss` выделяется при загрузке и инициализируется нулями. Многие линкеры не позволяют изменять атрибуты этих секций.

Директива начала субсекции имеет вид `.section name, subsection_number`. Допустимы номера от 0 до 8192. По умолчанию, весь код, создаваемый в секции без указания субсекции, считается принадлежащим к субсекции 0.

П2.4.4. Команды

Команда состоит из мнемоники и операндов. Мнемоники команд в целом совпадают с приведенными в табл. П2.2 и П2.3, за исключением команд пре-

образования типа целых чисел (*movsx* и *movzx*). После мнемоники команды без пробела может (и, если по operandам команды нельзя однозначно установить их тип, должен) указываться суффикс, описывающий размер целочисленного операнда:

- *b* — для байтовых операций (т. е. операций над восьмибитными значениями);
- *w* — для операций над двухбайтовыми полусловами;
- *l* — для операций над четырехбайтовыми полными словами.

Количество и типы operandов зависят от команды. Operand может представлять собой литерал, регистр общего назначения, сегментный или служебный регистр, описание режима адресации или метку. Если команда имеет несколько operandов, они перечисляются через запятую.

Литерал записывается как *\$value*, где *value* может представлять собой числовое значение, либо имя символа, либо арифметическое выражение, возможно с использованием имен символов.

Регистр записывается как *%name*, где *name* — имя регистра. Имена регистров (как общего назначения, так и сегментных и служебных) соответствуют номенклатуре Intel.

Способ записи режимов адресации прост, регулярен и похож на способ записи режимов адресации в ассемблерах компании DEC для процессоров PDP-11 и VAX.

Регистровый режим адресации обозначается *%reg*.

Косвенно-регистровый режим обозначается (*%reg*), где *reg* — имя регистра общего назначения.

Косвенно-регистровый режим со смещением обозначается *offset(%reg)*, где *offset* — арифметическое выражение, возможно с использованием символов.

Базово-индексный режим без смещения обозначается как (*%base, %index, scale*), где *base* — базовый регистр, *index* — индексный регистр, *scale* — масштаб в диапазоне от 0 до 3.

Базово-индексный режим со смещением обозначается как *offset(%base, %index, scale)*.

Команды перехода и вызова по фиксированным адресам используют в качестве операнда метки или выражения без каких-либо дополнительных символов.

Разобрав команду, ассемблер генерирует соответствующий машинный код и сдвигает текущую позицию в коде (значение спецсимвола *.*) на размер этой

команды в байтах. Если операнды команды содержали перемещаемые или неопределенные символы, генерируются записи в таблице перемещения и/или таблице внешних ссылок. Окончательная генерация адресных полей или литералов этой команды произойдет при сборке, а при создании относительного или динамически собираемого загрузочного модуля — уже только при загрузке.

П2.4.5. Директивы

Директивы ассемблера отличаются большим разнообразием. Некоторые директивы приводят к генерации кода или инициализированных данных. Некоторые другие директивы влияют на значения символов. Вместо того чтобы пытаться описать директивы общим коротким текстом, проще перечислить наиболее важные из директив.

В этой книге я не ставлю перед собой цели дать полное описание GNU assembler; я привожу описание тех директив, которые встречаются в примерах кода, приводимых в этой книге, а также директив, которые в примерах кода не встречаются, но которые я счел достаточно важными или интересными.

При описании директивы имя директивы обозначается полужирным шрифтом, а операнды директивы — полужирным курсивом. Необязательные операнды заключены в квадратные скобки.

```
.align abs-expr[, abs-expr[, abs-expr]]
```

Выравнивает текущую позицию на определенную границу. Граница определяется первым операндом; значение текущей позиции устанавливается таким образом, чтобы его адрес нацело делился на первый операнд. Если текущая позиция уже выровнена в этом смысле, то она оставляется без изменений. В противоположном случае, текущая позиция сдвигается на необходимое количество байт.

Образующееся при этом пустое пространство заполняется байтом, значение которого определяется вторым операндом. По умолчанию, в сегменте данных для заполнения используется 0, а в сегменте кода — код NOP (у x86 это 0x90).

Третий operand обозначает максимальное количество байт, которое может быть создано для заполнения. Если выравнивание требует сдвига на большее количество байт, то никакого выравнивания не происходит.

В примерах, приводимых в этой книге, оператор .align используется для того, чтобы выровнять элемент кода, управление на который передается только командами перехода, на границу слова. Благодаря этому, процессор не будет

тратить циклы шины на считывание команд, которые не будут исполняться в обозримом будущем.

.ascii "string" [...]

Директива `.ascii` создает один или более строчных литералов. Строки размещаются в генерируемом коде. В отличие от строчных литералов языка C, эти строки не завершаются нулевым байтом, если только программист не вставит этот завершающий нулевой байт сам. В строке могут использоваться все метасимволы, допускаемые стандартом ANSI C: `\n` обозначает перевод строки (ASCII LF), `\r` — возврат каретки (ASCII CR), `\"` — двойную кавычку, `\\` — обратный слэш, `\007` — символ с восьмеричным кодом 7 (ASCII BEL), `\0xA` — символ с шестнадцатиричным кодом A и т. д.

.asciz "string" [...]

Аналог директивы `.ascii`, дополняющий каждый создаваемый литерал нулевым байтом.

.byte [expressions]

Размещает произвольное целочисленное значение в текущем байте. Значение определяется значением выражения-операнда. Если operandов несколько, они размещаются в последовательных байтах.

.comm symbol , length

Определяет общий символ. Если одноименный символ будет определен в другом файле, то определение при помощи директивы `.comm` будет проигнорировано. Если такого определения найдено не будет, то редактор связей выделит количество байтов в секции `bss`, равное операнду `length`. Если в разных файлах одноименные символы будут иметь разную длину, при выделении памяти будет взята максимальная из длин.

При использовании ELF у директивы `.comm` можно задать третий operand, определяющий желаемое выравнивание объекта.

.data subsection

Последующие операторы будут ассемблироваться в секцию `data` или указанную субсекцию этой секции. Если субсекция не указана, операторы будут ассемблироваться в субсекцию 0.

.double flonums

Генерирует число с плавающей точкой двойной точности.

.else

Часть оператора условной компиляции (см. директиву `.if`).

.endif

Часть оператора условной компиляции (см. директиву **.if**).

.equ symbol, expression

Синоним директивы **.set**; устанавливает значение символа *symbol* равным значению выражения *expression*.

.equiv symbol, expression

Аналог директив **.equ** и **.set**, с тем отличием, что попытка изменить значение уже определенного символа с помощью этой директивы приведет к ошибке ассемблирования.

.extern symbol

В других ассемблерах описывает символ как внешний и определенный в другом файле. В GNU Assembler игнорируется.

.float flonums

Генерирует число с плавающей точкой одинарной точности.

.global symbol, .globl symbol

Делает символ глобальным, т. е. доступным при сборке программы. Обе формы записи этой директивы полностью функционально эквивалентны.

.hword expressions

Генерирует произвольные 16-битовые значения. Синоним директивы **.short**.

.if absolute expression

Начало директивы условной компиляции. Если выражение-операнд истинно (не равно нулю), то ассемблируется код, следующий за директивой **.if**, вплоть до директив **.else** или **.endif**.

Если директива **.else** присутствует, то если выражение-операнд истинно, то следующий за ней код игнорируется вплоть до директивы **.endif**.

Напротив, если выражение-операнд ложно (равно нулю), код до директивы **.else** или **.endif** игнорируется. Если директива **.else** присутствует, то код, следующий после нее и до директивы **.endif**, ассемблируется.

Важно понимать, что проверка условия происходит не во время исполнения программы, а во время ее компиляции.

Допустимы также следующие варианты директивы **.if**:

.ifdef symbol

Ассемблирует секцию кода, если символ *symbol* определен. Ассемблирует секцию **.else** в противном случае;

.ifndef symbol
 .ifndef symbol

Ассемблирует секцию кода, если символ *symbol* не определен.

.include "file"

Включает файл с именем *file* и обрабатывает его содержимое так, как будто бы оно было подставлено на место этой директивы.

.int expressions

Генерирует произвольные целочисленные 32-битные значения, равные значениям выражений-операндов.

.lcomm symbol , length

Выделяет *length* байт в секции *bss* и приравнивает символ *symbol* соответствующему перемещаемому адресу. Отличие от директивы **.comm** состоит в том, что символ не становится глобальным; если в каком-то другом файле будет определен одноименный символ, значение нашего символа не изменится.

.macro

Определяет макрос (макроопределение). Позволяет создавать собственные псевдокоманды, которые при ассемблировании раскрываются в последовательности команд и директив.

Синтаксис и точная семантика этой директивы довольно сложны и не будут подробно излагаться здесь.

.octa bignum

Генерирует 16-байтовые длинные целые числа.

.org new-1c , fill

Сдвигает текущую позицию (счетчик положения) в позицию, задаваемую операндом *new-1c*. Сдвиг может происходить только вперед и только в пределах секции.

Выражение *new-1c* должно быть определено к моменту использования и, как следует из предыдущего абзаца, может содержать только абсолютные символы и символы, принадлежащие к текущей секции.

При определенных обстоятельствах допускается использование абсолютного значения в качестве *new-1c*. Это значение задает не абсолютный адрес, а абсолютное смещение относительно начала текущей секции. Неаккуратное использование абсолютных значений таким образом может приводить к созданию нескольких вариантов кода или данных, которые должны были бы раз-

мещаться по одному и тому же адресу. Разрешение этой коллизии возлагается на линкер; разные линкеры решают ее различными способами, но даже если программа и создается, не гарантируется ее работоспособность.

Образующееся свободное пространство заполняется байтом, значение которого определяется выражением `fill`. Если выражение `fill` не указано, для сегментов данных используются нули, а для сегментов кода — NOP.

.quad *bignum*

Генерирует одно или несколько восьмибайтовых целых чисел со значениями, определяемыми выражениями-операндами.

.section *name*

Весь последующий код и метки считаются принадлежащими к секции *name*.

.set *symbol*, *expression*

Присваивает символу значение выражения.

.stabd, .stabn, .stabs

Генерация отладочной информации.

.text *subsection*

Последующий код ассемблируется в нумерованную субсекцию секции `text`. Если параметр `subsection` не указан, ассемблирование происходит в субсекцию 0.

Список источников информации

Документация

INMOS

INMOS 72 TRN 203 02

The Transputer Databook, INMOS document number: 72 TRN 203 02.

National Bureau of Standards

NBS FIPS PUB 46, 1977

Data Encryption Standard. National Bureau of Standards, Washington, FIPS PUB 46, 1977.

NASA

NASA 182505

Tomayko J.E. Computers in Spaceflight: The NASA Experience NASA Contractor Report 182505, 1988.

US Army Ordnance Dept.

1. *Goldstine/Neumann 1947*

Goldstine H., Neumann J., Planning and Coding of Problems for an Electronic Computing Instrument. Part II, I, Report under Contract W-36-034-ORD-7481, 1947.

2. *Clippinger 1948*

Clippinger R., A Logical Coding System Applied to the ENIAC (Electronic Numerical Integrator and Computer), Ballistic Research Laboratories, Report

No. 673, Project No. TB3-0007 of the Research and Development Division, Ordnance Department, 29 September 1948, текст доступен по адресу <http://ftp.arl.mil/~mike/comphist/48eniac-coding/>

Zortech

Zortech v3.x

Zortech C/C++ V3.x Programmers Guide, Zortech Inc. 1992.

КНИГИ И ПУБЛИКАЦИИ

1. *Anderson et al 1992*

Chutani S., Anderson O., Kazar M., Leverett B., Mason W., Sidebotham R., The Episode File System, Proceedings of the Winter 1992 USENIX Conference, January 1992, 43-60., текст доступен по адресу <http://transarc.com/~ota/episode.ps.gz>

2. *Barron 1971*

Barron D.W. Computer Operating Systems. Chapman and Hall, London, 1971.

3. *Bobrow et al 1972*

. Bobrow D., Burchfiel J., Murphy D., Tomlinson R., TENEX, a paged time sharing system for the PDP – 10, Communications of the ACM, Volume 15, Issue 3 (March 1972), Pages: 135 – 143, текст доступен по адресу http://portal.acm.org/ft_gateway.cfm?id=361271&type=pdf&coll=portal&dl=ACM&CFID=6337112&CFTOKEN=70794952

4. *Boling 2001*

Boling D. Programming Microsoft Windows Ce. Microsoft Press 2001.

5. *Bonwick 1994*

The Slab Allocator: An Object-Caching Kernel Memory Allocator, USENIX Summer 1994 Technical Conference, 1994, текст доступен по адресу http://www.usenix.org/publications/library/proceedings/bos94/full_papers/bonwick.ps

6. *Friedhelm/Shmidt 1997*

Friedhelm Schmidt. The Scsi Bus and Ide Interface: Protocols, Applications and Programming. Addison-Wesley Pub Co, 1997.

7. *Gibson/Katz/Patterson 1988*

Gibson G., Katz R., Patterson A. A Case for Redundant Arrays of Inexpensive Disks (RAID). Proceedings of the conference on Management of data. NY, ACM Press, 1988.

8. Goldstine 1972

Goldstine H., The Computer: from Pascal to von Neumann. Princeton, New Jersey: Princeton University Press, 1972.

9. Heising 1963

Heising W.P. IBM Systems J., 1963.

10. Huffman 1952

Huffman D.A. "A Method for the Construction of Minimum Redundancy Codes". Proceedings of the Institute of Radio Engineers 40, 1952.

11. Karger/Shell 1974

Karger P., Shell R., Multics Security Evaluation: Vulnerability Analysis, US Air Force report, 1974. повторно опубликовано Proceedings 18th Annual Computer Security Applications Conference. Los Alamitos, CA, , IEEE Computer Society, p.119-266 2002. Текст доступен по адресу <http://csrc.nist.gov/publications/history/karg74.pdf>

12. Kay/Lauder 1988

Kay J., Lauder P., A Fair Share Scheduler, Communications of the ACM, January 1988, текст доступен по адресу <http://cs.su.oz.au/piers/share.ps.gz>

13. Lempel/Ziv 1978

Lempel A., Ziv D. "Compression of individual sequences via variable rate coding". IEEE Trans. Inform. Th., IT-24 1978.

14. McGivern 2000

McGivern J. Interrupt-Driven PC Systems Design. The Coriolis Group, 2000.

15. Papamarcos/Patel 1984

Papamarcos M., Patel J., A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories, Proc. 11th Ann. Int'l Symp. on Computer Architecture, ACM, 1984.

16. Pareto 1897

Pareto V. Cours d'Economie Politique 2. Lausanne, Rouge, 1897.

17. PC Magazine 1995

Профит Б. Высокопроизводительная файловая система HPFS. //PC Magazine russian edition, 1995, no.10.

18. QNX 2004

Коллектив авторов. Практика работы с QNX, Комбук, 2004.

19. Shannon 1948

Shannon C.E. A mathematical theory of communication. Bell System Tech. J., 1948.

20. Smith 1997

Smith S.W. The Scientist and Engineer's Guide to Digital Signal Processing. California Technical Publishing, 1997, доступна также на сайте <http://www.dspguide.com>

21. Ungar 1984

Ungar D.M. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. ACM SIGPLAN Notices, 19(5):157-167, 1984.

22. Zipf 1949

Zipf G.K. Human Behavior and the Principle of Least Effort. Reading, Addison — Wesley 1949.

23. Weik 1961

Weik R., The ENIAC Story, The Journal of the American Ordnance Association, January-February 1961, текст доступен по адресу <http://ftp.arl.mil/~mike/comphist/eniac-story.html>

24. Wilkes/Wheeler 1951

Wilkes M., Wheeler D., Gill S., The Preparation of Programs for an Electronic Digital Computer, Special Reference to the EDSAC and the Use of a Library of Subroutines -Addison Wesley, 1951.

25. Агуров 2006

Агуров П. Интерфейс USB. Практика использования и программирования — СПб.: БХВ-Петербург 2006.

26. Андреев/Беззубов 1999

Андреев А., Беззубов Е., Емельянов М., Кокорева О., Чекмарев А. Новые технологии Windows 2000. Наиболее полное руководство в подлиннике. — СПб.: БХВ-Петербург, 1999.

27. Анин 2000

Анин Б. Защита компьютерной информации. — СПб.: БХВ-Петербург, 2000.

28. Ахметов 2001

Ахметов К. Знакомство с Microsoft Windows XP. Русская редакция Microsoft Press, 2001.

29. Баррон 1974

Баррон Д. Ассемблеры и загрузчики. — М.: Мир, 1974.

30. Баурн 1986

Баурн С. Операционная система UNIX. — М.: Мир, 1986.

31. Берлекэмп 1971

Берлекэмп Э. Алгебраическая теория кодирования. — М.: Мир, 1971.

32. Бобровски 1998

Бобровски С. Oracle 8: Архитектура. — М.: Лори, 1998.

33. Василеску 1990

Василеску Ю. Прикладное программирование на языке Ада. — М.: Мир, 1990.

34. Вебер 1999

Вебер Дж. Технология Java. Наиболее полное руководство. — СПб.: BHV-Петербург, 1999.

35. Вилански 1999

Вилански Э. Официальный тест MCSE 70 — 067: Microsoft Windows NT Server 4.0. Русская редакция Microsoft Press, 1999.

36. Грогоно 1982

Грогоно П. Программирование на языке Паскаль. — М.: Мир, 1982.

37. Гук 2000

Гук М. Аппаратные средства IBM PC. — СПб.: Питер, 2000.

38. Гук 2005

Гук М. Шины PCI, USB и FireWire. Энциклопедия. — СПб.: Питер, 2005.

39. Дейкстра 1978

Дейкстра Э. Дисциплина программирования. — М.: Мир, 1978.

40. Дейт 1988

Дейт К. Дж. Руководство по реляционной СУБД DB2. — М.: Финансы и Статистика, 1988.

41. Дейт 1999

Дейт К. Дж. Введение в системы баз данных. — М.: Вильямс, 1999.

42. Дейтел 1984

Дейтел Г. Операционные системы. — М.: Мир, 1984.

43. Дейтел 1987

Дейтел Г. Введение в операционные системы. — М.: Мир, 1987.

44. Иртегов 2004

Иртегов Д. Введение в сетевые технологии — СПб.: БХВ-Петербург 2004.

45. Исида 1988

Исида Х. Программирование для микрокомпьютеров. — М.: Мир, 1988.

46. Карнап 1971

Карнап Р. Философские основания физики. — М.: Мир, 1971.

47. Касперски 2001

Касперски К. Образ мышления — дизассемблер IDA. Том I. Описание функций встроенного языка IDA Pro. — М.: Солон-Р, 2001.

48. Кейслер 1986

Кейслер С. Проектирование операционных систем для малых ЭВМ. — М.: Мир, 1986.

49. Керн/Линд 2000

Керн С., Линд Д. Lotus Notes и Domino R5. Энциклопедия пользователя. — К.: Диасофт, 2000.

50. Керниган/Пайк 1992

Керниган Б., Пайк Р. Unix — универсальная среда программирования. — М.: Финансы и статистика, 1992.

51. Керниган/Ритчи 2000

Керниган Б., Ритчи Д. Язык программирования С. — СПб.: Невский Диалект, 2000.

52. Кичев/Некрасов 1988

Кичев Г., Некрасов Л. Архитектура и аппаратные средства мини-ЭВМ СМ 1600. — М.: Машиностроение, 1988.

53. Кнут 2000

Кнут Д.Э. Искусство программирования. — Издательский дом Вильямс, 2000.

54. КомпьютерПресс 1991

Моисеенков И. Суeta вокруг Роберта или Моррис-сын и все, все, все. //КомпьютерПресс, 1991, no. 8,9.

55. КомпьютерПресс 1993

Новосельцев С. Предисловие к ст. Мир Apple. //КомпьютерПресс, 1993, no.11.

56. Кормен/Лейзерсон/Ривест 2000

Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы, построение и анализ. — М.: МЦНМО, 2000.

57. Krakovjak 1988

Краковяк С. Основы организации и функционирования ОС ЭВМ. — М.: Мир, 1988.

58. Maxwell 2000

Максвелл С. Ядро Linux в комментариях. — М.: Диасофт, 2000.

59. MikroEVM 1988

МикроЭВМ, книга 2, Персональные ЭВМ; под ред. Преснухина Л. М. — Высшая Школа, 1988.

60. Minaasi/Kamarada 1996

Минаси М., Камарда Б. и др. OS/2 Warp изнутри. — СПб.: ПИТЕР-Пресс, 1996.

61. Miniskiy 1971

Минский М. Вычисления и автоматы. — М.: Мир, 1971.

62. Organik 1987

Органик С. Организация системы Интел 432. — М.: Мир, 1987.

63. Pankratov 2001

Панкратов Е. Операционная система MS-DOS 6.22. Справочное пособие. — Познавательная книга плюс, 2001.

64. Pappas/Marri 1993

Паппас К., Марри У. Микропроцессор 80386. — М.: Радио и Связь, 1993.

65. Pirogov 2005

Пирогов В.Ю. Ассемблер для Windows (3-е издание). — СПб.: БХВ-Петербург, 2005.

66. Pirogov 2006

Пирогов В.Ю. Ассемблер и дизассемблирование. — СПб.: БХВ-Петербург, 2006.

67. Prokhorov 1990

Малые ЭВМ высокой производительности. Архитектура и программирование; под ред. Н. Л. Прохорова. — М.: Радио и связь, 1990.

68. Robachevskiy 1999

Робачевский А. Операционная система Unix. — СПб.: БХВ-Петербург, 1999.

69. Sibesta 2001

Себеста Р. Основные концепции языков программирования. — М.: Вильямс, 2001.

70. Стивенс 2002

Стивенс У. UNIX: взаимодействие процессов. — СПб.: Питер, 2002.

71. Страуструп 1999

Страуструп Б. Язык программирования C++. — СПб., М.: Невский Диалект — Издательство БИНОМ, 1999.

72. Танненбаум 2002

Танненбаум Э. Компьютерные сети. 3-е изд. — СПб: Питер, 2002.

73. Танненбаум 2006

Танненбаум Э. Современные операционные системы. 2-е изд. — СПб: Питер, 2006.

74. Танненбаум/Ван-Стеен 2004

Танненбаум Э., Ван Стен М., Распределенные системы — принципы и парадигмы. — СПб: Питер, 2004.

75. Уэллин 2004

Уэллин С. Как не надо программировать на C++. — СПб.: Питер, 2004.

76. Финогенов 2001

Финогенов К. Самоучитель по системным функциям MS-DOS — М.: Горячая линия — Телеком, 2001.

77. Харп 1993

Транспьютеры. Архитектура и программное обеспечение; под ред. Г. Харпа. — М.: Радио и связь, 1993.

78. Хевиленд/Грэй/Салама 2000

Хевиленд К., Грэй Д., Салама Б. Системное программирование в UNIX. — М.: ДМК Пресс, 2000.

79. Хоар 1989

Хоар Ч. Взаимодействующие последовательные процессы. — М.: Мир, 1989.

80. Ховард/Лебланк 2004

Ховард М., Лебланк Д. Защищенный код. — М.: Microsoft Press, Русская редакция, 2004.

81. Хэрриот

Хэрриот Д. Из воспоминаний сельского ветеринара (неоднократно перевиздавалась).

82. Шнайер 2003

Шнайер Б. Секреты и ложь. Безопасность данных в цифровом мире — СПб.: Питер, 2003.

Интернет-ресурсы

Adobe

partners.adobe.com

Adobe Solutions Network: Postscript,

<http://partners.adobe.com/asn/developer/technotes/postscript.html>

ANSI

1. *www.t10.org architecture*

SCSI-3 Architecture, <http://www.t10.org/scsi-3.htm>

2. *www.t10.org commands*

SCSI-3 Primary Commands (SPC), <ftp://ftp.t10.org/t10/drafts/spc/spc-r11a.pdf>

3. *www.t13.org*

ATA specification working group, <http://www.t13.org>

Atmel

1. *www.atmel.com*

AVR Assembler User Guide, <http://www.atmel.com>

2. *www.atmel.com AVR305*

AVR305: Half Duplex Compact Software UART,

<http://www.atmel.com/atmel/acrobat/doc0952.pdf>

IBM

1. *www.ecomstation.com*

eComstation — Super Client For the I'net Generation,

<http://www.ecomstation.com/>

2. *www.128.ibm.com mcpi*

Great moments in microprocessor history,

<http://www-128.ibm.com/developerworks/library/pa-microhist.html>

3. *www.ibm.com OS/2 DDK*

IBM OS/2 On-Line Device Driver Kit — Home Page,
<http://service.software.ibm.com/ddk>

4. *www.ibm.com zOS*

z/OS home page, <http://www-1.ibm.com/servers/eserver/zseries/zos/>

5. *www6.software.ibm.com jfs-layout*

Best S., Kleikamp D., JFS layout,
<ftp://www6.software.ibm.com/software/developer/library/jfslayout.pdf>

6. *redbooks.ibm.com sg242222*

Bitterer A., AS/400 Programming with VisualAge for RPG, IBM Redbook,
<http://http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg242222.pdf>

7. *redbooks.ibm.com sg245992*

MacIsaac M., C/C++ Applications on z/OS and OS/390 UNIX,
<http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg245992.pdf>

8. *redbooks.ibm.com sg245234*

Murhammer M., A Comprehensive Guide to Virtual Private Networks,
Volume II: IBM Nways Router Solutions, IBM Redbook,
<http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg245234.pdf>

9. *redbooks.ibm.com sg245341*

Collins L., Lotus Notes and Domino R5.0 Security Infrastructure Revealed,
IBM Redbook,
<http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg245341.pdf>

10. *redbooks.ibm.com sg245393*

Naick I., OS/2 Warp Server for e-business,
<http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg245393.pdf>

11. *redbooks.ibm.com sg245597*

Rogers P., ABCs of System Programming Volume 1,
<http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg245597.pdf>

IETF

1. *IEN 137*

Cohen D., IEN137 On holy wars and a plea for peace, <http://www.ietf.org>
(также опубликована в IEEE Computer, Vol 14, Issue 10, Oct. 1981)

2. *RFC 0822*

Crocker D., RFC0822 Standard for the format of ARPA Internet text messages,
<http://www.ietf.org>

3. RFC 0793

Postel J., RFC0793 Transmission Control Protocol, <http://www.ietf.org>

4. RFC 1320

Rivest R., RFC1320 The MD4 Message-Digest Algorithm, <http://www.ietf.org>

5. RFC 1321

Rivert R., RFC1321 The MD5 Message-Digest Algorithm, <http://www.ietf.org>

Intel

1. *www.intel.com Pentium4*

Intel® Pentium® 4 Processor Family: Technical documentation,
<http://www.intel.com/design/Pentium4/documentation.htm>

2. *www.intel.com UHCI*

Universal Host Controller Interface (UHCI) Design Guide
<http://download.intel.com/technology/usb/UHCI11D.pdf>

Linux

1. *HOWTO library*

Program Library HOWTO, <http://www.linux.org/docs/ldp/howto/Program-Library-HOWTO/index.html>

2. *HOWTO khg*

Johnson Michael K., The Linux Kernel Hacker Guide,
<ftp://tsx11.mit.edu/pub/linux/docs/LPD/khg-0.6.tar.gz>

3. *www.linux.org*

The Linux Home Page at Linux Online, <http://www.linux.org>

Microchip

1. *www.microchip.com PICMicro*

PICMicro Mid-Range MCU Family Reference Manual,
<http://www.microchip.com>

2. *www.microchip.com AN585*

A Real-Time Operating System for PIC16/17, Microchip Application Note AN585, Documentation: <http://www.microchip.com/1010/suppdoc/appnote/category/pic16/940/index.htm>, Source code: <http://www.microchip.com/1010/suppdoc/appnote/category/pic16/941/index.htm>

Microsoft

1. *MS04-028*

Buffer Overrun in JPEG Processing (GDI+) Could Allow Code Execution (833987), Microsoft Security Bulletin MS04-028, 2004,
<http://www.microsoft.com/technet/security/bulletin/MS04-028.mspx>

2. *MSDN C++ exception*

C++ Language Reference, The try, catch, and throw Statements,
<http://msdn2.microsoft.com/en-us/library/6dekhbbc.aspx>

3. *MSkb RU270125*

RU270125 — после установки Microsoft Project 2000 приложения Microsoft Office 2000 завершают работу сразу после запуска,
<http://support.microsoft.com/kb/270125/ru>

4. *support.microsoft.com 155217*

Включение и отключение автоматического запуска компакт-дисков,
<http://support.microsoft.com/kb/155217/ru>

5. *www.microsoft.com boot.ini*

Boot INI Options Reference,
<http://www.microsoft.com/technet/sysinternals/information/bootini.mspx>

6. *www.microsoft.com UMDF*

User-Mode Driver Framework (UMDF)
<http://www.microsoft.com/whdc/driver/wdf/UMDF.mspx>

7. *www.microsoft.com*

Microsoft Corporation, <http://www.microsoft.com>

National Semiconductors

NS PC16552D

PC16552D Dual Universal Asynchronous Receiver/Transmitter with FIFOs,
<http://www.national.com/pf/PC/PC16552D.html>

SPARC

www.sparc.com v9

The SPARC Architecture Manual Version 9, <http://www.sparc.com>

Sun Microsystems

1. *docs.sun.com 816-0559-10*

Linker and Libraries Guide, part number 816-0559-10, <http://docs.sun.com>

2. *docs.sun.com 806-3774-10*

SPARC Assembly Language Reference Manual, part number 806-3774-10,
<http://docs.sun.com>

3. *docs.sun.com 805-7378-10*

Writing Device Drivers, part number 805-7378-10, <http://docs.sun.com>

4. *docs.sun.com 805-7478-10*

STREAMS programming guide, part number 805-7478-10,
<http://docs.sun.com>

5. *research.sun.com scalable*

Moir M., Who Will Rid Me of these Wretched Locks? http://research.sun.com/sunlabsday/docs/Talks/Track3/3.03_moir_hybridt2.pdf

6. *www.sun.com 2001-05*

Sun Fire [tm] 6800 Midframe Server With Sun StorEdge[tm] Arrays Leads Competition in Performance Per CPU, Sun Microsystems Inc. press release,
<http://www.sun.com/smi/Press/sunflash/2001-05/sunflash.20010514.2.html>

7. *www.sun.com USIIIv2.pdf*

UltraSPARC III Cu User's Manual
<http://www.sun.com/processors/manuals/USIIIv2.pdf>

8. *www.sun.com zfs*

ZFS — the last word in file systems, <http://www.sun.com/2004-0914/feature/>

Другие ресурсы

1. *bbc.co.uk password*

Passwords revealed by sweet deal,
<http://news.bbc.co.uk/1/hi/technology/3639679.stm>

2. *bdu.borland.com backdoor*

Kaster J., A little history on the Interbase security hole,
<http://bdu.borland.com/article/26611>

3. *blogs.msdn.com larryosterman 174516*

Why should I even bother to use DLL's in my system? Larry Osterman's WebLog, July 06, 2004, <http://blogs.msdn.com/larryosterman/archive/2004/07/06/174516.aspx>

4. *blogs.msdn.com theoldnewthing*

The Old New Thing, Raimond Chen Weblog,
<http://blogs.msdn.com/oldnewthing/default.aspx>

5. *blogs.sun.com paulsan 14 June 2005*

Changes to Hme Block Handling, Paul Sandhu's Weblog, June 14, 2005,
http://blogs.sun.com/paulsan/entry/changes_to_hme_block_handling

6. *blog.technet.com markrussinovich 2005/10/31*

Russinovich, M., Sony, Rootkits and Digital Rights Management Gone Too Far, <http://blogs.technet.com/markrussinovich/archive/2005/10/31/sony-rootkits-and-digital-rights-management-gone-too-far.aspx>

7. *Brevik 2003*

PDP-11 OS Release Dates,
<http://groups.google.com/group/comp.os.vms/msg/51870e4fc2d88da5>

8. *bugtraq RFP2K02*

UMBRA Advisory RFP2K02, "Netscape engineers are weenies!" A back door in Microsoft FrontPage extensions/authoring components, список рассылки bugtraq@securityfocus.com, <http://www.mail-archive.com/bugtraq@securityfocus.com/msg03492.html>

9. *www.cert.org*

CERT® Advisory CA-2002-28 Trojan Horse Sendmail Distribution, 2002,
<http://www.cert.org/advisories/CA-2002-28.html>

10. *citforum.ukrpak.net IDE*

Интерфейс IDE, <http://citforum.ukrpak.net/hardware/bookide/>

11. *chip.ms.mff.cuni.cz ATA2*

AT Attachment interface with extensions (ATA-2) Working draft Revision 3, 1995, <http://chip.ms.mff.cuni.cz/~pcguts/hd/ata2.ps.gz>

12. *chip.ms.mff.cuni.cz IDE*

IDE — Hardware Reference & Information Document,
http://chip.ms.mff.cuni.cz/~pcguts/hd/F_IDE-tech.html

13. *cp.sonybmg.com xcp*

CD's Containing XCP Content Protection Technology,
<http://cp.sonybmg.com/xcp/english/titles.html>

14. *dz.yandex.ru*

DZ online, <http://dz.yandex.ru>

15. *Goodwin 2005*

Goodwin R., Hyperthreading hurts server performance, say developers,
ZDNet UK 18 Nov 2005,
<http://news.zdnet.co.uk/hardware/0,1000000091,39237341,00.htm>

16. *java.sun.com HotSpot*

The Java HotSpot Virtual Machine, White paper,
http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.html

17. *jbayko v12.1.2*

Bayko J., Great Microprocessors of the Past and Present (V 12.1.2),
<http://www3.sk.sympatico.ca/jbayko/cpu.html>

18. *isc.sans.org Santy*

Santy worm defaces websites using php bug, 2004,
<http://isc.sans.org/diary.php?date=2004-12-21>

19. *micro.magnet.fsu.edu russians*

Steal The Best, Molecular Expressions: The Silicon Zoo,
<http://micro.magnet.fsu.edu/creatures/pages/russians.html>

20. *Murphy 1989*

Murphy D., Origins and Development of TOPS-20,
<http://www.opost.com/dlm/tenex/hbook.html>

21. *Neil 2000*

Neil T. Ethernet SANs: Panacea or fantasy,
http://www.lsilogic.com/contributed_articles/heli0501.pdf

22. *news.com.com Xbox challenge*

Hacker cracks Xbox challenge, <http://news.com.com/2100-1043-994794.html>

23. *opensolaris.org*

Home at OpenSolaris.org, <http://opensolaris.org>

24. *PC Guide PCI*

PC Guide — Ref — PCI Local Bus,
<http://www.pcguide.com/ref/mbsys/buses/types/pci.htm>

25. *PC Guide IDE*

PC Guide — Ref — Integrated Drive Electronics / AT Attachment (IDE/ATA) Interface, <http://www.pcguide.com/ref/hdd/if/ide/index.htm>

26. *PC Guide SCSI*

PC Guide — Ref — Small Computer Systems Interface,
<http://www.pcguide.com/ref/hdd/if/scsi/index.htm>

27. *pcisig.com*

PCI Special Interest Group, <http://pcisig.com>

28. *perso.wanadoo.fr*

UNIX history, <http://perso.wanadoo.fr/levenez/unix>

29. *port25.technet.com vista boot*

Using Vista's Boot Manager to Boot Linux and Dual Booting with BitLocker Protection with TPM Support

http://port25.technet.com/archive/2006/10/13/Using-Vista_2700_s-Boot-Manager-to-Boot-Linux-and-Dual-Booting-with-BitLocker-Protection-with-TPM-Support.aspx

30. *risks 1997*

What really happened on Mars Rover Pathfinder, The Risks Digest, Volume 19: Issue 49, 9 December 1997, <http://catless.ncl.ac.uk/Risks/19.49.html>

31. *solutions.sun.com 802-7100-01*

MicroSPARC-IIep User's Manual

<http://solutions.sun.com/embedded/databook/pdf/manuals/802-7100-01.pdf>

32. *sourceforge.net dosbox*

DOSBox, a x86 emulator with DOS, <http://dosbox.sourceforge.net>

33. *support.pkware.com appnote*

ZIP File Format Specification,

http://support.pkware.com/business_and_developers/developer/popups/appnote.txt

34. *techupdate.zdnet.com*

DiCarlo L., Compaq to halt NT on Alpha development,
<http://techupdate.zdnet.com/techupdate/stories/main/0,14179,2318096,00.html>

35. *VMSDOC CHF*

HP OpenVMS Programming Concepts Manual, Chapter 9 — Condition-Handling Routines and Services,

http://h71000.www7.hp.com/doc/82FINAL/5841/5841pro_026.html#cond_handling_chap

36. ***www.acnc.com***
RAID Levels, http://www.acnc.com/04_01_00.html
37. ***www.angstrom.ru mk-mp***
ОАО "Ангстрем", Продукты: Микроконтроллеры и микропроцессоры,
<http://www.angstrom.ru/product/mk-mp.htm>
38. ***www.benedelman.org 020305-1***
How VeriSign Could Stop Drive-By Downloads,
<http://www.benedelman.org/news/020305-1.html>
39. ***www.boingboing.net prehistory_of_the_so***
Pre-history of the Sony rootkit,
http://www.boingboing.net/2005/11/27/prehistory_of_the_so.html
40. ***www-ccs.cs.umass.edu***
Got a Match, <http://www-ccs.cs.umass.edu/~shri/iPic.html>
41. ***www.caldera.com***
С 2000 года ссылка на сайт The SCO Group, Inc (www.sco.com)
42. ***www.cert.org***
CERT Coordination Center, <http://www.cert.org>
43. ***www.cs.hut.fi SSH***
Ssh (Secure Shell) Home Page, <http://www.cs.hut.fi/ssh/>
44. ***www.cs.vu.nl minix***
Minix information sheet, <http://www.cs.vu.nl/~ast/minix.html>
45. ***www.digit-life.com NTFS***
Mikhailov D. NTFS file system, <http://www.digit-life.com/articles/ntfs>
46. ***www.distributed.net***
distributed.net: Node Zero, <http://www.distributed.net>
47. ***www.fsf.org***
Free Software Foundation
48. ***www3.gartner.com 101034***
Pescatore J., Nimda Worm Shows You Can't Always Patch Fast Enough,
http://www3.gartner.com/DisplayDocument?doc_cd=101034
49. ***www.gnu.org***
GNU Not Unix!, <http://www.gnu.org>

50. *www.gnu.org gas-2.9.1*

Elsner D., Fenlason J. & friends, Using as, The GNU Assembler,
http://www.gnu.org/software/binutils/manual/gas-2.9.1/html_mono/as.html

51. *www.gnu.org gas-2.9.1 16*

Elsner D., Fenlason J. & friends, Using as, The GNU Assembler, 80386 Dependent Features, http://www.gnu.org/software/binutils/manual/gas-2.9.1/html_chapter/as_16.html

52. *www.honeynet.org*

The Honeynet Project, <http://www.honeynet.org>

53. *www.ibm.com NUMA-Q*

The IBM NUMA-Q enterprise server architecture, <http://www.ibm.com>

54. *www.intel.com Moore*

What is Moore's law?,
<http://www.intel.com/intel/museum/25anniv/hof/moore.htm>

55. *www.jpeg.org*

Home site of the JPEG and JBIG committees, <http://www.jpeg.org>

56. *www.netapp.com 3002*

File System Design for an NFS File Server Appliance,
<http://www.netapp.com/library/tr/3002.pdf>

57. *www.ntbugtraq.com*

NTBugtraq, <http://www.ntbugtraq.com>

58. *www.opengroup.org xu007*

Certification > The Open Brand > Register of Open Branded Products > International Business Machines Corporation — UNIX 95,
<http://www.opengroup.org/regproducts/xu007.htm>

59. *www.phpbb.com*

phpBB: Creating Communities, <http://www.phpbb.com>

60. *www.research.ibm.com*

How Deep Blue Works,
<http://www.research.ibm.com/deepblue/meet/html/d.3.2.html>

61. *www.rsa.com FAQ*

RSA's FAQ About Today's Cryptography: RSA,
http://www.rsa.com/rsalabs/faq/faq_rsa.html

62. *www.sco.com*

The SCO Group, Inc.

63. *www.seti.org*

TeamSETI, <http://www.seti.org/teamseti.html>

64. *www.sourceforge.net bo2k*

Project: Back Orifice 2000, <http://sourceforge.net/projects/bo2k/>

65. *www.tuxedo.org homesteading*

Raymond E. S. The Cathedral and the Bazaar,
<http://www.tuxedo.org/~esr/writings/homesteading/>

66. *www.usb.org USB2*

USB 2.0 Specification, <http://www.usb.org/developers/docs/>

67. *www.x.org*

The X Protocol, http://www.x.org/about_x.htm

68. *www.wired.com slammer_pr*

Slammed!, Issue 11.07,
http://www.wired.com/wired/archive/11.07/slammer_pr.html

69. *Gottry 2002*

Gottry K. Pick up performance with generational garbage collection,
<http://www.javaworld.com/javaworld/jw-01-2002/jw-0111-hotspotgc.html>

70. *Bax 1986*

Бах М. Архитектура операционной системы UNIX,
http://httpmirror.hwc.ru/www.citforum.ru_80/operating_systems/bach/contents.shtml

71. *Коган/Роусон 1989*

Коган М.С., Роусон Ф.Л. ОПЕРАЦИОННАЯ СИСТЕМА/2,
<http://www.os2.ru/database/books/files/os2art.zip>

72. *Лафо/Нортон 1991*

Лафо Р., Нортон П. OS/2 изнутри, <http://lib.perm.ru/base/os2insid.zip>

Предметный указатель

3

32-разрядный доступ к диску 888

А

a.out 160, 161, 316, 867, 986
ABI (Application Binary Interface) 43, 181
ACB, OS/2 602
Account manager *См. Администратор учетных записей*
Account *См. Учетная запись*
ACL (Access Control List) 685, 687, 688, 772, 773—778, 780—784, 787, 794—796, 819, 825, 838, 841, 844, 863
ActiveX 716, 822, 830, 889, 905
Ada 135, 312, 406—408, 432
AdWare 716, 812
AGP (Accelerator Graphic Port) 510, 511—513
AIX 23, 38, 45, 318, 430, 574, 677, 680—682, 687, 689, 871, 873, 894
Allocator *См. Аллокатор*
Alpha 47, 182, 215, 366, 747, 861, 862, 902, 926
ANSI C 44, 57, 722, 873, 990
AoE 560
API (Application Programming Interface) 23, 43, 256, 421, 423, 428, 463, 464, 468, 690, 727, 742, 753, 868, 879, 880, 884, 888, 914, 915, 916, 923, 924
APIC (Advanced Programmable Interrupt Controller) 476, 947, 948, 950

ARPA Net 867

AS/400 (Application Server/400) 26, 58, 93, 192, 308, 340, 345

ASCII (American Standard Code for Information Interchange) 58, 59, 419, 608, 628, 637, 638, 644, 652, 653, 734, 735, 770, 850, 859, 957, 984, 990

ASP (Active Server Pages) 23, 458, 794, 817, 843

ASP (Application Service Provider) 23, 458, 794, 817, 843

AST (Asynchronous System Trap) 398, 622, 744, 855

ATA (AT Attachment) 26, 201, 486, 513, 540, 546, 552, 553, 560, 580, 600—603, 879, 896, 949

ATAPI (ATA Packet Interface) 552, 580

Б

Backdoor 811, 818

Backend 426

Basic 155, 157, 194, 255, 271, 272, 727, 748, 831, 882, 923

Best fit 226

Big endian 105, 181

BIOS (Basic Input/Output Service) 200—205, 208, 211, 513, 514, 567, 568, 736, 760, 832, 881—887, 911, 912, 923—925, 949

Bit *См. Бит*

Block suballocation *См. Субаллокация*

Boot record *См. Загрузочная запись*

- B**
- Boot virus См. Загрузочный вирус
BOOT.INI 211
Bootstrap См. Бутстрон
Botnet 762
Breakpoint См. Точка останова
Bridge См. Мост
BSD 38, 43, 82, 205, 273, 301, 334, 412, 459, 572, 638, 661, 665, 676, 734, 754, 803, 814, 833, 867—869, 871—879, 894, 899, 921, 947
BSOD (Blue Screen of Death) 714
Buffer overrun См. Переполнение буфера, Срыв буфера
Bug-for-bug compatibility 198, 326
Bus error См. Ошибка шины
Bus master См. Задатчик шины
Bus См. Шина
В-деревья 657, 658, 660
В+-деревья 659
- C**
- C 80, 81, 94, 95, 114, 115, 117, 124, 130, 135, 137, 143, 155, 156, 164, 165, 169, 170, 173, 178, 179, 192, 194, 197—199, 201, 213, 217, 218, 223, 229, 230, 233, 241, 253, 254, 255, 258, 259, 260, 265, 271, 272, 284, 299, 301, 308, 318, 319, 388, 399, 406, 410, 413, 420, 427, 432, 434, 523, 572, 583, 618, 619, 622, 623, 635, 638, 639, 643, 644, 647, 662, 703, 722, 724, 725, 727, 730, 732, 734, 735, 741, 748, 750, 751, 752, 753, 754, 800—804, 825, 833, 855, 859, 864, 870, 873, 878, 879, 881, 884, 886, 892, 893, 948, 956, 990
C++ 752, 753
Cache См. Кэши
Cache line См. Стока кэша
Cache washing См. Промывание кэша
Callback 574—576, 616, 623, 727, 733
Capability См. Полномочие
Carry См. Перенос
CC-NUMA 372
CD (Compact Disk) 67, 204, 212, 329, 387, 388, 527, 543, 552, 579, 580, 653, 710, 820, 821, 826, 896
- CDDL (Common Development and Distribution License) 584, 697, 875
CDFS (Compact Disk File System) 651, 654, 698
Centronics 498, 923
Checkpoint См. Контрольная точка
CHF (Condition Handling Facility) 744
Chicago 14
Chipset См. Чипсет
CISC 129—132
Cisco IOS 42
Citrix ICA 34, 767
Clock-алгоритм 332—334, 346
COFF (Common Object File Format) 316, 318, 985, 986
COM (Common Object Model) 167, 193, 199, 200, 255, 267, 585, 675, 748, 822, 854, 884, 889
COMA 372
Commit См. Подтверждение транзакции
Completion ports См. Порты завершения
Compression См. Сжатие данных, Упаковка данных
Connectix Virtual PC 26
Consistency См. Согласованность, Целостность
Copy-on-write 336, 338, 344, 693
CP/M 38, 41, 160, 216, 567, 634—638, 641, 644, 645, 822, 825, 845, 866, 881—885, 888, 915
CPL, x86 932
CPU (Central Processing Unit) 455, 456
CRC (Cyclic Redundancy Check/Code) 70, 71, 72, 78
CSMA/CD 387
CUA (Common User Access) 716
- D**
- Daemon См. Демон
DARPA (Defence Advanced Research Project Agency) 849, 867
DASD (Direct Access Storage Device) 474
Data fork См. Ветвь данных
DBCS (Double Byte Character Set) 58
DCL (DEC Command Language) 854, 856, 857, 882, 885, 900
DCU 209

Debug information См. Отладочная информация
 Debugger См. Отладчик
 DEC Alpha 307, 846, 902, 926
 DEC PDP-11 22
 DES (Digital Encryption Standard) 74, 76, 771
 DesqView 213, 887, 914, 939, 940
 DFM 541
 Dictionary attack См. Словарная атака
 Directory service См. Служба каталогов
 Directory traversal bug См. Ошибка подъема по каталогам
 Directory См. Директория
 Dirty flag См. Флаг загрязнения
 Disk См. Диск
 Display См. Дисплей
 DLL (Dynamic Link Library) 189, 193—199, 211, 214, 299, 306, 315—320, 322, 336, 711, 810, 817, 822, 831, 834, 888—892, 897, 899, 915, 922, 941
 DMA (Direct Memory Access) 22, 353
 DMA controller 353
 DMCA (Digital Millennium Copyright Act) 152, 907, 912, 925
 DMD, OS/2 578
 Doom 14
 DoS (Denial of Service) 756, 757, 758, 805, 839
 DOS extender См. Расширитель DOS
 DOS 17, 20—25, 28, 44, 82, 165—169, 174, 200—205, 208, 211—213, 265—269, 398, 467, 563, 566—568, 583, 609, 622—625, 634, 637, 641, 645, 652, 654—656, 661, 673, 675, 728, 736—738, 748, 822, 824, 843, 852, 854, 855, 866, 883—891, 893, 913—915, 919, 920, 922, 924, 935—939, 940, 982
 DOSBox 893
 DPI (Dots Per Inch) 62
 DPMI (DOS Protected Mode Interface) 886, 922
 DR DOS 174, 467, 583, 623, 634, 637, 652, 654, 736, 737, 748, 822, 885, 887, 913, 920
 Driver См. Драйвер
 DRM (Digital Rights Management) 584, 826, 832, 880, 907, 925
 DVD (Digital Video Disk) 329, 584, 651, 653, 654, 896, 911

E

Early/static binding См. Раннее/статическое связывание
 EBCDIC (Extended Binary Coded Decimal Information Code) 58
 ECL, Lotus Notes/Domino 831
 EDSAC 136
 EFI (Extensible Firmware Interface) 881, 925
 ELF 169, 179, 320
 Embedded application См. Встраиваемое приложение
 Embedded system 19
 ENIAC 85—87, 85, 86, 87, 103, 136
 Error См. Ошибка
 ESDI 551
 Event flag См. Флаг события
 Exception См. Исключения, Исключительные ситуации
 EXE (DOS) 165
 Exec 158
 Explorer 200, 212, 643, 658, 716, 802, 828, 829, 830
 Exponent См. Порядок
 Extended attributes См. Расширенные атрибуты
 Extent См. Экстент

F

FAT (File Allocation Table) 205, 623, 654, 655—657, 661, 665, 669, 670, 673, 692, 698, 884
 FAT16 639, 656
 FAT32 212, 639, 656, 671, 682, 885
 Fault tolerance См. Устойчивость к сбоям
 FC-AL (Fiber Channel) 510, 528, 558, 559
 FDT, VMS 611
 FFS (Fast File System) 649, 661, 662, 664
 Fibers См. Волокна
 FIFO (First In — First Out) 330, 334, 346, 900
 File См. Файл
 Firefox 897
 FireWire 206, 498, 879
 First fit См. Первый подходящий
 Fixup table См. Таблица исправлений

Flash drive См. *Флэш-накопитель, Флэш-диск*
FLOPS (Floating Operations Per Second) 57
fnode 657
Folder См. *Папка*
Fork 158
Fork-процесс 592, 593, 596, 602, 610, 611
Fort 93
Frame См. *Фрейм, Кадр*
Frontend 426
fsdb 675
FUD (Fear, Uncertainty and Doubt) 898

G

Garbage collection См. *Сборка мусора* 254
Gate A20, PC 923
Gate 305, 923, 936, 937, 945
GDI (Graphical Device Interface) 299, 802
GDT (Global Descriptor Table) 306, 933, 936, 941
GDT, x86 933
GDTR, x86 933
Generational garbage collection См. *Генерационный сборщик мусора*
GNU (GNU Not Unix) 42, 66, 82, 115, 117, 137, 169, 170, 182, 218, 230—233, 241, 278, 410, 583, 644, 666, 722, 869, 870, 873, 875, 876, 982, 983, 986, 989, 991
GOT (Global Offset Table) 169, 171, 185, 186, 321, 322
GPL (General Public License) 584, 682, 697, 870, 871, 875, 876, 887
GRPS (General Packet Radio Service) 764
GRUB 208
Guard area См. *Сторожевая зона* 217

H

Handle 44, 144, 173, 259, 270, 571
Handler 356, 602, 733, 736, 737, 738, 743
Hard link См. *Жесткая связь*
Harvard Mark I 175
HBA (Host Bus Adapter) 476, 528, 529, 567, 575, 576, 579, 948, 949
Heap См. *Куча*
HMA 935
Honeynet 835

Hook 164, 234, 235, 237, 242, 247, 733
Hot plug См. *Горячее подключение*
Hotfix См. *Горячая замена*
HPFS (High Perfomance File System) 226, 639, 652, 657, 660, 661, 664, 665, 671, 676, 685, 901
Hub См. *Хаб*
HURD 870, 871, 876
HyperSCSI 560
Hyperthreading См. *Гипертрединг*
Hypervisor См. *Гипервизор*

I

iAPX 432 926
iBCS (Intel Binary Compatibility Standard) 731, 876, 947
IBM 3270 33
IBM ASSC 85
IDE (Integrated Drive Electronics) 26, 549, 550, 551, 552, 600, 602, 603
IDL (Interface Definition Language) 889
IDTR, x86 933
IDT6, x86 946
IE (Internet Explorer) 200, 295, 716, 802, 829, 830
IEEE 1342 498
IEEE 754 57
IFS (Installable File System) 580
IIS (Internet Information Service) 803, 819, 909, 910
Init 207, 208, 210, 218, 219, 234, 439, 569, 576, 865, 875, 879
I-node (Index Node) 661
Intention log См. *Журнальный файл*
Interrupt context См. *Контекст прерывания*
Interrupt handler См. *Обработчик прерывания*
Interrupt latency См. *Задержка прерывания*
Interrupt См. *Прерывание*
IOPL, x86 932
IORB, OS/2 602
IRC 836
IRF, Netware 777
IRP, VMS 611
IRQ (Interrupt ReQuest) 356, 603, 947, 948, 949, 950

ISA (Industry Standard Architecture) 208, 476, 478, 512, 551, 567, 616, 709, 923, 934, 948, 949, 950
 ISA (Instruction Set Architecture) 208, 476, 478, 512, 551, 567, 616, 709, 923, 934, 948, 949, 950
 iSCSI 560
 ISO 13346 См. UDF
 ISO 9660 См. CDFS
 Itanium 747, 846, 862, 922, 926
 iTunes 880

J

JFIF 67–68, 69
 JFS (Journal File System) 580, 676, 680, 681, 682, 684—688, 693, 699, 894, 896
 JIT (Just In Time) 93, 156, 862
 JIT-компиляция 93, 156
 Job См. Задание
 Joliet 652, 653, 682

K

Kernel context См. Контекст ядра
 Kernel debugger См. Отладчик ядра
 Kernel panic 708
 Keylogging 760, 812

L

LAN Manager 32, 890
 Latch register См. Регистр-защелка
 Late/dynamic binding См.
 Позднее/динамическое связывание
 LATEX 43, 563, 643
 Lazy write См. Отложенная запись
 LCD (Liquid Crystall Display) 133
 ld (Link eDitor) 110, 114, 321, 326, 433
 LDT (Local Descriptor Table) 306, 427, 440, 936, 939, 944, 945
 LDTR, x86 933
 LIFO 111
 Lilith 190
 Link editor См. Редактор связей
 Link См. Линк
 Linker См. Редактор связей
 Linking См. Сборка

Linux 19, 23—27, 38, 42, 44, 80, 82, 179, 205—207, 211, 214, 218, 249, 250—252, 298, 299, 307, 326, 327, 345, 363, 427, 430, 439, 444, 572, 581, 583, 584, 587, 603, 619, 633, 638, 677, 680—682, 687, 689, 690, 697, 754, 819, 820, 832, 867, 870—876, 886, 887, 893—899, 912, 922, 924, 925, 938, 944, 947
 Little-endian 105, 291, 292, 935
 Log 158, 159, 226, 237—239, 249, 684, 716, 717, 724, 767
 Login 767, 806, 814
 Longjmp 742
 Losing compression См. Упаковка
 с потерями
 Lotus Notes 342, 401, 787, 790, 814, 831, 890
 LRU (Least Recently Used) 332, 333
 LSI-11 (Large Scheme Integrated-11) 852, 860
 LVM (Logical Volume Manager) 633, 682, 690, 693
 LWP (Light Weight Process) 430, 456, 465, 873
 LX (Linear Executable) 645

M

MacOS 20, 29, 38, 270, 646, 647, 791, 869, 877—880, 925
 MACRO 847
 MacroMedia Flash 62, 831, 897
 Magic number См. Магическое число
 Mailbox См. Почтовый ящик
 Malware 585, 792, 840, 897, 905, 907
 Mantiss См. Мантисса
 Mark and sweep См. Просмотр ссылок
 Master См. Ведущее устройство,
 Инициатор
 MBR (Master Boot Record) 208, 209, 211
 MC680x0 106, 108, 110, 111, 114, 121
 MCB 267
 MCR (Monitor Control Routine) 856
 Memory leak См. Утечка памяти 252
 MESI 365
 MFT, NTFS 667
 Microkernel См. Микроядро
 MicroVAX 860

MIDI (Musical Instrument Digital Interface)

64

Minix 38, 869, 870, 876, 899

Minor node См. Минорная запись

MIPS (Microprocessor without Interlocked

Pipeline Stages) 105, 132, 181, 182, 335,
861, 876, 902

MMU (Memory Management Unit) 283, 293

MMX 931

Monitor См. Монитор

MP3 67, 826

MPEG2 67

MPEG3 67

MS DOS 24, 265, 267, 269, 398, 563, 566,
609, 622, 625, 637, 641, 654, 655, 656,
661, 675, 728, 736, 866, 883—887, 890,
924

MS Office 42, 199, 200, 320, 454, 714, 720,
792, 796, 802, 810, 820, 827, 897, 908,
909, 916

MSX 125

Multics 410, 814, 820, 863, 864

Multimedia 37

Multiprocessing См. Многопроцессность

Multithreading См. Многопоточность

Mutex См. Мутекс

Mutual exclusion См. Взаимоисключение

MVS 25, 27, 38, 45, 267, 642, 766, 843, 844,
875, 890

MVS (Multiple Virtual Storage) 25, 27, 38,
45, 843, 844, 875, 890

MVT (Multiprogramming with Variable
number of Tasks) 843, 844

N

N9000 190—192, 638

Nice level 455, 456, 459

Nimda 810

Novell NetWare 20, 671

NP-полная задача 453, 702

NTFS (NT File System) 212, 639, 667, 677,
680—682, 685, 699, 866, 901

NTLDR 211

NuBus 567, 878

NUMA (Non-Uniform Memory Access)
287, 369, 370, 372, 409, 508, 925

O

Occam 432

OpenGL (Open Graphics Library/Language)
538, 880

OpenOffice 42, 897, 899

OpenVMS 44, 745, 747, 846, 860, 862, 926

OS/2 23—27, 39, 42, 45, 82, 189, 193, 199,
205, 214, 306, 307, 316—318, 322, 325,
335, 395, 420, 426, 456, 464, 468, 563—
568, 574, 578—580, 583, 600, 602, 628,
633, 635, 637, 639, 641, 645—647, 652,
653, 655, 657, 661, 671, 673, 677, 680—
682, 685, 687, 689, 742—744, 748, 754,
814, 845, 854, 866, 886, 890—902, 906,
908, 914, 919, 921, 922, 936, 938, 940,
941, 982

OS/390 13, 26, 38, 44, 455, 464, 766, 843,
844, 845, 872, 890

OS/SVS (OS/Single Virtual Storage) 844

Overcommit См. Оптимистическая
аллокация

Overflow См. Переполнение

Overlay manager См. Менеджер оверлеев

Overlay См. Оверлей

P

Page fault См. Страницочный отказ

Palm OS 19, 21, 29

PAM (Pluggable Authentication Module) 766,
785

Parity bit См. Бит четности

Partition См. Раздел диска

Password См. Пароль

Patch 803, 840

Path name См. Путевое имя

PCI (Peripheral Component

Interconnect/Interface) 206, 208, 301, 368,
476, 478, 486, 502, 510, 511—514, 523—
526, 567, 584, 585, 614, 832, 874, 879,
923, 934, 948, 949

PCL (Printer Control Language) 62, 133,
352

PCP (Primary Control Program) 843

PC-совместимые компьютеры 923

PDF (Portable Document Format) 62, 880

PDP-11 87, 88, 90, 104—108, 111, 113, 114, 122, 128, 129, 200, 283, 290, 356, 357, 476, 547, 567, 631, 721, 842, 846, 851—858, 860, 864, 867, 930, 956, 988

PDP-endian 105

PE (Portable Executable) 318, 645, 824, 914, 915, 975

Perl 155, 194, 284, 705, 807

Phishing 812, 836

phpBB Santy 807

PIC (Peripheral Interface Controller) 13, 92, 94, 100, 104, 108, 125, 133, 169, 170, 321, 350, 358, 492, 539

PIP (Peripheral Interface Program) 847, 850, 854, 856

Pipe См. Труба

Pipeline См. Конвеер

PLT (Procedure Linkage Table) 169, 171, 172, 185, 186, 321—324, 379, 800, 801

Plug'n'Play 208, 949

Polling См. Опрос

Pool См. Пул

POSIX (Portable OS Interface [based on] uniX) 44, 403, 412, 413, 463, 465, 622, 623, 639, 641, 662, 685, 712, 730, 731, 766, 866, 871, 876

Postmortem dump См. Посмертный дамп

PostScript 61, 563, 573, 583, 827, 878, 880

PowerPC 22, 47, 105, 129, 182, 274, 567, 682, 895, 902

Process См. Процесс

Protection fault См. Исключение по защите памяти

PSW 96

Public domain 43, 437, 789, 792

Q

QNX 14, 19, 37, 38, 39, 42, 464, 468, 585, 869

Quota См. Квота

R

Race condition См. Ошибка соревнования

RADIX-50 858

RAID (Redundant Array of Inexpensive Disks) 352, 355, 527, 555, 556—558, 696, 900

Read/write lock См. Блокировки чтения/записи

Reentrant См. Рекурентность

Reference counting См. Подсчет ссылок

Remembered set 261, 279

Reset См. Сброс

Resource fork См. Ветвь ресурсов

REXX 194, 195, 727, 748

RISC (Reduced/Rational Instruction Set Computer) 90, 92, 96, 100; 106—108, 129—132, 169, 182, 191, 283, 306, 449, 860, 861, 873, 878, 895, 902, 918, 926

RLE (Run-Length Encoding) 67

RLL 541

RMS 304

Rockridge 652, 682

Rollback segment См. Сегмент отката

Rollback См. Откат транзакции

ROM (Read-Only Memory) 201, 514, 527, 543, 552, 579, 580, 653, 710, 911

Rootkit 207, 826

RS232 486, 487, 488, 492, 496, 505, 564, 571, 577, 582

RS232 (Recommended Standard 232) 19, 104, 486, 487, 488, 492, 496, 498, 499, 505, 564, 571, 577, 582, 847, 923, 948

RSA (Rivest/Shamir/Adleman) 74, 77, 790

RSX-11 39, 45, 205, 397, 406, 417, 467, 622, 635, 637, 639, 642, 643, 644, 845, 846, 854—859, 863, 882

RSX-15 854, 855, 857

RT-11 39, 45, 160, 166, 169, 172, 216, 467, 610, 622, 634, 635, 637, 644, 648, 650, 651, 845, 846, 851, 852, 854, 855, 882

RTTI (RunTime Type Information) 753, 879

S

SAN (Storage Access Network) 559, 560, 682, 697

Sandbox См. Песочница

Sapphire 770, 834, 840

SAS (Serially Attached SCSI) 486, 528, 553

SATA (Serial ATA) 201, 486, 552, 553, 896

S-Bus 478

SCO OpenDesktop 574

Script injection См. Внедрение скрипта

Script См. Скрипт

- SCSI (Small Computer System Interface) 416, 476, 486, 498, 505, 506, 510, 513, 514, 527, 528, 529, 546, 552, 553, 559, 560, 561, 567, 575, 576, 579, 580, 614, 618, 879, 948, 949
- Security context См. *Контекст доступа*
- SEH, Win32 753
- Select 413
- Sendmail 428, 818, 820, 833, 897
- Sequent 873, 925
- Service 23, 304, 458, 756, 775, 792, 794, 810, 844, 905
- Setjmp 742
- Setuid 785, 788—792, 819, 825
- Setuid, Unix 788
- SGI Irix 38, 468, 574, 873
- Shared time См. *Разделенное время*
- Shrink-wrap software 757
- Shutdown См. *Останов системы*
- Signal 733, 736, 738, 741, 964
- Signature См. *Сигнатура*
- Significant См. *Мантисса*
- Simula 67 435, 750
- Slashdotting 758
- Slave См. *Ведомое устройство, Целевое устройство*
- Slice 209, 632
- SMP (Symmetric MultiProcessing) 355, 439, 446, 447
- SNA 844
- Snapshot См. *Моментальный снимок*
- Sniffer 569
- Socket 115, 146, 194—196, 410
- Soft real time См. *Мягкое реальное время*
- SOM (System Object Model) 894, 896
- SPARC 34, 42, 82, 87, 100, 102, 104, 105, 108—110, 114, 115, 119, 120, 121, 124, 129, 132, 153, 163, 165, 181, 182, 209, 290, 291, 293, 295, 296, 300, 301, 324, 326, 359, 367, 385, 444, 449, 477, 511, 567, 667, 721, 860, 874, 876, 935
- Spectrum 20, 28, 340
- Spinlock 384
- Spooling 627
- Spyware 207
- SQL (Sequential Query Language) 32, 320, 362, 401, 402, 404, 805—807, 814, 817, 834, 842, 906
- SQL injection См. *Внедрение SQL*
- SQL Slammer 834
- SSE 931
- ST506 549
- Stack unwind 751
- State machine См. *Конечный автомат*
- Stealth-вирус 823
- Sticky bit 665
- Sudo, Unix 790
- Sun Solaris 27, 38, 44, 208, 296, 460, 468, 569, 574, 680, 872, 874, 947
- Superblock См. *Суперблок*
- Supervisor См. *Супервизор*
- Swapping См. *Свопинг*
- Switch 124, 130, 159, 242, 419, 421—423, 439, 447
- Symbian 19
- Symbolic link См. *Символическая связь*
- Syslog 718
- System/360 843, 844, 851

Т

- Task См. *Процесс*
- TCO (Total Cost of Ownership) 35
- TCP/IP 43, 104, 408, 560, 574, 814, 845, 854, 859, 867, 876, 894, 903, 915
- TENEX 845, 846, 849, 850, 851, 854, 856
- Terminal Server 34, 767
- TeX 43, 623
- TFT (Thin-Film Transistor) 535
- Thread queue См. *Очередь нитей*
- Thread 158, 170, 380, 427, 432—435, 439, 463—465, 712, 866, 876, 944
- Thread-safe/unsafe 380
- Thunderbird 897
- Tiled memory 891, 939
- Time slicing 437
- TLB (Translation Lookaside Buffer) 284, 286, 288, 290, 293—298, 934, 962
- TR, x86 933
- Transaction См. *Транзакция*
- Transputer 93, 132, 172, 191, 406, 413
- Troyan См. *Троянская программа*
- Tru64 38
- TSO (TimeSharing Option) 843
- TSR-программа 268

TSS, x86 933
TSX-11 854

U

UAC, Windows Vista 792
UART (Universal Asynchronous Receiver Transmitter) 488
UDF (Universal Disk Format) 653, 654, 896
UFS (Unix File System) 649, 650, 661, 676, 677, 680, 684, 694, 874
uid, Unix 765
UMA (Uniform Memory Access) 369
UNC-имя 635
Underflow 738, 740
Unibus 476, 567
Unicode 58, 639, 652, 687
Unix 13, 23, 25, 28, 31, 34, 38, 39, 42—44, 82, 138, 157—159, 160, 169, 179, 181, 187, 204—208, 215, 217, 222, 230, 256, 288, 296, 299, 301, 307, 316—320, 325, 326, 332—339, 343, 347, 398, 403, 406—413, 427—429, 434, 437, 455, 456, 459, 460, 464, 465, 468, 563, 564, 567—583, 608—612, 617, 618, 622, 623, 628, 629, 632, 635—639, 641—649, 652, 655, 657, 659, 661—667, 671, 674, 675, 681, 685, 687—689, 694, 699, 707, 708, 718, 722, 727, 730—735, 737, 742, 744, 754, 765—768, 771, 772, 780—782, 785—790, 793, 803, 807, 809, 810, 814, 819, 822, 825, 833, 841, 845, 859, 861, 862—880, 883, 884, 894, 899, 902, 921, 924, 925, 926, 947, 982
UnixWare 25, 468, 469, 574, 677, 689, 872, 921, 925, 947
Upper memory 935
USART (Universal Synchronous—Asynchronous Receiver Transmitter) 492—496
USB (Universal Serial Bus) 204, 206, 486, 497, 498, 499, 501—503, 506, 512, 513, 561, 577, 585, 879, 896, 903, 931, 950
USB OTG 499
User context См. Пользовательский контекст
User ID См. Идентификатор пользователя
UTF-8 638

V

VAX 31, 39, 87, 90, 92, 104—114, 121, 122, 129, 130, 192, 200, 288, 301—305, 310, 316, 318, 330, 332, 334, 346, 385, 413, 417, 444, 449, 451, 455, 457, 464, 467, 567, 593, 611, 622, 644, 652, 667, 780, 803, 833, 842, 846, 851, 852, 857—863, 866, 867, 882, 901, 902, 956, 988
VAX/VMS 39, 301, 305, 316, 318, 330, 332, 334, 346, 417, 444, 451, 455, 457, 464, 611, 622, 644, 652, 667, 846, 857—863, 866, 882, 901, 902
Veritas 664, 677, 682, 689
Version control system См. Системы управления версиями
Virtuozzo 25, 36
Visual Basic 255, 265, 705, 748
VM 25, 26, 261
VM/ESA 26
VMT (Virtual Method Table) 81
Volume См. Том
VSAM, MVS 844
VxWorks 14, 37, 39, 189, 200, 413

W

WAFL (Write Anywhere File Layer) 694, 695, 696
Watchdog timer См. Сторожевой таймер
Web-хостинг 23
Well-known DLL 320
WFS (Windows File System) 905, 906
Win32s 645, 915
Window fill См. Заполнение окна
Windows, ОС семейства 13, 14, 19—26, 29, 31, 38—44, 62, 189, 193, 199, 206, 207, 210—213, 272, 299, 306, 307, 316—320, 325, 326, 330, 332, 335, 339, 346, 430, 435, 456, 558, 563—568, 574, 583, 584, 585, 622, 625, 627, 635, 637, 639, 641, 643, 645, 652, 653, 656—658, 667, 671, 677, 680, 681, 690, 708, 714—716, 720, 736, 748, 757, 758, 767, 772, 775, 791—796, 801—803, 810, 814, 819, 820, 821, 824—835, 854, 862, 866, 877, 881, 885—888, 893, 895, 899, 900—915, 921—925, 936, 938—940

Windows Genuine Software Advantage 904
 WMF (Windows MetaFile) 62
Worm См. *Червь*
Worst fit См. *Наименее подходящий*
 WPS (WorkPlace Shell) 647
Write anywhere См. *Запись повсюду*

X

X Window 34, 299, 426, 573, 583, 859, 871,
 900
 x64 874
 x86 22, 24, 25, 33, 39, 47, 52, 82, 83, 85,
 87—92, 97—107, 113, 114, 117, 129, 147,
 150, 169, 178, 192, 199, 208—215, 228,
 283, 288, 289, 295, 298, 299, 301, 307,
 326, 346, 355, 358, 370, 384, 385, 393,

427, 434, 437, 439, 442, 468, 476,
 510—514, 526, 552, 565—568, 656, 667,
 721, 738, 800, 801, 832, 842, 845, 860,
 862, 870—876, 881, 891, 893, 895, 902,
 905, 911—930, 937, 943—947, 950—952,
 956, 982, 983, 989

Xbox 585, 832, 902, 911, 912, 913, 925

XBox Linux Challenge 832

Xenix 316, 318, 632, 664, 867, 868, 872,
 883, 884, 919, 921

Z

z/OS 13, 23, 25, 27, 38, 42, 363, 642, 843,
 845

ZFS 696, 697, 875

Zombie 865

A

Абсолютная адресация 110, 122, 163, 954,
 956
 Абсолютная загрузка 160, 161, 162, 336
 Абсолютный загрузочный модуль 160
 Аварийное завершение 138, 708, 711, 712,
 713, 714, 754
 Аварийный останов 708, 714, 720, 725
 Автоконфигурация 514, 567, 569, 613,
 614, 615, 949
 Автокорреляция 65—69, 74, 75
 Автомат общего вида 600
 Авторизация 17, 765, 772, 780, 783, 787,
 790, 791, 802, 819, 829
 Администратор данных 401, 786, 816
 Администратор учетных записей 786, 788
 Адрес 16, 21, 45, 81, 82, 84, 86, 87, 90, 91,
 92, 96, 97, 101—114, 120—129, 136, 137,
 139, 142, 146, 147, 151, 155, 156, 157,
 160—175, 178, 179, 184—187, 191, 192,
 193, 200, 208, 211, 215—218, 223, 225,
 227—233, 242, 244, 247, 248, 266, 267,
 270—277, 282—288, 292—323, 336, 339,
 340, 341, 354, 356, 357, 359, 366, 393,
 413, 414, 433, 439, 442, 475—477, 479,
 481, 488, 496, 500—503, 506, 508,

510—514, 523—528, 538, 548—552, 567,
 572, 608, 611, 614—616, 632, 641, 650,
 655, 715, 721, 741, 743, 744, 746, 758,
 799, 800, 803, 813, 822, 828, 829, 833—
 835, 844, 848, 849, 857, 864, 884, 887,
 891, 903, 905, 923, 931—947, 951, 952,
 955, 956, 963, 975, 985—989, 992

Адресация 21, 87, 88, 97, 102—111, 114,
 120—125, 129, 163—169, 172, 192,
 273—277, 279, 282, 283, 286, 294, 298,
 315, 339, 340, 343, 345, 475—477, 513,
 527, 528, 546, 552, 561, 605, 614, 648,
 649, 804, 848, 851, 852, 857, 864, 882,
 885, 919, 921, 922, 924, 935, 951, 952,
 954—957, 988

Адресное поле 103—111, 114, 123, 124,
 137, 156, 160, 163, 165, 168, 846, 955,
 956, 983, 989

Адресное пространство 21, 103, 104, 106,
 110, 124—129, 148, 157, 158, 160, 169,
 172, 173, 175, 192, 198, 200, 215—219,
 223, 230, 265, 272—277, 282, 283, 286,
 287, 289, 290—310, 314—321, 325, 327,
 330—332, 337—339, 354, 356, 357, 366,
 413, 430, 442, 464, 465, 466, 475—477,
 502, 512—514, 538, 561, 563, 572, 581,
 583, 594, 608—613, 617, 619, 620, 621,

- 633, 641, 649, 651, 656, 707, 722, 781, 783, 784, 787, 793, 834, 843, 844, 848, 849, 852, 856—860, 863—865, 867, 873, 885, 888, 890, 891, 894, 920, 934—939, 941, 943
- Адресное пространство ввода/вывода 475, 476, 502, 538, 561, 619, 934, 935
- Адресный регистр 98, 125, 163, 165
- Азбука Морзе 65
- Аккумулятор 21, 85, 86, 91, 92, 97, 98, 350
- Аккумуляторная архитектура 92
- Активная матрица 535
- Активный контент 827, 828, 829, 832
- Алгоритм
- ◊ близнецов 231, 247, 248, 249, 279
 - ◊ Деккера 381, 383, 429
 - ◊ парных меток 228, 230, 231, 232, 233, 278
 - ◊ Лемпеля-Зиффа 66
- Аллокатор 220, 223, 225, 226, 249, 250, 251, 260
- АЛУ (Арифметико-Логическое Устройство) 47, 49, 50, 52, 96—104, 129, 362
- Анти-алиасинг 62
- Апплет Java 716
- Арбитр шины 368, 369, 370, 523, 524, 525
- Арбитраж шины 387
- Арифметические флаги 93
- Арифметический регистр 97, 98, 99, 928, 929
- Архивные библиотеки 187
- Архитектура
- ◊ память-память 106
 - ◊ регистр-память 106, 919
 - ◊ регистр-регистр 106, 153
- Асимметричная многоточечность 406
- Асинхронная обработка сообщений 426
- Асинхронная передача 483, 484
- Асинхронное взаимодействие 378
- Асинхронные исключительные ситуации 706
- Асинхронный ввод/вывод 467, 610, 622, 852, 853, 854, 855
- Ассемблер 14, 79, 80—83, 88, 94, 98, 102, 132—136, 137, 138, 139, 142, 149, 151—153, 156, 164, 165, 168, 178, 200, 301, 393, 410, 432, 443, 453, 644, 742, 847, 853, 854, 858, 859, 879, 927, 982—989, 991
- Ассоциативная память 284, 294, 366
- Атака затоплением 758
- Аутентификация 17, 78, 759, 763—767, 770—772, 775, 784—786, 790, 791, 841
- Аутсорсинг 816
- АЦП (Аналогово-Цифровой Преобразователь) 56, 64, 931
- ## Б
- Базовая адресация 123, 167—169, 273—279, 315, 345, 864, 882
- Базовая виртуальная адресация 273
- Базово-индексная адресация 120, 121, 955
- Байт 104
- Байт-код 93, 192, 322
- Банк памяти 124—129
- Банки памяти 95, 124—126, 304, 513, 593
- Банковая адресация См. Банки памяти
- Безадресная команда 90, 106, 113
- Бесконечный автомат 596, 703
- Библиотека объектных модулей 186
- Бинарная совместимость 83, 101, 129, 359, 442, 862, 884, 890, 895
- Бит 47, 48—52, 56, 58, 63, 64, 66, 70—77, 84, 87, 90, 94, 98, 99, 103—110, 124, 125, 128, 133, 146, 248, 266, 287, 293—296, 298, 301—308, 330, 332—334, 339, 343, 349, 367, 478—482, 485, 487, 488, 489, . 490, 493—495, 499, 502—506, 510, 512, 513, 524, 526, 529, 537, 538, 541, 542, 546—552, 571, 656—659, 665, 684, 686, 696, 704, 721, 722, 731, 734, 736, 737, 744, 746, 771, 778, 780, 781, 790, 801, 825, 846—851, 891, 902, 905, 919, 928, 931—937, 942—944, 947, 950, 951, 955—957, 968
- Бит переноса 48
- Бит четности 70—72, 493, 494, 722, 932
- Битовая карта свободных блоков 654, 661
- Битовая плоскость 537
- Блок диска 544
- Блочные устройства, Unix 570
- Большая сборка мусора 260
- Бот 207, 716, 762, 800, 813, 833, 835—839, 905

Ботнет 762
 Бутстррап 202, 212
 Буферизованная передача 407
 Быстрые смежные транзакции (PCI) 525
 БЭСМ-6 57, 97, 103, 104, 438, 452

В

Ведомое устройство 497, 499, 523, 525
 Ведомый 483, 494, 497
 Ведущее устройство 499, 523—525
 Ведущий 483, 494
 Вектор прерывания 356
 Векторное изображение 60, 61, 62, 878
 Ветвь данных 646
 Ветвь ресурсов 646, 647
 Взаимонедоверяющие подсистемы 308, 311—314, 346, 926
 Взаимоисключение 379, 381—386, 389, 391, 392, 398, 403, 405, 406, 467, 559, 564, 602, 865
 Видеобуфер 299, 476, 536, 537, 538, 561, 934
 Видеопамять 299, 538, 583
 Виртуальная адресация 103
 Виртуальная память 21
 Виртуальное адресное пространство 286
 Виртуальные линки 413
 Виртуальный 8086, x86 888, 937, 914
 Вирус 149, 169, 201—204, 757, 762, 795, 796, 803, 810—813, 819, 822, 823—829, 836, 837, 880, 897, 909, 910
 Висячая ссылка 253, 705
 Внедрение SQL 805, 807, 817, 842
 Внедрение скрипта 805, 807, 819, 821
 Внешнее устройство 19, 23—25, 125, 126, 165, 206, 277, 295, 357, 361, 365, 368, 378, 391, 404, 410, 413, 416, 426, 438, 468—478, 480, 482, 509, 563, 564, 586, 597, 616, 619, 627, 631, 632, 641, 781, 884, 901, 914, 923, 947
 Внешние устройства 473
 Внешняя фрагментация 221, 224, 225, 265, 267, 278, 281, 287, 310, 942
 Внутренняя фрагментация 221, 224, 278, 682
 Внутрисхемная перепрошивка 18
 Внутрисхемный эмулятор 18

Возраст 457
 Волокна 430, 464
 Встраиваемое приложение 15, 18, 19, 30, 47, 258, 273, 274, 314, 351, 358, 468, 498, 531, 538, 709, 852, 861, 863, 918, 923, 926
 Вторичный загрузчик 202, 204, 205, 208—214, 477, 514, 874, 885, 887, 896, 924, 925
 Выборы 505
 Выполнение транзакции 404
 Высокоимпедансное состояние 481
 Вытесняющая многозадачность 29, 30, 437, 438, 466, 468, 470, 471, 564, 566, 569, 723, 843, 844, 864, 877, 884, 890, 939
 Вэйвлет 68

Г

Гарвардская архитектура 103, 110, 169, 175
 Гармонически взаимодействующие последовательные потоки 405
 Генератор событий 474, 573, 574
 Генераторы событий 474, 573
 Генерационная сборка мусора 259
 Генерационный сборщик мусора 259, 261, 264
 Генерация системы 205
 Географическая адресация 477, 513, 561
 Гибкий диск 202, 208, 541, 543, 547, 566, 589, 593, 635, 691, 882, 914, 948
 Гипервизор 274, 927
 Гиперкубическая топология 371
 Гипертрединг 361, 362, 363, 921
 Глобальный символ 178, 179, 184, 186, 193, 984
 Глобальный флаг события 397
 Голодание по записи 399
 Голодание по чтению 399
 Горячая замена 206, 668, 692, 693, 844, 874
 Горячее подключение 510, 879, 903
 Граф состояний 600
 Графический ускоритель 352, 355, 911

Д

Двоично-десятичная коррекция 52, 129
 Двоично-десятичное представление 52

- Двоично-десятичные числа 52
Двоичное дополнение 49—52
Двоичный семафор 395
Двоичный семафор Дейкстры 394
Двухадресная команда 90
Двухпроходное ассемблирование 137
Двухточечные примитивы
 взаимодействия 406
Деассемблер 88, 90, 138, 149, 150—152,
 814
Деассемблирование 149—151
Демон 207, 389, 461, 628, 718, 734, 879
Денормализация числа с плавающей
 точкой 57
Дескриптор
 ◊ нити 432
 ◊ сегмента 129, 296, 310, 427, 919, 920,
 933
 ◊ страницы 286, 294, 296, 297, 302, 333,
 934, 942, 943
 ◊ файла 640
Дефрагментация 221, 260, 263, 264, 271,
 275, 310, 311, 651, 661, 685
Дефрагментация памяти 222
Диаграмма переходов, конечный автомат
 597
Динамическая загрузка См. *Динамическая
 сборка и Сборка в момент загрузки*
Динамическая сборка 192, 206, 269, 884,
 915, 983,
Динамические библиотеки 193, 194, 801,
 872
Динамическое выделение памяти 220,
 222, 230
Динамическое связывание 155
Директивы условной компиляции 135,
 142, 724, 725, 818, 985, 991
Директивы; ассемблер 989
Директория 632, 675
Диск 16, 18, 20, 22, 26, 32, 139, 174,
 201—214, 270, 275, 281, 286, 288, 289,
 310, 311, 314, 319, 327, 329—340, 345,
 347, 353, 359, 360, 410, 450, 452, 473,
 477, 513, 527, 539, 540—570, 579, 580,
 581, 585, 589, 603, 623—628, 631, 632,
 635, 637, 638, 641, 648, 650—662, 664,
 667—700, 708, 731, 736, 737, 760, 796,
 823, 826, 827, 832, 846, 847, 850, 853,
 855, 859, 864, 874, 875, 882, 888, 896,
 905, 915, 924
Дискета 202—204, 329, 473, 540, 544, 545,
 548, 566, 625, 634, 638, 655, 669, 670,
 691, 736, 790, 820—823, 887, 914
Дисковая операционная система См. *ДОС*
Дисковод 357, 377, 571, 589, 625, 634, 655,
 662, 668—670, 691, 736
Дисковый контроллер 20, 32, 201, 212,
 335, 352, 355, 505, 528, 546, 547, 554,
 567, 568, 623, 635, 691
Дисковый кэш 330—332, 466, 579, 612,
 616, 623, 624—626, 656, 682, 688, 708,
 901
Дисковый массив 555
Дисковый накопитель 497, 539, 553, 555
Дисплак 452
Диспетчер задач См. *Планировщик*
Диспетчер памяти 21—26, 97, 99, 158,
 215, 216, 225, 267, 268, 271, 274, 282—
 293, 297, 298, 300, 303, 309, 311—315,
 332, 333, 339, 341, 346, 354, 357, 359,
 439, 449, 464, 582, 583, 593, 612, 721,
 723, 780, 801, 843, 844, 846, 850—852,
 855—858, 864, 868, 876, 882, 926—928,
 933, 934
Диспетчер системных вызовов 23, 276,
 469, 819, 826
Дисплей 23, 62, 474, 531—539, 585, 760,
 878, 880, 931
Домен i432 312
Дорожка диска 540, 544—550, 554
ДОС 16, 19—20, 21, 22, 25, 28—30, 39, 40
ДОС (Дисковая Операционная Система)
 16, 20—30, 37, 39, 40, 147, 215, 631, 634,
 638, 661, 667, 668, 673, 708, 710, 727,
 728, 823, 852, 877, 882, 883, 914, 930
Драйвер 26, 133, 167, 169, 203—214, 249,
 299, 301, 404, 417, 468—470, 501—504,
 513, 552, 557, 563—619, 624, 628, 629,
 632, 633, 644, 653, 657, 669, 675, 682,
 684, 725, 801, 804, 849, 854, 859, 868,
 874, 879, 881, 882, 884—887, 890—893,
 896, 903, 907, 914, 922, 925, 935, 939,
 948, 949
Драйвер файловой системы 205, 417, 580,
 633, 682, 725

Е

Единица обслуживания, MVS 844

Ж

Жертва 329

Жесткая связь 665, 667, 685, 699, 866
 Жесткий диск 17, 26, 192, 201—204, 208,
 211, 212, 338—340, 355, 375, 428, 474,
 481, 527, 540—546, 549, 551—554, 566,
 567, 570, 579—581, 635, 649, 651, 653,
 655, 667, 691—693, 758, 823, 825, 832,
 911, 915, 923, 925

Живая блокировка 385, 387, 388

Жидкокристаллический дисплей (ЖКД)
 531, 533—535

Журнал регистрации намерений 678

Журнализование 677, 680, 681, 699, 700

Журнальная ФС 677

Журнальный файл 675, 677

З

Зависание 29, 253, 461, 661, 668, 709, 712,
 737

Загрузка по сети 204, 514, 667

Загрузка программы 16, 28, 157, 172, 189,
 192, 324, 336, 645, 985, 987

Загрузочная запись 208, 657

Загрузочное ПЗУ 874, 925

Загрузочный вирус 201—204, 823

Загрузочный модуль 156

Загрузочный монитор 20, 28, 160, 200—
 202, 614

Загрузочный сектор 202—204, 209, 211,
 800, 823, 824

Загрузчик 17—20, 37, 136, 153, 157, 165,
 167, 169, 193, 202, 205, 209—212, 216,
 224, 286, 318—320, 339, 514, 631, 645,
 800, 832, 855, 881, 883, 924, 925, 984, 986

Задание 158

Задатчик шины 354, 368, 369, 505, 508,
 526

Задача 15—31, 37, 39, 40, 43, 157

Задержка прерывания 358, 443, 501, 615,
 723, 918

Задние двери 817

Закладка 815

Закладка Томсона 814

Закон Мура 364, 696

Закон Парето 328

Замки, MVS 844

Запись повсюду 693, 694, 697, 700

Заполнение окна 119

Запоминающее устройство 65, 84, 96, 474,
 498, 527, 539, 555, 558, 560, 561, 573,
 574, 579, 648, 649, 696, 717, 722, 791,
 825, 906

Запоминающее устройство прямого
 доступа 474, 539, 574, 579

Запрос прерывания 356

Захват

◊ для записи, чтения 398

◊ файла для записи 403

◊ файла для чтения 398, 403

Защелка 481

Зеркалирование 556

Зомби 813, 865

И

Идентификатор пользователя 765—767,
 773, 784, 787—789, 855, 902

Иерархическое имя 698

Изохронная передача 484, 500, 501

Именованные трубы 412

Инверсия приоритета 396, 454, 461, 462—
 464, 470, 471

Индексный регистр 97, 99, 121, 988

Инициатор 338, 505, 506, 523—528, 559

Инициатор (SCSI) 528

Инкрементальный сборщик мусора 264,
 265

Инод (индексная запись) 632, 657, 661,
 662—665, 672—676, 680, 684—689,
 693—695, 699

Интерпретатор 79, 93, 156, 158, 192, 194,
 200, 265, 272, 321—323, 379, 410, 563,
 583, 646, 747, 748, 790, 803, 806, 820,
 832, 833, 847, 849, 850—853, 856, 879,
 880, 883, 884

Иключение 35, 83, 92, 100, 120, 125, 146,
 178, 274, 285, 286, 291—295, 304, 310,
 346, 356, 359, 360, 379, 380, 385, 392,
 395, 439, 449, 473, 486, 557, 614, 617,

652, 685, 697, 705, 721, 727, 733, 734, 737, 738, 740—747, 750, 753, 754, 775, 795, 802, 803, 846, 874, 879, 890, 919, 930—935, 938, 943, 945—947, 950, 955, 956, 976, 987
 Исключительная ситуация 104, 274, 359, 701, 704, 706, 707
 Исполняемый файл 645, 667, 822

K

Кадр 37, 67, 114—119, 433, 434, 483, 484, 487, 492, 494, 495, 500, 503, 504, 523, 528, 531, 536, 544, 560, 741, 742, 746, 747, 754, 799, 800, 802, 803, 956, 964
 Кадровый гасящий импульс 532
 Канальный процессор 27, 352, 353, 355, 358
 Каталог 199, 211, 326, 340, 430, 559, 566, 623, 628, 632—639, 646, 648, 650—670, 673—676, 682, 685, 687, 688, 692, 694, 698, 699, 735, 775—782, 808—810, 822, 825—827, 832, 838, 842, 844, 850, 853, 855, 856, 863, 866, 874, 884, 903, 904, 920, 930, 934, 941, 943
 Квант времени 30, 437, 438, 452—456
 Квитирование 408
 Квота 332, 341, 458, 697, 699, 703, 723, 734, 757, 781, 793, 794, 841, 901
 Кейлоггер 760
 Класс планирования 438, 456, 462, 465, 468, 583, 585, 785, 848, 855, 872, 876, 879
 Кластер 27, 243, 648, 649, 650, 655, 656, 657, 661, 673, 692, 697, 844
 Ключ шифрования 73, 74—78, 161, 584, 907
 Когерентность кэша 365, 366
 Код команды 85, 87, 88, 101, 107, 122, 129, 132, 135, 136, 163, 524, 733, 798, 846, 920, 930, 951, 952, 955, 957, 976
 Код
 ◊ ошибки 730
 ◊ Рида-Соломона 72, 722
 ◊ символа 58, 133
 ◊ Хаффмана и Шеннона-Фано 66
 ◊ Хэмминга 71, 72, 557, 691, 722
 Кодировка 58, 59, 60, 65, 108, 136, 180, 343, 637, 638, 652, 687, 691, 858, 984

КОИ (Код Обмена Информацией) 59
 Кольца доступа 841, 863, 936
 Кольца защиты 301, 308, 311, 919
 Коммутатор 370, 372, 463, 498, 499, 507, 508, 511, 528, 559
 Коммутационная панель 85, 86
 Компакт-диск 204, 353, 579, 651, 652, 722, 760, 820, 826, 859
 Компилятор 18, 54, 80, 102, 130—132, 135, 137, 139, 141—143, 152, 156, 160, 164, 169, 170, 173, 174, 178, 179, 198, 199, 267, 271, 272, 289, 292, 320, 321, 328, 362, 410, 434, 436, 443, 619, 623, 643, 644, 703, 722, 724, 739, 753, 754, 800, 806, 814, 815, 847, 862, 870, 873, 881, 886, 891, 892, 915, 919, 926, 944, 956, 982
 Компоновка См. Сборка
 Компоновщик См. Редактор связей
 Конвойер 410
 Конечный автомат 419, 596, 597, 598, 600, 602, 603, 703
 Консольный монитор 200, 201, 209, 631
 Конструктив 18, 201, 206, 478, 510, 546, 551, 552, 557, 561, 567, 575, 911
 Контекст
 ◊ доступа 765, 767, 773, 792
 ◊ нити 417, 427, 438
 ◊ прерывания 466, 609, 611, 615
 ◊ процесса 288, 321, 438, 442, 443, 449, 799, 918, 933, 946
 ◊ ядра 440, 466, 612
 Контроллер
 ◊ ПДП 353—355, 368, 504, 505, 616, 923
 ◊ прерываний 357, 476, 923, 924, 947, 950
 ◊ прямого доступа 353
 Контрольная сумма 69—72, 78, 166, 504, 544—547, 557, 696, 736, 932
 Контрольная точка 690, 693, 694, 874
 Контрольный код 72
 Конфликт репликации 401
 Кооперативная многозадачность 432, 435, 438, 466, 467, 470, 471, 569, 852
 Копирование при модификации 336
 Копирующий сборщик мусора 260
 Корневая файловая система 209, 210
 Корневая файловая система, Unix 635
 Корневой хаб 498

"Коробочное" ПО 757
 Косвенная адресация 951
Косвенно-регистровая адресация 111—113, 114, 955, 956
 Косвенный блок 664, 695
 Коэффициент масштабирования 361
 КПК (Карманный Персональный Компьютер) 499
 Кража идентичности 812
 Криптоанализ 74
 Криптография 72—78, 771
 Криптостойкость 74, 75, 77
 Критическая секция 378—380, 383—385, 393—396, 405, 428, 435, 461, 463, 466, 736, 737
 Кронос 190
 Кросс-разработка 18, 881, 915
 Куча 217, 220, 222, 223, 226, 236, 237, 242, 246, 259
 Кэш 35, 130, 132, 250, 251, 261, 272, 284, 286, 288, 296—298, 327—332, 358, 362—369, 372, 373, 400, 440, 442, 444, 466, 467, 468, 525, 526, 579, 612, 616, 623—627, 653, 655, 656, 661, 682, 688, 707, 708, 796, 901, 903, 909, 918, 921, 934, 962, 974

Л

Ленивое связывание 157
 Линия запроса прерывания 208, 356, 477, 514, 523, 715, 947, 948, 950
 Линк 406—408, 413—416, 429, 449, 450, 462, 468, 687
 Линкер См. Редактор связей
 Листинг 13, 139, 141, 676, 809, 882
 Литеральная адресация 108, 110, 123
 Лог 684, 687—689, 694, 695, 713, 716—719, 731, 746, 791, 839
 Логическая бомба См. Закладка
 Логические блоки 624, 648, 675, 682, 684, 686, 694, 696—698
 Логические линки 413
 Логический блок См. Кластер
 Логический диск 208, 209, 555, 556, 632, 637, 737
 Логическое адресное пространство 315
 Лог-файл 716

Локальный символ 184, 984
 Локальный флаг события 397

М

Магистраль 363, 368, 369, 508
 Магическая кавычка, PHP 806
 Магическое число 160, 165, 179, 180, 645—647, 657
 Магнитный диск 539
 Макровирус 810
 Макроопределение 80, 132, 133, 135, 142, 149, 231, 300, 619, 730, 731, 853, 855, 859, 992
 Макрос 43, 132, 133, 135, 180, 602, 619, 724, 796, 853, 992
 Малая сборка мусора 260
 Мандат 308—313, 346, 783
 Мантисса 56, 57, 981
 Манчестерская архитектура 103, 150, 882
 Маска сигналов, Unix 736
 Массивно-параллельные компьютеры 363
 Мастер шины 354
 Масштабируемость 35, 373, 379
 Машинный код 80, 88, 89, 139, 142, 149, 150, 153, 156, 814, 862, 988
 Машинный язык 79, 81, 82, 83, 87, 132, 155, 215, 281, 349, 375, 431, 473, 563, 631, 701, 755, 843, 917
 Менеджер
 ◊ банков 298
 ◊ оверлеев 127
 ◊ перекрытий 173
 ◊ событий 425, 426
 Мертвая блокировка 385—390, 396, 398, 402, 405, 406, 429
 Метаданные 251, 632, 681, 688, 689, 693—695, 699, 700, 855, 859, 894
 Метаданные, файловая система 632
 Метафайл 657
 Метка 135
 Микропрограммный автомат 84, 85, 149
 Микроядро 450, 465, 468, 469, 470, 586, 869, 870, 872
 Минорная запись 570, 572, 581
 Многозадачность 29, 30, 32, 431, 432, 435, 437, 468, 471, 900, 914
 Многопортовое ОЗУ 365

- Многопоточность 33, 392, 431, 465, 873, 876, 879, 890, 915
Многоточечные примитивы
 взаимодействия 406
Многоуровневая память 328
Многоядерный процессор 363, 369
Модальный диалог 714, 716, 736, 749, 792
Модемные линии 487
Моментальный снимок 690, 695, 700
Монитор 14, 20, 28, 63, 160, 200, 201, 202, 209, 403, 404, 407, 426, 531, 533, 564, 614, 631, 764, 834, 843, 853, 911, 969
Монитор ресурса 404
Мониторный процесс 404, 409, 416, 559
Монолитное ядро 466, 468, 469, 470, 869, 871, 875
Монтирование 633
Мост 508, 511, 512, 513, 514, 526, 527
Мультисессионный CD 653
Мутекс 395, 396, 397, 403, 429, 461—465, 469, 471, 711—713
Мягкое реальное время 37, 573
- Н**
- Наиболее подходящий 226
Наименее подходящий 226, 660
Наследование приоритета 462, 464, 471
Неразрушающее вычитание 59
Нереентерабельная программа 379
Несанкционированный доступ 757
Нить 28—30, 146, 157, 158, 257, 259, 264, 279, 303, 362, 363, 377—388, 391—400, 403—410, 413, 417, 425—438, 442, 443, 449—455, 461—471, 564, 569, 575, 586—594, 600, 602, 608—613, 617, 622, 623, 712, 713, 735—742, 844, 872, 944
Нить управления 157
Нить ядра 384, 466—469, 569, 613, 617, 872
Номер банка 125
- О**
- Обработчик 119, 146, 208, 250, 274, 285, 286, 294—296, 356—360, 375—378, 385, 392, 393, 398, 416—420, 425, 426, 438, 461, 466, 467, 468, 567, 570, 573, 587—589, 592—596, 602, 604, 607—719, 727, 733, 734—750, 753, 754, 853, 866, 892, 894, 930, 947, 948
Обработчик прерывания 208, 250, 356—358, 375—378, 385, 393, 466—468, 570, 587—589, 592—595, 602, 604, 609, 612, 613, 853, 947, 948
Обработчик событий 419, 425, 426, 744
Образ процесса 157, 158, 160, 162, 173, 267, 320, 832, 864, 865
Обратнаяпольская нотация 92
Объектная библиотека 186
Объектно-ориентированный язык 124, 143, 155, 156, 255, 265, 343, 344, 427, 732, 750, 879
Объектные библиотеки 186, 187, 644
Объектный модуль 175
Объектный файл 178—181, 187, 984, 985
Оверлей 172, 173, 174, 856
Одноадресная команда 90
Однопроходное ассемблирование 137, 138
Одноразовый блокнот 75
Одноуровневая память 338—340, 343
ОЗУ (Оперативное Запоминающее
 Устройство) 18, 19, 22, 27, 32, 35, 85, 87, 97, 99, 100, 104, 114, 136, 138, 139, 148, 172, 206, 220—223, 258, 267, 283, 284, 288, 292, 295, 296, 299, 327, 329, 332—335, 339, 340, 345, 346, 363, 365—369, 372, 384, 403, 476, 477, 503, 504, 509, 510, 513, 526, 547, 548, 561, 616, 651, 661, 676, 704, 707, 721, 723, 733, 846, 847, 849, 852—858, 863, 864, 874, 875, 887, 888, 891, 894, 901, 922, 929—935, 938, 947, 950
Операнд 87
Операнд команды 92, 97, 106, 107, 137, 297, 721, 951, 983, 988, 989
Оперативное представление данных 343
Операционная система См. ОС
Опережающее чтение 620, 624, 707
Опрос 350—358, 377, 391, 496, 501, 503, 548, 617, 719, 768, 792
Оптимальный код Хэмминга 72
Оптимистическая аллокация 338, 347
Оптический диск 16, 473, 527, 648, 718
Ортогональная система команд 101

ОС (Операционная Система) 13—44, 80, 82, 83, 137, 138, 147, 148, 155, 158, 174, 184, 189, 193, 200—220, 225, 226, 233, 249, 252, 253, 256, 265, 267, 275, 277, 281, 284—287, 290, 292, 294—301, 306, 315—320, 326, 327, 332, 336, 339, 341, 343, 346, 347, 359—363, 369, 377, 384, 396, 397, 403, 404, 409, 417, 426, 428, 434, 436—438, 441, 442, 449, 451, 452, 455—457, 464, 467, 468, 469—473, 477, 478, 502, 510, 514, 558, 563, 564—569, 574, 577, 581—587, 592, 603, 608, 610—622, 626—639, 642—644, 648, 652—654, 657, 659, 661, 662, 664, 667—672, 676, 677, 681, 685, 690, 692, 694, 698, 699, 703, 708, 710—715, 719—725, 729, 730, 733, 738, 744, 747, 760, 763, 766, 768, 772, 775, 777, 782, 785—787, 792, 794, 795, 797, 801, 807, 814, 819, 820—824, 832, 833, 838—871, 876—885, 890—904, 908—930, 936, 938, 943, 946—950
 ОС встраиваемая 19
 ОС общего назначения 13, 15, 19, 39, 40, 43, 233, 315, 341, 470, 564, 622
 Особая ситуация ошибки сегментации 285
 Остановов системы 295, 669, 746
 Отказ 565
 Откат 680
 Откат транзакции 404
 Отладочная информация 139, 141—143, 148—151, 161, 178, 213, 993
 Отладчик 138, 139, 141, 142, 146—153, 268, 430, 675, 724, 735, 746, 749, 750, 782, 822, 915, 956, 983, 984
 Отладчик ядра 139
 Отложенная запись 547, 554, 620, 621—627, 641, 670—674, 678, 707
 Отложенное связывание 157
 Относительная адресация 122—124, 168
 Относительная загрузка 162, 176, 192, 336, 843
 Относительный загрузочный модуль 165
 Отображение файлов на память 220
 Отображение цветов 538
 Отображенный на память ввод/вывод 475, 476, 561, 852
 Очередь
 ♦ нитей 395, 434

♦ процессов 434, 444
 ♦ сообщений 407, 408, 426, 429, 713, 782, 868, 890, 915
 Ошибка 15, 16, 20, 22, 24, 25, 54—57, 69—72, 104, 106, 118, 135, 138, 139, 147, 148, 159, 179, 198, 203, 219, 230, 252—254, 269, 272—274, 277, 281, 285, 286, 292, 294—296, 300, 307, 308, 314, 315, 326, 332, 340—342, 353, 359, 362, 376—379, 386, 392—395, 405—412, 428, 436, 462, 477, 487, 492, 494, 501, 504, 548—552, 557, 564, 576, 581, 583, 587—592, 596, 603, 607, 614, 617—621, 624, 634, 637, 638, 640, 643, 661, 665, 673—675, 680, 681, 691, 701—759, 769, 778, 791, 793, 795, 797, 798, 801—812, 815, 817—822, 831, 832, 833, 835, 838—842, 868, 877, 902, 905, 908, 910, 912, 928, 935, 950, 982, 984, 991
 ♦ подъема по каталогам 808, 832, 842
 ♦ потерянного пробуждения 392, 393, 736
 ♦ соревнования 377, 379, 405, 428, 723, 807
 ♦ шины 104, 359, 614, 721, 733, 935

П

Пакетное приложение 349
 Палитра 538
 Папка 631, 646
 Параллельная шина 505, 528
 Параллельный порт 485, 486, 498, 923, 948
 Пароль 161, 212, 760, 764, 767—772, 785—795, 802, 811, 817, 833, 836, 841
 Пассивный хаб (USB) 498
 ПДП (Прямой Доступ к Памяти)
 353—357, 477, 496, 504, 526, 547, 548, 552, 584, 585, 589, 608, 616
 Первичный загрузчик 200, 202, 208, 209, 812, 834, 887, 923
 Первый отдел 15
 Первый подходящий 225, 226, 231
 Передача с негарантированной доставкой 407
 Перекрытие См. *Оверлей*
 Перемычка 949

- Перенос 23, 42, 47—52, 57, 59, 60, 90, 94, 95, 175, 260, 272, 310, 329, 342, 344, 347, 367, 369, 435, 453, 466, 470, 471, 489, 490, 537, 551, 582, 653, 666, 677, 864, 875, 888, 890, 902, 915, 929, 932, 933, 965, 966, 968, 971, 972
- Переполнение 49, 50, 94, 95, 119, 124, 133, 295, 299, 337, 359, 463, 494, 703, 722, 798, 800, 803, 932, 962, 975
- Переполнение буфера 463, 494, 703, 798, 803
- Переполнение стека 722
- Периферийная шина 206, 369, 478, 502, 505, 509, 512, 528, 538, 561, 567, 614, 616, 619, 874, 879, 923, 949, 950
- Периферийное устройство 14—24, 368; 473, 476, 509, 510, 526, 528, 614, 629, 703, 707, 719—723, 878, 881, 931, 948
- Периферийные процессоры 352, 358, 561
- Песочница 829, 831, 832
- ПЗУ (Постоянное Запоминающее Устройство) 16, 18, 19, 85, 104, 136, 155, 189, 200—204, 208, 211, 468, 476, 514, 548, 567, 568, 614, 629, 874, 883, 912, 915, 918, 923, 924
- Пиксель 62, 63, 67, 535, 536, 537, 538, 561
- Плавающая точка 54—58
- Планировщик 432
- Плиточная память 891, 893, 939
- ПЛМ (Программируемые Логические Матрицы) 14, 84, 858
- Плохой блок 691, 692, 693
- ПО (Программное обеспечение) 25, 27, 35, 36, 43, 57, 272, 462, 638, 653, 705, 757, 832, 839, 840, 847, 850, 860, 869, 870, 872, 878, 898, 904
- Подкачка 22, 139, 281, 289, 319, 327, 360, 456, 625, 847, 855, 890, 900, 919
- Подсистема ввода/вывода 473
- Подсчет ссылок 255, 256, 257, 279, 711
- Подтверждение транзакции 401
- Позднее/динамическое связывание 155, 156, 157, 194, 800
- Позиционно-независимый код 168, 169, 170, 320, 336
- Поиск жертвы 296, 311, 329, 330—334, 346, 612, 900
- Поиск ◊ наиболее подходящего 226 ◊ наименее подходящего 226 ◊ первого подходящего 225
- Полнодуплексный порт 484, 507
- Полномочие 207, 277, 291, 703, 772, 773, 781—787, 790—794, 817, 829, 832, 841, 937
- Полномочия 772, 783, 784, 787, 794
- Полностью упорядоченный доступ 367
- Полудуплексный порт 484, 508
- Пользовательские нити 430, 464—466
- Пользовательский контекст 465, 466, 584, 587, 590, 595, 609, 611
- Порт ввода 24, 480, 481, 482, 487, 513, 525, 585, 931, 938, 969
- Порт ввода/вывода 24, 481, 482, 487, 525, 931, 938, 969
- Порт вывода 478, 479, 480, 481
- Порты завершения 622
- Порядок 37, 56, 57—61, 73, 79, 84, 85, 101, 105, 111, 113, 119, 126, 130, 142, 186, 202—205, 213, 221—224, 267—269, 295, 299, 301, 317, 326, 340, 343, 347, 358, 367, 379, 388, 402, 419, 441, 450, 463, 466, 473, 504, 576, 619, 622, 623, 628, 631, 650, 658, 659, 677, 689, 697, 705, 718, 732, 741, 742, 746, 769, 810, 819, 822, 831, 833, 838, 850, 884, 902, 913, 925, 935, 949, 977, 981—984
- Последовательная шина 486, 497, 509, 528, 538, 552, 577, 947
- Последовательный порт 19, 104, 148, 485, 486, 488, 492, 494, 496, 506, 511, 538, 544, 564, 571, 577, 582, 923
- Посмертный дамп 138, 141, 147, 253, 720, 735
- Потерянный блок 673
- Поток 37, 65—69, 75, 76, 81, 149, 157, 265, 353, 361, 362, 364—367, 375—378, 392, 396, 405—410, 417, 429, 431, 433, 437, 455, 457, 461, 484, 536, 541, 542, 572, 573, 577, 578, 602, 609, 617, 623, 641, 643, 647, 731, 804, 808, 813, 865, 884, 917, 921
- Потоковые устройства, Unix 574
- Потолок приоритета 463
- Почтовый ящик 406, 413, 563, 794

- Пошаговое исполнение 146, 147, 153
 Преобразование Фурье 57, 67, 68, 84
 Прерывание 30, 95, 119, 147, 167, 201,
 208, 216, 297, 346, 356—360, 375, 376,
 377, 385, 392, 393, 398, 416, 417, 437,
 438, 441, 443, 466—468, 476, 488, 495,
 496, 500—504, 526, 548—550, 564, 568,
 584, 585—589, 592, 593, 596, 602—604,
 609—619, 671, 721, 723, 733, 734, 853,
 919—924, 930—933, 945—950, 962, 973
 Примитивы взаимоисключения 381—388,
 405, 844, 866, 890
 Приоритет 451
 Приоритет процесса 438, 451, 455, 459,
 854
 Приоритетный арбитраж шины 368
 Проблема голодного философа 389, 391,
 406
 Провайдер приложений 23, 36
 Проводник 212, 497, 528, 552
 Программа 157
 Программатор 18, 912
 Программная секция 179
 Программное обеспечение 25, 27, 35, 36,
 43, 57, 272, 462, 638, 653, 705, 757, 832,
 839, 840, 847, 850, 860, 869, 870, 872,
 878, 898, 904
 Программный канал 409
 Производитель-потребитель 396
 Пролог прерывания 947
 Промах TLB 288
 Промах кэша 362, 367
 Промывание кэша 328, 918
 Просмотр ссылок 255—259, 279
 Пространство имен 308, 633, 634—636,
 865
 Процесс 17, 19, 21, 28, 29, 31, 32, 34, 73,
 119, 123, 129, 136, 138, 155—162, 174,
 175, 187, 189, 198, 203, 207—216, 249,
 250, 253, 260, 265, 267, 273—277, 281,
 284, 286, 288, 295—299, 304, 307—323,
 327, 329, 332, 333, 336, 338, 353, 357,
 359, 363, 364, 369, 375, 377, 380, 387,
 392, 395, 397, 399, 403, 404, 409—418,
 427, 429, 430, 434, 438—452, 455—470,
 476, 514, 559, 563—566, 569, 570, 572,
 576, 582, 583, 585, 586, 589, 592, 593,
 596, 600, 602, 608—613, 617, 618, 625,
 627, 628, 638, 641, 674, 675, 696, 703,
 711—713, 718—720, 725, 731—736, 741,
 754, 765, 767, 772, 781, 784—801, 815—
 818, 826, 833, 838—841, 850, 854, 856,
 859, 863—869, 872—876, 884, 900, 901,
 905, 908, 918, 921, 933, 946
 Процессор 14, 17, 18, 19, 22, 24—32, 35,
 44, 47, 50—60, 80—87, 90—114, 121—
 125, 128—132, 146—153, 158, 160, 169,
 174, 189—192, 200, 209—215, 228, 261,
 272—276, 283—293, 295—298, 300, 306,
 310, 312, 314, 326, 334, 335, 344, 345,
 346, 350—378, 383—385, 391, 392, 413,
 425, 428, 431, 434—444, 449—456, 461,
 467, 468, 473—481, 497, 505, 508—511,
 514, 526, 538, 548—552, 557, 561, 565,
 567, 612, 614, 619, 624, 635, 638, 645,
 646, 651, 682, 703, 714, 721, 723, 731,
 733, 735, 737, 748, 800, 801, 803, 807,
 844—852, 855, 858, 860—862, 866, 870—
 878, 881—895, 900, 902, 905, 911—937,
 943—952, 956, 985, 988, 989
 Прямой доступ к памяти (ПДП) См. DMA
 Псевдоустройство 563, 574, 585, 628, 632,
 874
 Пул 217, 220, 231, 232, 249, 250, 259,
 260—264, 330—332, 427, 429, 496, 612,
 624, 692, 697, 798, 800
 Пул нитей 427
 Путевое имя 199, 214, 318, 326, 632, 666

P

- Рабочее множество 297, 332, 334, 335,
 793, 901
 Рабочие нити 427
 Разбрасывание-сборка 505
 Развернутый автомат 600
 Разветвитель 498
 Раздел диска 208, 335, 675
 Разделение времени 437, 451
 Разделенное время 29—33, 36, 37, 361,
 417, 438, 451, 452, 455—459, 471, 854,
 855, 864
 Разделы памяти 161, 162, 213, 267, 843,
 854
 Разделяемая библиотека 169, 187, 189,
 193, 220, 256, 281, 289, 315—327, 336,

- 346, 644, 667, 703, 785, 810, 848, 855, 867, 876, 902
Размер точки 533
Разрешение растрового изображения 62
Разрядность процессора 47, 95
Раннее/статическое связывание 155—157, 176, 194, 213, 469
Распараллеливание 361
Распределение Зипфа 328
Растр 62
Растровое изображение 43, 60, 62—67, 534, 739
Расширение файла 643, 645, 646
Расширенные атрибуты 646, 647, 652, 660, 685, 687, 688, 774
Расширитель DOS 213, 885, 886, 915
Расширитель адреса 125
Расщепленная транзакция (PCI) 526
Реальное время 16, 36—39, 189, 247, 249, 261, 288, 304, 346, 349, 385, 398, 407, 417, 437, 438, 451, 456, 461—471, 484, 500, 573, 583, 585, 592, 617, 621—623, 669, 676, 785, 848, 854, 855, 863, 869, 872, 876, 879, 923, 948
Регистр 80, 84, 85, 88, 90—132, 138, 153, 163—169, 174, 176, 265, 266, 272—277, 283, 284, 290—298, 301, 307, 322, 327, 339, 340, 341, 349, 350, 353, 356—358, 367, 378, 379, 384, 434, 438—442, 448, 449, 464, 471, 475—481, 488—496, 502, 503, 512—514, 525, 526, 538, 546—552, 561, 564, 593, 609, 612—615, 619, 637—639, 652, 704, 708, 714, 715, 721, 736—742, 769, 843, 846, 848, 852, 857—859, 919, 920—922, 927—935, 943—947, 951—957, 963—974, 977, 982, 984, 988
Регистр направления данных 481
Регистр общего назначения 97—101, 122, 130, 153, 266, 277, 290, 301, 438, 439, 442, 449, 852, 858, 920, 922, 927—931, 945, 954—956, 988
Регистрация намерений 677
Регистр-защелка 480
Регистровая адресация 857, 956
Редактирование связей См. Сборка
Редактор связей 156, 175, 178, 185—189, 193, 205, 321, 323, 326, 586, 644, 856, 984, 990
Реентерабельная/реентрантная программа 380
Реентерабельность 587, 612, 613, 864
Режим адресации 105—124
Режим адресации 87, 106—114, 121—123, 153, 163, 169, 274, 852, 857—860, 920, 951—957, 988
Резидентная часть 174
Резидентное ядро 174
Ретенция логов 717, 718
Робастность 712
Ротационная задержка 553
Ротация логов 839
Руткит 207
Ручка 270
- ## С
- Сборка 25, 155—157, 160, 162, 168, 174—179, 188, 189, 192, 193, 196, 198, 205, 213, 223, 252, 257, 264, 265, 315, 318—321 336, 346, 400, 469, 505, 547, 560, 568, 586, 612, 643, 704—711, 746, 810, 818, 820, 854, 856, 888, 890, 895, 900, 915, 922, 983—989, 991
Сборка в момент загрузки 188, 189, 193, 205, 315, 336, 890, 915
Сборка мусора 223, 252—265, 271, 279, 346, 400, 612, 705, 707, 711, 732
Сборка мусора подсчетом ссылок 255
Сборка мусора просмотром ссылок 256
Сброс 22, 28, 29, 90, 119, 200, 288, 290, 294, 295, 297, 298, 311, 314, 332, 393, 462, 463, 500, 512, 514, 523, 548, 617, 709, 710, 864, 888, 903, 936
Сброс окна 119
СВМ (Система Виртуальных Машин) 24—26, 359
Свободный доступ к памяти 367
Свопинг 275, 281, 327, 872
Своп-пространство 320, 335, 341, 345, 707
Своп-файл 319, 335, 336, 337, 876
Связывание См. Сборка
Связь, файловые системы 664
Сегмент отката 344, 401, 680, 689
Сегмент памяти 282
Сегментная адресация 282, 286, 288, 289, 306

- Сегментно-страничная трансляция 274
 Сегментный отказ 306, 890, 919
 Сектор диска 544, 553
 Секции, ассемблер 986
 Селектор банка 125, 128, 129, 538
 Селектор сегмента 282, 305—308, 340, 783, 891, 928, 931, 933, 936—938, 943, 945, 946
 Селектор страницы 282, 284, 293, 294
 Семафор
 ◊ Дейкстры 394, 398, 429
 ◊ общего вида 395
 ◊ счетчик 395, 429, 463, 464, 711, 712
 Сервер транзакций 363, 403, 404, 559, 560, 758, 845
 Серверные агенты, Lotus Notes/Domino 790
 Сервис 15, 23—29, 32, 40, 194, 201—214, 252, 290, 320, 332, 341, 343, 377, 385, 395, 404, 408, 413, 455, 458, 468, 514, 558, 567, 568, 574, 578, 580, 608, 612, 613, 699, 706, 709, 718, 719, 756, 757—759, 767, 792, 796, 800—803, 807—809, 813, 818—821, 826, 833, 835, 839, 841, 842, 859, 863, 875, 880, 883, 886, 905, 909, 924, 948
 Сервисы, Win32 767
 Сериализация 441—443, 573, 921
 Сессия 26, 77, 158, 342, 404, 427, 568, 647, 653, 688, 734, 765, 766, 791, 855, 867, 890, 893, 914, 940
 Сигнал 30, 31, 54, 57, 67, 68, 85, 350, 356, 357, 358, 366, 369, 387, 392, 393, 398, 414, 434, 437, 438, 450, 462, 465, 476, 479, 482—487, 493, 497, 499, 500, 505, 506, 510, 523—526, 536, 540, 544, 549—551, 579, 584, 592, 593, 617, 618, 672, 727, 733—744, 747, 785, 867, 947
 Сигнал запроса прерывания 592
 Сигнал прерывания 356
 Сигнал, Unix 733
 Сигнатура 165, 180, 646
 Символическая связь 326, 652, 665, 666, 667, 685, 688, 699, 865, 894
 Символьные устройства, Unix 571
 Символьный отладчик 161, 983
 Симметричная многопроцессорность 355, 844, 876
 Симметричная многоточечность 406
 Симплексный порт 484
 Синий экран смерти 708
 Синхронизация 37, 363, 365, 368, 391, 392, 396—407, 413, 432, 443, 463, 465, 479, 499, 500, 501, 510, 528, 540, 544, 552, 587, 622, 628, 688, 727
 Синхронная обработка сообщений 426
 Синхронная передача 407, 482, 484, 487
 Синхронное взаимодействие 378
 Синхронные исключительные ситуации 706
 Синхронный ввод/вывод 609
 Сирота, инод 674
 Система Вернама 75, 76
 Система команд 53, 57, 60, 62, 82, 83, 88, 90—110, 119—132, 152, 153, 189—192, 326, 349, 352, 359, 370, 442, 477, 582, 820, 848, 852, 857, 858, 862, 895, 917—922, 926—931, 951, 956
 Система разделенного времени 349
 Система управления версиями 401, 639
 Система, управляемая событиями 416
 Системная магистраль 368
 Системная шина 192, 352—354, 363, 366, 368, 409, 505, 506, 509, 511, 512, 528, 538, 547, 552, 561, 572, 614, 721, 723, 733, 947, 950
 Системные данные 662, 667, 675, 680, 681, 690
 Системные нити 430, 466, 468
 Системный администратор 205, 320, 326, 332, 681, 719, 764, 783, 786, 795—797, 810, 824, 840, 841, 909, 924
 Системный вызов 22, 24, 28, 40, 44, 135, 158, 193, 207, 218, 270, 276, 277, 279, 289, 290, 292, 297, 298, 300, 307, 309, 314, 337, 338, 339, 340, 357, 406, 410, 412, 413, 417, 428, 434, 435, 455, 457, 463, 465, 466, 467, 470, 565, 572, 573, 577, 588, 590, 591, 595, 608, 612, 618, 622, 629, 633, 638, 641, 662, 664, 718, 730—741, 744, 745, 785, 787, 788, 823, 834, 846, 849—855, 858, 859, 863—868, 879, 884, 886, 888, 890, 894, 902, 914, 916, 938, 947
 Системный реестр, Win32 901
 Скрипт 319, 327, 794, 801, 805, 806, 807, 817, 820, 831

- Слаб 249
 Слабовый аллокатор 249, 250
 Слабый символ 179, 184
 Слайс 209, 570, 632
 Словарная атака 759, 768, 769, 772, 794, 833, 841
 Слово 17, 36, 45, 49, 60, 65, 66, 73—76, 82, 85, 87, 90, 91, 94, 96—109, 122, 124, 136, 138, 146, 152, 156, 174, 180, 198, 201, 228, 229, 230, 231, 265, 266, 272, 277, 290—304, 308, 310, 311, 335, 346, 353, 356, 357, 359, 361, 362, 413, 414, 431, 438, 441, 449, 454, 466, 470, 488, 489, 490, 491, 503, 512, 528, 548, 563, 571, 619, 627, 631, 664, 669, 693, 697, 703, 716, 721, 723, 726, 733, 736, 740, 743, 746, 767, 768, 769, 771, 782, 816, 846, 848, 849, 851, 855—857, 882, 892, 893, 901, 922, 927, 931, 932, 933, 935, 945, 946, 947, 950, 951, 964, 967, 972, 979, 980, 988, 989
 Служба каталогов 775, 903, 904
 Смена идентичности 784, 785, 787, 791
 Снiffeр 569, 756, 758
 Событие 417
 Совместимость по языку ассемблера 83, 98, 927
 Совместимость снизу вверх 83
 Согласованность 729
 Сокет 118, 408, 410, 413, 574, 718, 734, 868, 903
 Сообщения 417
 Сопрограмма 435, 466, 742
 Сопроцессор 98, 99, 359, 360, 439, 443, 858, 927—933, 948, 954—957, 976—981
 Социальная инженерия 759, 760
 Спинлок 384, 429, 442, 444, 467
 Список контроля доступа 662, 759, 772—774, 783
 Список свободных блоков 228, 229, 233, 260, 657, 659, 672, 675, 676, 697
 Справедливый арбитраж шины 368
 Спулинг 627, 628
 Спупинг 758, 829
 Среда исполнения ЯВУ 226
 Среда кросс-разработки 18
 Срыв буфера 315, 708, 759, 797, 798, 801—805, 821, 832, 833, 835, 842, 905, 909
 Срыв стека 81, 722, 797, 798, 799, 800, 801; 819, 834, 842
 Статическое выделение памяти 220
 Статическое связывание 155
 Стек 81, 90, 91, 92, 97, 101, 111—117, 130, 132, 146, 161, 167, 216, 217, 259, 277, 289, 295, 297, 301—305, 323, 356, 357, 413, 427, 432—434, 438, 443, 447—450, 467, 561, 596, 600, 722, 741, 742, 747—753, 797—801, 814, 819, 834, 842, 844, 852, 859, 864, 867, 876, 884, 888, 894, 903, 915, 928, 929, 933, 937, 944—947, 954, 956, 964, 965, 967—969, 976—981
 Стековая архитектура 92, 93, 113, 153
 Стековый кадр 114—119, 434, 741, 742, 746, 747, 754, 799, 800—803, 956, 964
 Сторожевая зона 217
 Сторожевой таймер 90, 462, 709, 723
 Страница памяти 282
 Страницчная адресация 277, 282, 287, 315
 Страницчная подкачка 21, 319, 346, 844, 849, 891, 893, 900, 901, 921
 Страницный отказ 286, 328—336, 346, 359, 613, 625, 626, 866, 894, 920, 934, 950
 Стратегическая функция 564, 610, 611
 Стриппинг 555
 Строб 482
 Стока кэша 367, 526
 Строчный гасящий импульс 532
 Субаллокация 649, 650, 654, 656, 698
 Суперблок 657, 661, 662, 672, 683, 684, 688, 689, 694
 Супервизор 290, 292, 294, 298, 301, 309, 346, 778, 791, 841, 942
 Суперскалярный процессор 80, 86, 272, 358, 361, 362, 367, 441, 565, 921, 947
 Счетчик команд 90, 96, 97, 99, 108, 122—125, 168, 169, 302, 356, 359, 375, 438, 441, 442, 449, 852, 919, 927, 931
- T**
- Таблица
 ◊ векторов прерываний 297, 356, 733, 882
 ◊ глобальных символов 178
 ◊ дескрипторов 240, 290, 298, 303—307, 310, 359, 920, 933, 936, 941, 944
 ◊ импорта 178

Таблица (*прод.*):

- ◊ инодов 623, 657, 661, 662, 665, 683—685, 688, 692—694, 699, 793
- ◊ перекрестных ссылок 152, 176
- ◊ перемещений 165, 166, 176, 178, 184, 185, 213, 224, 319
- ◊ разделов диска 208
- ◊ символов 176, 183, 184, 983, 984
- ◊ страниц 304, 440, 942
- ◊ трансляции 283—290, 294, 296, 298, 306, 345, 721, 858, 942, 943
- ◊ экспорта 178

Табулятор 85, 86, 796

Тег 308

Текстовый файл 139, 211, 582, 642, 643, 646, 765, 855, 884

Том 19, 22, 25, 27, 31, 36, 38, 40, 45, 57, 74, 82, 85, 86, 88, 90, 97, 101, 102, 111, 123, 126—133, 139, 142, 153, 157, 162, 165, 169, 172, 187, 193, 199—203, 208, 212, 215, 220, 227, 231, 247, 250, 252, 264, 267, 268, 274, 275, 286, 295, 298, 305, 308, 309, 315—318, 322, 332, 334, 339, 341, 349, 357, 362, 364, 367, 384—388, 418, 419, 425, 427, 443, 458, 460, 463, 464, 469, 483, 486, 512, 527, 535, 549, 583, 584, 596, 609, 611, 614, 618, 623, 631, 633, 634, 639, 650—658, 665, 669, 671, 675—689, 693, 696, 701, 724, 737, 749, 761, 762, 766, 769, 777, 778, 791, 794, 796, 802, 814, 820, 832, 833, 849, 850, 855, 859, 866, 868, 870, 871, 876, 877, 879, 881, 883, 884, 891, 894, 897, 898, 905, 908, 909—915, 926, 935, 936, 937

Точка монтирования 635

Точка останова 146—148, 153, 735, 822, 962

Точка согласования, WAFL 695

Транзакция 328, 379, 388, 400—406, 500—504, 523—526, 559, 677—680, 684, 687—690, 694, 695, 699, 700, 900, 906

Транзакция (PCI) 523

Транзакция (USB) 500

Трехадресная команда 90

Тристабильные выходы 481

Тројанская программа 207, 268, 629, 759, 795, 796, 801, 809, 810—813, 820—830, 837—842, 880, 909

Труба 406—413, 429, 563, 574, 641, 688, 713, 718, 734, 866

Трэм 415

Трэшинг 334

ТТЛ-совместимые напряжения 478

У

Удаленная загрузка 204

Удаленное исполнение кода 905

Указатель стека 111

Упаковка без потерь 65

Упаковка данных 64, 66, 67, 72

Упаковка с потерями 65, 98

Условно-бесплатная программа 836

Условные точки останова 147, 724

Устойчивость к сбоям 655, 662, 668, 670

Устройство ввода 416, 474, 486, 561, 563, 572—574, 577

Устройство ввода/вывода 474, 561, 563, 572—574

Устройство вывода 474, 561, 574, 582, 882

Утечка памяти 252, 705, 723

Уточнение 136, 256, 313, 600, 727

Уточнения i432 313

УУП (Устройство Управления Памятью)
283, 284, 290, 293—298, 309, 314

Учетная запись 23, 759, 765, 766, 767, 772, 775, 786, 790—792, 806, 817, 818, 825, 828, 833, 838, 839, 841, 859, 875, 900, 901

Ф

Файл 44, 64, 100, 101, 105, 114, 119, 120, 136, 137—143, 156—160, 165—167, 174, 176—187, 193, 196—198, 202, 205—212, 218, 220, 233, 250, 256, 257, 299, 300, 304, 319, 325, 328, 329, 332, 335—339, 342—345, 347, 402, 403, 410—413, 430, 439, 449, 453, 454, 469, 554, 559, 569, 571, 582, 618, 623, 624, 626, 628, 631—735, 746, 748, 765, 766, 772, 774, 777, 780—793, 796, 801—810, 822—832, 839, 844, 847, 849—859, 863—866, 870, 874, 876, 880, 884, 890, 894, 896, 900—908, 912, 984—986, 990—992

Файл подкачки 139

Файловая запись (fnode) 659

Файловая система (ФС) 17, 19, 193, 202—214, 225, 226, 256, 327, 340, 341, 468, 546, 554, 557, 558, 579, 581—585, 618, 619, 623—625, 632—641, 647, 648—657, 661—708, 725, 727, 774, 777, 781, 782, 793, 804, 808, 825, 826, 842, 850—852, 855, 858, 859, 865, 866, 870, 874, 875, 884—886, 891, 894, 896, 900, 901, 906
 Файловый сервер 29, 450, 559, 631, 632, 634, 635, 647, 665, 694, 727, 804, 805, 808, 904, 905
 Физическая адресация 103
 Физические линки 413
 Физическое адресное пространство 129, 275, 294, 413, 475, 476, 852, 856, 935, 936, 943
 Фиксированная адресация 477, 561
 Фиксированная точка 53, 54
 Финитность 597
 Флаг загрязнения 633, 672
 Флаг события 397, 398, 418, 467, 622, 855
 Флэш-накопитель 498, 499, 500
 Флэш-память 16, 18, 104, 338, 340, 473, 498, 539, 760
 Форматирование диска 546, 553, 554
 Фрагментация 221—226, 247, 266—271, 275, 287, 288, 341, 649, 651, 656, 685
 Фрагментация памяти 221
 Фрейм 483

X

Хаб 498—502, 561
 Холодная перезагрузка 618, 674
 Холостая нить 435
 Холостой цикл 384, 435, 732
 Хостинг-провайдер 794
 Хранимое представление данных 343
 Хранимые данные 344, 559, 708
 Хэш 771
 Хэширование 771
 Хеш-таблица 36, 184, 226, 284, 285, 289, 296, 906

Ц

Цветовая глубина 63
 Целевое устройство 488, 524, 526, 528, 559, 575, 576

Целостность 71, 78, 207, 254, 341, 343, 376, 378, 428, 546, 609, 625, 633, 669—672, 680, 681, 699, 761, 805
 Центральный арбитр шины 505
 Цикл шины 353, 366, 507, 514, 935, 990
 Цилиндр диска 544
 ЦОС (Цифровая Обработка Сигналов) 57
 ЦПУ (Центральное Процессорное Устройство) 27, 28, 32, 296, 365, 439, 445, 475, 526, 538, 548, 551, 553, 557, 592, 607, 895

Ч

Частично упорядоченный доступ 367
 Частота кадровой развертки 531
 Частота строчной развертки 531
 Чекпойнт См. Контрольная точка
 Червь 149, 207, 716, 757, 758, 762, 768—772, 800, 803, 807, 813, 818, 819, 833—840, 897, 905, 909
 Червь Морриса 768—772, 803, 818, 833, 909
 Чередование 553, 554
 Черные списки 836
 Честное планирование 458
 Чипсет 502, 950
 Числа одинарной и двойной точности 57
 Число с плавающей точкой 56, 57, 61, 88, 98, 105, 343, 740, 889, 928, 931, 935, 990, 991
 Число с фиксированной десятичной точкой 54
 Число с фиксированной точкой 97

Ш

Шина 95, 99, 103, 104, 128, 192, 206, 339, 352—354, 357, 359, 361—373, 383—387, 409, 442, 462, 467, 475—481, 486, 496—502, 505—514, 523—529, 538, 544, 547, 551—553, 558, 561, 567, 572, 577, 579, 614—619, 709, 721, 723, 733, 832, 858, 874, 878, 879, 895, 896, 923, 931, 935, 947—950, 964, 975, 990
 Шинная архитектура 368
 Шифрование 73—77, 268, 585, 764, 771, 773, 783, 907

Шлюз 305, 719, 819, 919, 937, 945—947

Шлюз (x86) 305, 946

Шпиндель 539

Э

Экстент 658—661, 664, 684—688

Электронная подпись 78, 764, 773, 790, 821, 829

Электронно-лучевая трубка (ЭЛТ) 531, 532, 355, 536

Эмулятор 18, 19, 25, 27, 33, 42, 655, 876, 880, 886, 887, 893, 897, 915, 922, 930, 936, 938, 947

Эпилог прерывания 357, 615, 948

ЭСППЗУ (Электронно-Стираемое Программируемое ПЗУ) 258, 473, 631

Эффект плацебо 69

Я

ЯВУ (Язык Высокого Уровня) 79, 81, 85, 102, 135, 137, 142, 146, 147, 149, 432, 619, 705, 732, 747, 889, 964

Ядро 19, 23—27, 32, 37, 38, 80, 132, 139, 148, 167, 174, 199, 202—214, 220, 249—252, 256, 268, 273, 277, 281, 285, 290, 292, 297—307, 314, 317, 332, 341, 357, 359, 362, 363, 369, 384, 397, 417, 418, 439—442, 464—471, 477, 563—569, 578—586, 608, 609, 612—621, 642, 645, 680, 682, 697, 708, 710, 714, 725, 731—737, 746, 784, 785, 788—801, 804, 810, 819, 823—826, 832, 837, 844, 847, 849, 853—859, 864, 865, 866—880, 884, 888, 890—903, 907—915, 922, 924, 925, 930, 935, 938, 939, 941, 948

Язык ассемблера 79, 82, 132, 917, 982